File 1 - __init__.py:

```
1: (0)              """
2: (0)              An object-oriented plotting library.
3: (0)              A procedural interface is provided by the companion pyplot module,
4: (0)              which may be imported directly, e.g.::
5: (4)                  import matplotlib.pyplot as plt
6: (0)              or using ipython::
7: (4)                  ipython
8: (0)              at your terminal, followed by::
9: (4)                  In [1]: %matplotlib
10: (4)                 In [2]: import matplotlib.pyplot as plt
11: (0)             at the ipython shell prompt.
12: (0)             For the most part, direct use of the explicit object-oriented library is
13: (0)             encouraged when programming; the implicit pyplot interface is primarily for
14: (0)             working interactively. The exceptions to this suggestion are the pyplot
15: (0)             functions `.pyplot.figure`, `.pyplot.subplot`, `.pyplot.subplots`, and
16: (0)             `.pyplot.savefig`, which can greatly simplify scripting.  See
17: (0)             :ref:`api_interfaces` for an explanation of the tradeoffs between the implicit
18: (0)             and explicit interfaces.
19: (0)             Modules include:
20: (0)             :mod:`matplotlib.axes`
21: (4)                 The `~.axes.Axes` class.  Most pyplot functions are wrappers for
22: (4)                 `~.axes.Axes` methods.  The axes module is the highest level of OO
23: (4)                 access to the library.
24: (0)             :mod:`matplotlib.figure`
25: (4)                 The `.Figure` class.
26: (0)             :mod:`matplotlib.artist`
27: (4)                 The `.Artist` base class for all classes that draw things.
28: (0)             :mod:`matplotlib.lines`
29: (4)                 The `.Line2D` class for drawing lines and markers.
30: (0)             :mod:`matplotlib.patches`
31: (4)                 Classes for drawing polygons.
32: (0)             :mod:`matplotlib.text`
33: (4)                 The `.Text` and `.Annotation` classes.
34: (0)             :mod:`matplotlib.image`
35: (4)                 The `.AxesImage` and `.FigureImage` classes.
36: (0)             :mod:`matplotlib.collections`
37: (4)                 Classes for efficient drawing of groups of lines or polygons.
38: (0)             :mod:`matplotlib.colors`
39: (4)                 Color specifications and making colormaps.
40: (0)             :mod:`matplotlib.cm`
41: (4)                 Colormaps, and the `.ScalarMappable` mixin class for providing color
42: (4)                 mapping functionality to other classes.
43: (0)             :mod:`matplotlib.ticker`
44: (4)                 Calculation of tick mark locations and formatting of tick labels.
45: (0)             :mod:`matplotlib.backends`
46: (4)                 A subpackage with modules for various GUI libraries and output formats.
47: (0)             The base matplotlib namespace includes:
48: (0)             `~matplotlib.rcParams`
49: (4)                 Default configuration settings; their defaults may be overridden using
50: (4)                 a :file:`matplotlibrc` file.
51: (0)             `~matplotlib.use`
52: (4)                 Setting the Matplotlib backend.  This should be called before any
53: (4)                 figure is created, because it is not possible to switch between
54: (4)                 different GUI backends after that.
55: (0)             The following environment variables can be used to customize the behavior:
56: (0)             :envvar:`MPLBACKEND`
57: (4)                 This optional variable can be set to choose the Matplotlib backend. See
58: (4)                 :ref:`what-is-a-backend`.
59: (0)             :envvar:`MPLCONFIGDIR`
60: (4)                 This is the directory used to store user customizations to
61: (4)                 Matplotlib, as well as some caches to improve performance. If
62: (4)                 :envvar:`MPLCONFIGDIR` is not defined, :file:`{HOME}/.config/matplotlib`
63: (4)                 and :file:`{HOME}/.cache/matplotlib` are used on Linux, and
64: (4)                 :file:`{HOME}/.matplotlib` on other platforms, if they are
65: (4)                 writable. Otherwise, the Python standard library's `tempfile.gettempdir`
66: (4)                 is used to find a base directory in which the :file:`matplotlib`
67: (4)                 subdirectory is created.
```

```
(0)             Matplotlib was initially written by John D. Hunter (1968-2012) and is now
(0)             developed and maintained by a host of others.
(0)             Occasionally the internal documentation (python docstrings) will refer
(0)             to MATLAB®, a registered trademark of The MathWorks, Inc.
(0)             """
(0)             __all__ = [
(4)                 "__bibtex__",
(4)                 "__version__",
(4)                 "__version_info__",
(4)                 "set_loglevel",
(4)                 "ExecutableNotFoundError",
(4)                 "get_configdir",
(4)                 "get_cachedir",
(4)                 "get_data_path",
(4)                 "matplotlib_fname",
(4)                 "MatplotlibDeprecationWarning",
(4)                 "RcParams",
(4)                 "rc_params",
(4)                 "rc_params_from_file",
(4)                 "rcParamsDefault",
(4)                 "rcParams",
(4)                 "rcParamsOrig",
(4)                 "defaultParams",
(4)                 "rc",
(4)                 "rcdefaults",
(4)                 "rc_file_defaults",
(4)                 "rc_file",
(4)                 "rc_context",
(4)                 "use",
(4)                 "get_backend",
(4)                 "interactive",
(4)                 "is_interactive",
(4)                 "colormaps",
(4)                 "color_sequences",
(0)             ]
(0)             import atexit
(0)             from collections import namedtuple
(0)             from collections.abc import MutableMapping
(0)             import contextlib
(0)             import functools
(0)             import importlib
(0)             import inspect
(0)             from inspect import Parameter
(0)             import locale
(0)             import logging
(0)             import os
(0)             from pathlib import Path
(0)             import pprint
(0)             import re
(0)             import shutil
(0)             import subprocess
(0)             import sys
(0)             import tempfile
(0)             from packaging.version import parse as parse_version
(0)             from . import _api, _version, cbook, _docstring, rcsetup
(0)             from matplotlib.cbook import sanitize_sequence
(0)             from matplotlib._api import MatplotlibDeprecationWarning
(0)             from matplotlib.rcsetup import cycler  # noqa: F401
(0)             from matplotlib.rcsetup import validate_backend
(0)             _log = logging.getLogger(__name__)
(0)             __bibtex__ = r"""@Article{Hunter:2007,
(2)               Author    = {Hunter, J. D.},
(2)               Title     = {Matplotlib: A 2D graphics environment},
(2)               Journal   = {Computing in Science \& Engineering},
(2)               Volume    = {9},
(2)               Number    = {3},
(2)               Pages     = {90--95},
(2)               abstract  = {Matplotlib is a 2D graphics package used for Python
(2)               for application development, interactive scripting, and
```

```
(2)             publication-quality image generation across user
(2)             interfaces and operating systems.},
(2)             publisher = {IEEE COMPUTER SOC},
(2)             year      = 2007
(0)           }"""
(0)           _VersionInfo = namedtuple('_VersionInfo',
(26)                                     'major, minor, micro, releaselevel, serial')
(0)           def _parse_to_version_info(version_str):
(4)               """
(4)               Parse a version string to a namedtuple analogous to sys.version_info.
(4)               See:
(4)               https://packaging.pypa.io/en/latest/version.html#packaging.version.parse
(4)               https://docs.python.org/3/library/sys.html#sys.version_info
(4)               """
(4)               v = parse_version(version_str)
(4)               if v.pre is None and v.post is None and v.dev is None:
(8)                   return _VersionInfo(v.major, v.minor, v.micro, 'final', 0)
(4)               elif v.dev is not None:
(8)                   return _VersionInfo(v.major, v.minor, v.micro, 'alpha', v.dev)
(4)               elif v.pre is not None:
(8)                   releaselevel = {
(12)                      'a': 'alpha',
(12)                      'b': 'beta',
(12)                      'rc': 'candidate'}.get(v.pre[0], 'alpha')
(8)                   return _VersionInfo(v.major, v.minor, v.micro, releaselevel, v.pre[1])
(4)               else:
(8)                   return _VersionInfo(v.major, v.minor, v.micro + 1, 'alpha', v.post)
(0)           def _get_version():
(4)               """Return the version string used for __version__."""
(4)               root = Path(__file__).resolve().parents[2]
(4)               if ((root / ".matplotlib-repo").exists()
(12)                      and (root / ".git").exists()
(12)                      and not (root / ".git/shallow").exists()):
(8)                   try:
(12)                      import setuptools_scm
(8)                   except ImportError:
(12)                      pass
(8)                   else:
(12)                      return setuptools_scm.get_version(
(16)                          root=root,
(16)                          version_scheme="release-branch-semver",
(16)                          local_scheme="node-and-date",
(16)                          fallback_version=_version.version,
(12)                      )
(4)               return _version.version
(0)           @_api.caching_module_getattr
(0)           class __getattr__:
(4)               __version__ = property(lambda self: _get_version())
(4)               __version_info__ = property(
(8)                   lambda self: _parse_to_version_info(self.__version__))
(0)           def _check_versions():
(4)               from . import ft2font  # noqa: F401
(4)               for modname, minver in [
(12)                      ("cycler", "0.10"),
(12)                      ("dateutil", "2.7"),
(12)                      ("kiwisolver", "1.3.1"),
(12)                      ("numpy", "1.23"),
(12)                      ("pyparsing", "2.3.1"),
(4)               ]:
(8)                   module = importlib.import_module(modname)
(8)                   if parse_version(module.__version__) < parse_version(minver):
(12)                      raise ImportError(f"Matplotlib requires {modname}>={minver}; "
(30)                                        f"you have {module.__version__}")
(0)           _check_versions()
(0)           @functools.cache
(0)           def _ensure_handler():
(4)               """
(4)               The first time this function is called, attach a `StreamHandler` using the
(4)               same format as `logging.basicConfig` to the Matplotlib root logger.
```

```
206: (4)                Return this handler every time this function is called.
207: (4)                """
208: (4)                handler = logging.StreamHandler()
209: (4)                handler.setFormatter(logging.Formatter(logging.BASIC_FORMAT))
210: (4)                _log.addHandler(handler)
211: (4)                return handler
212: (0)            def set_loglevel(level):
213: (4)                """
214: (4)                Configure Matplotlib's logging levels.
215: (4)                Matplotlib uses the standard library `logging` framework under the root
216: (4)                logger 'matplotlib'.  This is a helper function to:
217: (4)                - set Matplotlib's root logger level
218: (4)                - set the root logger handler's level, creating the handler
219: (6)                  if it does not exist yet
220: (4)                Typically, one should call ``set_loglevel("info")`` or
221: (4)                ``set_loglevel("debug")`` to get additional debugging information.
222: (4)                Users or applications that are installing their own logging handlers
223: (4)                may want to directly manipulate ``logging.getLogger('matplotlib')`` rather
224: (4)                than use this function.
225: (4)                Parameters
226: (4)                ----------
227: (4)                level : {"notset", "debug", "info", "warning", "error", "critical"}
228: (8)                    The log level of the handler.
229: (4)                Notes
230: (4)                -----
231: (4)                The first time this function is called, an additional handler is attached
232: (4)                to Matplotlib's root handler; this handler is reused every time and this
233: (4)                function simply manipulates the logger and handler's level.
234: (4)                """
235: (4)                _log.setLevel(level.upper())
236: (4)                _ensure_handler().setLevel(level.upper())
237: (0)            def _logged_cached(fmt, func=None):
238: (4)                """
239: (4)                Decorator that logs a function's return value, and memoizes that value.
240: (4)                After ::
241: (8)                    @_logged_cached(fmt)
242: (8)                    def func(): ...
243: (4)                the first call to *func* will log its return value at the DEBUG level
using
244: (4)                %-format string *fmt*, and memoize it; later calls to *func* will directly
245: (4)                return that value.
246: (4)                """
247: (4)                if func is None:  # Return the actual decorator.
248: (8)                    return functools.partial(_logged_cached, fmt)
249: (4)                called = False
250: (4)                ret = None
251: (4)                @functools.wraps(func)
252: (4)                def wrapper(**kwargs):
253: (8)                    nonlocal called, ret
254: (8)                    if not called:
255: (12)                       ret = func(**kwargs)
256: (12)                       called = True
257: (12)                       _log.debug(fmt, ret)
258: (8)                    return ret
259: (4)                return wrapper
260: (0)            _ExecInfo = namedtuple("_ExecInfo", "executable raw_version version")
261: (0)            class ExecutableNotFoundError(FileNotFoundError):
262: (4)                """
263: (4)                Error raised when an executable that Matplotlib optionally
264: (4)                depends on can't be found.
265: (4)                """
266: (4)                pass
267: (0)            @functools.cache
268: (0)            def _get_executable_info(name):
269: (4)                """
270: (4)                Get the version of some executable that Matplotlib optionally depends on.
271: (4)                .. warning::
272: (7)                   The list of executables that this function supports is set according to
273: (7)                   Matplotlib's internal needs, and may change without notice.
```

```
    Parameters
    ----------
    name : str
        The executable to query.  The following values are currently supported:

        "dvipng", "gs", "inkscape", "magick", "pdftocairo", "pdftops".  This
        list is subject to change without notice.
    Returns
    -------
    tuple
        A namedtuple with fields ``executable`` (`str`) and ``version``
        (`packaging.Version`, or ``None`` if the version cannot be determined).
    Raises
    ------
    ExecutableNotFoundError
        If the executable is not found or older than the oldest version
        supported by Matplotlib.  For debugging purposes, it is also
        possible to "hide" an executable from Matplotlib by adding it to the
        :envvar:`_MPLHIDEEXECUTABLES` environment variable (a comma-separated
        list), which must be set prior to any calls to this function.
    ValueError
        If the executable is not one that we know how to query.
    """
    def impl(args, regex, min_ver=None, ignore_exit_code=False):
        try:
            output = subprocess.check_output(
                args, stderr=subprocess.STDOUT,
                text=True, errors="replace")
        except subprocess.CalledProcessError as _cpe:
            if ignore_exit_code:
                output = _cpe.output
            else:
                raise ExecutableNotFoundError(str(_cpe)) from _cpe
        except OSError as _ose:
            raise ExecutableNotFoundError(str(_ose)) from _ose
        match = re.search(regex, output)
        if match:
            raw_version = match.group(1)
            version = parse_version(raw_version)
            if min_ver is not None and version < parse_version(min_ver):
                raise ExecutableNotFoundError(
                    f"You have {args[0]} version {version} but the minimum "
                    f"version supported by Matplotlib is {min_ver}")
            return _ExecInfo(args[0], raw_version, version)
        else:
            raise ExecutableNotFoundError(
                f"Failed to determine the version of {args[0]} from "
                f"{' '.join(args)}, which output {output}")
    if name in os.environ.get("_MPLHIDEEXECUTABLES", "").split(","):
        raise ExecutableNotFoundError(f"{name} was hidden")
    if name == "dvipng":
        return impl(["dvipng", "-version"], "(?m)^dvipng(?: .*)? (.+)", "1.6")
    elif name == "gs":
        execs = (["gswin32c", "gswin64c", "mgs", "gs"]  # "mgs" for miktex.
                 if sys.platform == "win32" else
                 ["gs"])
        for e in execs:
            try:
                return impl([e, "--version"], "(.*)", "9")
            except ExecutableNotFoundError:
                pass
        message = "Failed to find a Ghostscript installation"
        raise ExecutableNotFoundError(message)
    elif name == "inkscape":
        try:
            return impl(["inkscape", "--without-gui", "-V"],
                        "Inkscape ([^ ]*)")
        except ExecutableNotFoundError:
```

```
(12)                        pass  # Suppress exception chaining.
(8)                     return impl(["inkscape", "-V"], "Inkscape ([^ ]*)")
(4)                 elif name == "magick":
(8)                     if sys.platform == "win32":
(12)                        import winreg
(12)                        binpath = ""
(12)                        for flag in [0, winreg.KEY_WOW64_32KEY, winreg.KEY_WOW64_64KEY]:
(16)                            try:
(20)                                with winreg.OpenKeyEx(
(28)                                        winreg.HKEY_LOCAL_MACHINE,
(28)                                        r"Software\Imagemagick\Current",
(28)                                        0, winreg.KEY_QUERY_VALUE | flag) as hkey:
(24)                                    binpath = winreg.QueryValueEx(hkey, "BinPath")[0]
(16)                            except OSError:
(20)                                pass
(12)                        path = None
(12)                        if binpath:
(16)                            for name in ["convert.exe", "magick.exe"]:
(20)                                candidate = Path(binpath, name)
(20)                                if candidate.exists():
(24)                                    path = str(candidate)
(24)                                    break
(12)                        if path is None:
(16)                            raise ExecutableNotFoundError(
(20)                                "Failed to find an ImageMagick installation")
(8)                     else:
(12)                        path = "convert"
(8)                     info = impl([path, "--version"], r"^Version: ImageMagick (\S*)")
(8)                     if info.raw_version == "7.0.10-34":
(12)                        raise ExecutableNotFoundError(
(16)                            f"You have ImageMagick {info.version}, which is unsupported")
(8)                     return info
(4)                 elif name == "pdftocairo":
(8)                     return impl(["pdftocairo", "-v"], "pdftocairo version (.*)")
(4)                 elif name == "pdftops":
(8)                     info = impl(["pdftops", "-v"], "^pdftops version (.*)",
(20)                                ignore_exit_code=True)
(8)                     if info and not (
(16)                            3 <= info.version.major or
(16)                            parse_version("0.9") <= info.version < parse_version("1.0")):
(12)                        raise ExecutableNotFoundError(
(16)                            f"You have pdftops version {info.version} but the minimum "
(16)                            f"version supported by Matplotlib is 3.0")
(8)                     return info
(4)                 else:
(8)                     raise ValueError(f"Unknown executable: {name!r}")
(0)             def _get_xdg_config_dir():
(4)                 """
(4)                 Return the XDG configuration directory, according to the XDG base
(4)                 directory spec:
(4)                 https://specifications.freedesktop.org/basedir-spec/basedir-spec-
latest.html
(4)                 """
(4)                 return os.environ.get('XDG_CONFIG_HOME') or str(Path.home() / ".config")
(0)             def _get_xdg_cache_dir():
(4)                 """
(4)                 Return the XDG cache directory, according to the XDG base directory spec:
(4)                 https://specifications.freedesktop.org/basedir-spec/basedir-spec-
latest.html
(4)                 """
(4)                 return os.environ.get('XDG_CACHE_HOME') or str(Path.home() / ".cache")
(0)             def _get_config_or_cache_dir(xdg_base_getter):
(4)                 configdir = os.environ.get('MPLCONFIGDIR')
(4)                 if configdir:
(8)                     configdir = Path(configdir).resolve()
(4)                 elif sys.platform.startswith(('linux', 'freebsd')):
(8)                     configdir = Path(xdg_base_getter(), "matplotlib")
(4)                 else:
(8)                     configdir = Path.home() / ".matplotlib"
```

```
408: (4)                try:
409: (8)                    configdir.mkdir(parents=True, exist_ok=True)
410: (4)                except OSError:
411: (8)                    pass
412: (4)                else:
413: (8)                    if os.access(str(configdir), os.W_OK) and configdir.is_dir():
414: (12)                       return str(configdir)
415: (4)                try:
416: (8)                    tmpdir = tempfile.mkdtemp(prefix="matplotlib-")
417: (4)                except OSError as exc:
418: (8)                    raise OSError(
419: (12)                       f"Matplotlib requires access to a writable cache directory, but
the "
420: (12)                       f"default path ({configdir}) is not a writable directory, and a
temporary "
421: (12)                       f"directory could not be created; set the MPLCONFIGDIR environment
"
422: (12)                       f"variable to a writable directory") from exc
423: (4)                os.environ["MPLCONFIGDIR"] = tmpdir
424: (4)                atexit.register(shutil.rmtree, tmpdir)
425: (4)                _log.warning(
426: (8)                    "Matplotlib created a temporary cache directory at %s because the
default path "
427: (8)                    "(%s) is not a writable directory; it is highly recommended to set the
"
428: (8)                    "MPLCONFIGDIR environment variable to a writable directory, in
particular to "
429: (8)                    "speed up the import of Matplotlib and to better support
multiprocessing.",
430: (8)                    tmpdir, configdir)
431: (4)                return tmpdir
432: (0)            @_logged_cached('CONFIGDIR=%s')
433: (0)            def get_configdir():
434: (4)                """
435: (4)                Return the string path of the configuration directory.
436: (4)                The directory is chosen as follows:
437: (4)                1. If the MPLCONFIGDIR environment variable is supplied, choose that.
438: (4)                2. On Linux, follow the XDG specification and look first in
439: (7)                   ``$XDG_CONFIG_HOME``, if defined, or ``$HOME/.config``.  On other
440: (7)                   platforms, choose ``$HOME/.matplotlib``.
441: (4)                3. If the chosen directory exists and is writable, use that as the
442: (7)                   configuration directory.
443: (4)                4. Else, create a temporary directory, and use it as the configuration
444: (7)                   directory.
445: (4)                """
446: (4)                return _get_config_or_cache_dir(_get_xdg_config_dir)
447: (0)            @_logged_cached('CACHEDIR=%s')
448: (0)            def get_cachedir():
449: (4)                """
450: (4)                Return the string path of the cache directory.
451: (4)                The procedure used to find the directory is the same as for
452: (4)                `get_configdir`, except using ``$XDG_CACHE_HOME``/``$HOME/.cache``
instead.
453: (4)                """
454: (4)                return _get_config_or_cache_dir(_get_xdg_cache_dir)
455: (0)            @_logged_cached('matplotlib data path: %s')
456: (0)            def get_data_path():
457: (4)                """Return the path to Matplotlib data."""
458: (4)                return str(Path(__file__).with_name("mpl-data"))
459: (0)            def matplotlib_fname():
460: (4)                """
461: (4)                Get the location of the config file.
462: (4)                The file location is determined in the following order
463: (4)                - ``$PWD/matplotlibrc``
464: (4)                - ``$MATPLOTLIBRC`` if it is not a directory
465: (4)                - ``$MATPLOTLIBRC/matplotlibrc``
466: (4)                - ``$MPLCONFIGDIR/matplotlibrc``
467: (4)                - On Linux,
468: (8)                    - ``$XDG_CONFIG_HOME/matplotlib/matplotlibrc`` (if
```

```
``$XDG_CONFIG_HOME``
469: (10)                     is defined)
470: (8)                    - or ``$HOME/.config/matplotlib/matplotlibrc`` (if
``$XDG_CONFIG_HOME``
471: (10)                     is not defined)
472: (4)                - On other platforms,
473: (6)                  - ``$HOME/.matplotlib/matplotlibrc`` if ``$HOME`` is defined
474: (4)                - Lastly, it looks in ``$MATPLOTLIBDATA/matplotlibrc``, which should
always
475: (6)                  exist.
476: (4)                """
477: (4)                def gen_candidates():
478: (8)                    yield 'matplotlibrc'
479: (8)                    try:
480: (12)                       matplotlibrc = os.environ['MATPLOTLIBRC']
481: (8)                    except KeyError:
482: (12)                       pass
483: (8)                    else:
484: (12)                       yield matplotlibrc
485: (12)                       yield os.path.join(matplotlibrc, 'matplotlibrc')
486: (8)                    yield os.path.join(get_configdir(), 'matplotlibrc')
487: (8)                    yield os.path.join(get_data_path(), 'matplotlibrc')
488: (4)                for fname in gen_candidates():
489: (8)                    if os.path.exists(fname) and not os.path.isdir(fname):
490: (12)                       return fname
491: (4)                raise RuntimeError("Could not find matplotlibrc file; your Matplotlib "
492: (23)                                  "install is broken")
493: (0)            _deprecated_map = {}
494: (0)            _deprecated_ignore_map = {}
495: (0)            _deprecated_remain_as_none = {}
496: (0)            @_docstring.Substitution(
497: (4)                "\n".join(map("- {}".format, sorted(rcsetup._validators, key=str.lower)))
498: (0)            )
499: (0)            class RcParams(MutableMapping, dict):
500: (4)                """
501: (4)                A dict-like key-value store for config parameters, including validation.
502: (4)                Validating functions are defined and associated with rc parameters in
503: (4)                :mod:`matplotlib.rcsetup`.
504: (4)                The list of rcParams is:
505: (4)                %s
506: (4)                See Also
507: (4)                --------
508: (4)                :ref:`customizing-with-matplotlibrc-files`
509: (4)                """
510: (4)                validate = rcsetup._validators
511: (4)                def __init__(self, *args, **kwargs):
512: (8)                    self.update(*args, **kwargs)
513: (4)                def _set(self, key, val):
514: (8)                    """
515: (8)                    Directly write data bypassing deprecation and validation logic.
516: (8)                    Notes
517: (8)                    -----
518: (8)                    As end user or downstream library you almost always should use
519: (8)                    ``rcParams[key] = val`` and not ``_set()``.
520: (8)                    There are only very few special cases that need direct data access.
521: (8)                    These cases previously used ``dict.__setitem__(rcParams, key, val)``,
522: (8)                    which is now deprecated and replaced by ``rcParams._set(key, val)``.
523: (8)                    Even though private, we guarantee API stability for ``rcParams._set``,
524: (8)                    i.e. it is subject to Matplotlib's API and deprecation policy.
525: (8)                    :meta public:
526: (8)                    """
527: (8)                    dict.__setitem__(self, key, val)
528: (4)                def _get(self, key):
529: (8)                    """
530: (8)                    Directly read data bypassing deprecation, backend and validation
531: (8)                    logic.
532: (8)                    Notes
533: (8)                    -----
534: (8)                    As end user or downstream library you almost always should use
```

```
535: (8)                    ``val = rcParams[key]`` and not ``_get()``.
536: (8)                    There are only very few special cases that need direct data access.
537: (8)                    These cases previously used ``dict.__getitem__(rcParams, key, val)``,
538: (8)                    which is now deprecated and replaced by ``rcParams._get(key)``.
539: (8)                    Even though private, we guarantee API stability for ``rcParams._get``,
540: (8)                    i.e. it is subject to Matplotlib's API and deprecation policy.
541: (8)                    :meta public:
542: (8)                    """
543: (8)                    return dict.__getitem__(self, key)
544: (4)                def __setitem__(self, key, val):
545: (8)                    try:
546: (12)                       if key in _deprecated_map:
547: (16)                           version, alt_key, alt_val, inverse_alt = _deprecated_map[key]
548: (16)                           _api.warn_deprecated(
549: (20)                               version, name=key, obj_type="rcparam",
alternative=alt_key)
550: (16)                           key = alt_key
551: (16)                           val = alt_val(val)
552: (12)                       elif key in _deprecated_remain_as_none and val is not None:
553: (16)                           version, = _deprecated_remain_as_none[key]
554: (16)                           _api.warn_deprecated(version, name=key, obj_type="rcparam")
555: (12)                       elif key in _deprecated_ignore_map:
556: (16)                           version, alt_key = _deprecated_ignore_map[key]
557: (16)                           _api.warn_deprecated(
558: (20)                               version, name=key, obj_type="rcparam",
alternative=alt_key)
559: (16)                           return
560: (12)                       elif key == 'backend':
561: (16)                           if val is rcsetup._auto_backend_sentinel:
562: (20)                               if 'backend' in self:
563: (24)                                   return
564: (12)                       try:
565: (16)                           cval = self.validate[key](val)
566: (12)                       except ValueError as ve:
567: (16)                           raise ValueError(f"Key {key}: {ve}") from None
568: (12)                       self._set(key, cval)
569: (8)                    except KeyError as err:
570: (12)                       raise KeyError(
571: (16)                           f"{key} is not a valid rc parameter (see rcParams.keys() for "
572: (16)                           f"a list of valid parameters)") from err
573: (4)                def __getitem__(self, key):
574: (8)                    if key in _deprecated_map:
575: (12)                       version, alt_key, alt_val, inverse_alt = _deprecated_map[key]
576: (12)                       _api.warn_deprecated(
577: (16)                           version, name=key, obj_type="rcparam", alternative=alt_key)
578: (12)                       return inverse_alt(self._get(alt_key))
579: (8)                    elif key in _deprecated_ignore_map:
580: (12)                       version, alt_key = _deprecated_ignore_map[key]
581: (12)                       _api.warn_deprecated(
582: (16)                           version, name=key, obj_type="rcparam", alternative=alt_key)
583: (12)                       return self._get(alt_key) if alt_key else None
584: (8)                    elif key == "backend" and self is globals().get("rcParams"):
585: (12)                       val = self._get(key)
586: (12)                       if val is rcsetup._auto_backend_sentinel:
587: (16)                           from matplotlib import pyplot as plt
588: (16)                           plt.switch_backend(rcsetup._auto_backend_sentinel)
589: (8)                    return self._get(key)
590: (4)                def _get_backend_or_none(self):
591: (8)                    """Get the requested backend, if any, without triggering
resolution."""
592: (8)                    backend = self._get("backend")
593: (8)                    return None if backend is rcsetup._auto_backend_sentinel else backend
594: (4)                def __repr__(self):
595: (8)                    class_name = self.__class__.__name__
596: (8)                    indent = len(class_name) + 1
597: (8)                    with _api.suppress_matplotlib_deprecation_warning():
598: (12)                       repr_split = pprint.pformat(dict(self), indent=1,
599: (40)                                                   width=80 - indent).split('\n')
600: (8)                    repr_indented = ('\n' + ' ' * indent).join(repr_split)
```

```
(8)                 return f'{class_name}({repr_indented})'
(4)             def __str__(self):
(8)                 return '\n'.join(map('{0[0]}: {0[1]}'.format, sorted(self.items())))
(4)             def __iter__(self):
(8)                 """Yield sorted list of keys."""
(8)                 with _api.suppress_matplotlib_deprecation_warning():
(12)                    yield from sorted(dict.__iter__(self))
(4)             def __len__(self):
(8)                 return dict.__len__(self)
(4)             def find_all(self, pattern):
(8)                 """
(8)                 Return the subset of this RcParams dictionary whose keys match,
(8)                 using :func:`re.search`, the given ``pattern``.
(8)                 .. note::
(12)                    Changes to the returned dictionary are *not* propagated to
(12)                    the parent RcParams dictionary.
(8)                 """
(8)                 pattern_re = re.compile(pattern)
(8)                 return RcParams((key, value)
(24)                                for key, value in self.items()
(24)                                if pattern_re.search(key))
(4)             def copy(self):
(8)                 """Copy this RcParams instance."""
(8)                 rccopy = RcParams()
(8)                 for k in self:  # Skip deprecations and revalidation.
(12)                    rccopy._set(k, self._get(k))
(8)                 return rccopy
(0)         def rc_params(fail_on_error=False):
(4)             """Construct a `RcParams` instance from the default Matplotlib rc file."""
(4)             return rc_params_from_file(matplotlib_fname(), fail_on_error)
(0)         @functools.cache
(0)         def _get_ssl_context():
(4)             try:
(8)                 import certifi
(4)             except ImportError:
(8)                 _log.debug("Could not import certifi.")
(8)                 return None
(4)             import ssl
(4)             return ssl.create_default_context(cafile=certifi.where())
(0)         @contextlib.contextmanager
(0)         def _open_file_or_url(fname):
(4)             if (isinstance(fname, str)
(12)                    and fname.startswith(('http://', 'https://', 'ftp://', 'file:'))):
(8)                 import urllib.request
(8)                 ssl_ctx = _get_ssl_context()
(8)                 if ssl_ctx is None:
(12)                    _log.debug(
(16)                        "Could not get certifi ssl context, https may not work."
(12)                    )
(8)                 with urllib.request.urlopen(fname, context=ssl_ctx) as f:
(12)                    yield (line.decode('utf-8') for line in f)
(4)             else:
(8)                 fname = os.path.expanduser(fname)
(8)                 with open(fname, encoding='utf-8') as f:
(12)                    yield f
(0)         def _rc_params_in_file(fname, transform=lambda x: x, fail_on_error=False):
(4)             """
(4)             Construct a `RcParams` instance from file *fname*.
(4)             Unlike `rc_params_from_file`, the configuration class only contains the
(4)             parameters specified in the file (i.e. default values are not filled in).
(4)             Parameters
(4)             ----------
(4)             fname : path-like
(8)                 The loaded file.
(4)             transform : callable, default: the identity function
(8)                 A function called on each individual line of the file to transform it,
(8)                 before further parsing.
(4)             fail_on_error : bool, default: False
(8)                 Whether invalid entries should result in an exception or a warning.
```

```
670: (4)                """
671: (4)                import matplotlib as mpl
672: (4)                rc_temp = {}
673: (4)                with _open_file_or_url(fname) as fd:
674: (8)                    try:
675: (12)                       for line_no, line in enumerate(fd, 1):
676: (16)                           line = transform(line)
677: (16)                           strippedline = cbook._strip_comment(line)
678: (16)                           if not strippedline:
679: (20)                               continue
680: (16)                           tup = strippedline.split(':', 1)
681: (16)                           if len(tup) != 2:
682: (20)                               _log.warning('Missing colon in file %r, line %d (%r)',
683: (33)                                            fname, line_no, line.rstrip('\n'))
684: (20)                               continue
685: (16)                           key, val = tup
686: (16)                           key = key.strip()
687: (16)                           val = val.strip()
688: (16)                           if val.startswith('"') and val.endswith('"'):
689: (20)                               val = val[1:-1]  # strip double quotes
690: (16)                           if key in rc_temp:
691: (20)                               _log.warning('Duplicate key in file %r, line %d (%r)',
692: (33)                                            fname, line_no, line.rstrip('\n'))
693: (16)                           rc_temp[key] = (val, line, line_no)
694: (8)                    except UnicodeDecodeError:
695: (12)                       _log.warning('Cannot decode configuration file %r as utf-8.',
696: (25)                                    fname)
697: (12)                       raise
698: (4)                config = RcParams()
699: (4)                for key, (val, line, line_no) in rc_temp.items():
700: (8)                    if key in rcsetup._validators:
701: (12)                       if fail_on_error:
702: (16)                           config[key] = val  # try to convert to proper type or raise
703: (12)                       else:
704: (16)                           try:
705: (20)                               config[key] = val  # try to convert to proper type or skip
706: (16)                           except Exception as msg:
707: (20)                               _log.warning('Bad value in file %r, line %d (%r): %s',
708: (33)                                            fname, line_no, line.rstrip('\n'), msg)
709: (8)                    elif key in _deprecated_ignore_map:
710: (12)                       version, alt_key = _deprecated_ignore_map[key]
711: (12)                       _api.warn_deprecated(
712: (16)                           version, name=key, alternative=alt_key, obj_type='rcparam',
713: (16)                           addendum="Please update your matplotlibrc.")
714: (8)                    else:
715: (12)                       version = ('main' if '.post' in mpl.__version__
716: (23)                                  else f'v{mpl.__version__}')
717: (12)                       _log.warning("""
718: (0)            Bad key %(key)s in file %(fname)s, line %(line_no)s (%(line)r)
719: (0)            You probably need to get an updated matplotlibrc file from
720: (0)            https://github.com/matplotlib/matplotlib/blob/%(version)s/lib/matplotlib/mpl-
data/matplotlibrc
721: (0)            or from the matplotlib source distribution""",
722: (25)                                  dict(key=key, fname=fname, line_no=line_no,
723: (30)                                       line=line.rstrip('\n'), version=version))
724: (4)                return config
725: (0)            def rc_params_from_file(fname, fail_on_error=False,
use_default_template=True):
726: (4)                """
727: (4)                Construct a `RcParams` from file *fname*.
728: (4)                Parameters
729: (4)                ----------
730: (4)                fname : str or path-like
731: (8)                    A file with Matplotlib rc settings.
732: (4)                fail_on_error : bool
733: (8)                    If True, raise an error when the parser fails to convert a parameter.
734: (4)                use_default_template : bool
735: (8)                    If True, initialize with default parameters before updating with those
736: (8)                    in the given file. If False, the configuration class only contains the
```

```
(8)                 parameters specified in the file. (Useful for updating dicts.)
(4)             """
(4)             config_from_file = _rc_params_in_file(fname, fail_on_error=fail_on_error)
(4)             if not use_default_template:
(8)                 return config_from_file
(4)             with _api.suppress_matplotlib_deprecation_warning():
(8)                 config = RcParams({**rcParamsDefault, **config_from_file})
(4)             if "".join(config['text.latex.preamble']):
(8)                 _log.info("""
(0)         ***************************************************************
(0)         You have the following UNSUPPORTED LaTeX preamble customizations:
(0)         %s
(0)         Please do not ask for support with these customizations active.
(0)         ***************************************************************
(0)         """, '\n'.join(config['text.latex.preamble']))
(4)             _log.debug('loaded rc file %s', fname)
(4)             return config
(0)         rcParamsDefault = _rc_params_in_file(
(4)             cbook._get_data_path("matplotlibrc"),
(4)             transform=lambda line: line[1:] if line.startswith("#") else line,
(4)             fail_on_error=True)
(0)         dict.update(rcParamsDefault, rcsetup._hardcoded_defaults)
(0)         dict.setdefault(rcParamsDefault, "backend", rcsetup._auto_backend_sentinel)
(0)         rcParams = RcParams()  # The global instance.
(0)         dict.update(rcParams, dict.items(rcParamsDefault))
(0)         dict.update(rcParams, _rc_params_in_file(matplotlib_fname()))
(0)         rcParamsOrig = rcParams.copy()
(0)         with _api.suppress_matplotlib_deprecation_warning():
(4)             defaultParams = rcsetup.defaultParams = {
(8)                 key: [(rcsetup._auto_backend_sentinel if key == "backend" else
(15)                       rcParamsDefault[key]),
(14)                      validator]
(8)                 for key, validator in rcsetup._validators.items()}
(0)         if rcParams['axes.formatter.use_locale']:
(4)             locale.setlocale(locale.LC_ALL, '')
(0)         def rc(group, **kwargs):
(4)             """
(4)             Set the current `.rcParams`.  *group* is the grouping for the rc, e.g.,
(4)             for ``lines.linewidth`` the group is ``lines``, for
(4)             ``axes.facecolor``, the group is ``axes``, and so on.  Group may
(4)             also be a list or tuple of group names, e.g., (*xtick*, *ytick*).
(4)             *kwargs* is a dictionary attribute name/value pairs, e.g.,::
(6)               rc('lines', linewidth=2, color='r')
(4)             sets the current `.rcParams` and is equivalent to::
(6)               rcParams['lines.linewidth'] = 2
(6)               rcParams['lines.color'] = 'r'
(4)             The following aliases are available to save typing for interactive users:
(4)             =====   =================
(4)             Alias   Property
(4)             =====   =================
(4)             'lw'    'linewidth'
(4)             'ls'    'linestyle'
(4)             'c'     'color'
(4)             'fc'    'facecolor'
(4)             'ec'    'edgecolor'
(4)             'mew'   'markeredgewidth'
(4)             'aa'    'antialiased'
(4)             =====   =================
(4)             Thus you could abbreviate the above call as::
(10)                  rc('lines', lw=2, c='r')
(4)             Note you can use python's kwargs dictionary facility to store
(4)             dictionaries of default parameters.  e.g., you can customize the
(4)             font rc as follows::
(6)               font = {'family' : 'monospace',
(14)                      'weight' : 'bold',
(14)                      'size'   : 'larger'}
(6)               rc('font', **font)  # pass in the font dict as kwargs
(4)             This enables you to easily switch between several configurations.  Use
(4)             ``matplotlib.style.use('default')`` or :func:`~matplotlib.rcdefaults` to
```

```
(4)                 restore the default `.rcParams` after changes.
(4)                 Notes
(4)                 -----
(4)                 Similar functionality is available by using the normal dict interface,
i.e.
(4)                 ``rcParams.update({"lines.linewidth": 2, ...})`` (but ``rcParams.update``
(4)                 does not support abbreviations or grouping).
(4)                 """
(4)                 aliases = {
(8)                     'lw':  'linewidth',
(8)                     'ls':  'linestyle',
(8)                     'c':   'color',
(8)                     'fc':  'facecolor',
(8)                     'ec':  'edgecolor',
(8)                     'mew': 'markeredgewidth',
(8)                     'aa':  'antialiased',
(8)                     }
(4)                 if isinstance(group, str):
(8)                     group = (group,)
(4)                 for g in group:
(8)                     for k, v in kwargs.items():
(12)                        name = aliases.get(k) or k
(12)                        key = f'{g}.{name}'
(12)                        try:
(16)                            rcParams[key] = v
(12)                        except KeyError as err:
(16)                            raise KeyError(('Unrecognized key "%s" for group "%s" and '
(32)                                            'name "%s"') % (key, g, name)) from err
(0)             def rcdefaults():
(4)                 """
(4)                 Restore the `.rcParams` from Matplotlib's internal default style.
(4)                 Style-blacklisted `.rcParams` (defined in
(4)                 ``matplotlib.style.core.STYLE_BLACKLIST``) are not updated.
(4)                 See Also
(4)                 --------
(4)                 matplotlib.rc_file_defaults
(8)                     Restore the `.rcParams` from the rc file originally loaded by
(8)                     Matplotlib.
(4)                 matplotlib.style.use
(8)                     Use a specific style file.  Call ``style.use('default')`` to restore
(8)                     the default style.
(4)                 """
(4)                 with _api.suppress_matplotlib_deprecation_warning():
(8)                     from .style.core import STYLE_BLACKLIST
(8)                     rcParams.clear()
(8)                     rcParams.update({k: v for k, v in rcParamsDefault.items()
(25)                                     if k not in STYLE_BLACKLIST})
(0)             def rc_file_defaults():
(4)                 """
(4)                 Restore the `.rcParams` from the original rc file loaded by Matplotlib.
(4)                 Style-blacklisted `.rcParams` (defined in
(4)                 ``matplotlib.style.core.STYLE_BLACKLIST``) are not updated.
(4)                 """
(4)                 with _api.suppress_matplotlib_deprecation_warning():
(8)                     from .style.core import STYLE_BLACKLIST
(8)                     rcParams.update({k: rcParamsOrig[k] for k in rcParamsOrig
(25)                                     if k not in STYLE_BLACKLIST})
(0)             def rc_file(fname, *, use_default_template=True):
(4)                 """
(4)                 Update `.rcParams` from file.
(4)                 Style-blacklisted `.rcParams` (defined in
(4)                 ``matplotlib.style.core.STYLE_BLACKLIST``) are not updated.
(4)                 Parameters
(4)                 ----------
(4)                 fname : str or path-like
(8)                     A file with Matplotlib rc settings.
(4)                 use_default_template : bool
(8)                     If True, initialize with default parameters before updating with those
(8)                     in the given file. If False, the current configuration persists
```

```
(8)                 and only the parameters specified in the file are updated.
(4)             """
(4)             with _api.suppress_matplotlib_deprecation_warning():
(8)                 from .style.core import STYLE_BLACKLIST
(8)                 rc_from_file = rc_params_from_file(
(12)                    fname, use_default_template=use_default_template)
(8)                 rcParams.update({k: rc_from_file[k] for k in rc_from_file
(25)                                 if k not in STYLE_BLACKLIST})
(0)         @contextlib.contextmanager
(0)         def rc_context(rc=None, fname=None):
(4)             """
(4)             Return a context manager for temporarily changing rcParams.
(4)             The :rc:`backend` will not be reset by the context manager.
(4)             rcParams changed both through the context manager invocation and
(4)             in the body of the context will be reset on context exit.
(4)             Parameters
(4)             ----------
(4)             rc : dict
(8)                 The rcParams to temporarily set.
(4)             fname : str or path-like
(8)                 A file with Matplotlib rc settings. If both *fname* and *rc* are given,
(8)                 settings from *rc* take precedence.
(4)             See Also
(4)             --------
(4)             :ref:`customizing-with-matplotlibrc-files`
(4)             Examples
(4)             --------
(4)             Passing explicit values via a dict::
(8)                 with mpl.rc_context({'interactive': False}):
(12)                    fig, ax = plt.subplots()
(12)                    ax.plot(range(3), range(3))
(12)                    fig.savefig('example.png')
(12)                    plt.close(fig)
(4)             Loading settings from a file::
(9)                  with mpl.rc_context(fname='print.rc'):
(13)                     plt.plot(x, y)  # uses 'print.rc'
(4)             Setting in the context body::
(8)                 with mpl.rc_context():
(12)                    mpl.rcParams['lines.linewidth'] = 5
(12)                    plt.plot(x, y)
(4)             """
(4)             orig = dict(rcParams.copy())
(4)             del orig['backend']
(4)             try:
(8)                 if fname:
(12)                    rc_file(fname)
(8)                 if rc:
(12)                    rcParams.update(rc)
(8)                 yield
(4)             finally:
(8)                 dict.update(rcParams, orig)  # Revert to the original rcs.
(0)         def use(backend, *, force=True):
(4)             """
(4)             Select the backend used for rendering and GUI integration.
(4)             If pyplot is already imported, `~matplotlib.pyplot.switch_backend` is used
(4)             and if the new backend is different than the current backend, all Figures
(4)             will be closed.
(4)             Parameters
(4)             ----------
(4)             backend : str
(8)                 The backend to switch to.  This can either be one of the standard
(8)                 backend names, which are case-insensitive:
(8)                 - interactive backends:
(10)                  GTK3Agg, GTK3Cairo, GTK4Agg, GTK4Cairo, MacOSX, nbAgg, notebook, QtAgg,
(10)                  QtCairo, TkAgg, TkCairo, WebAgg, WX, WXAgg, WXCairo, Qt5Agg, Qt5Cairo
(8)                 - non-interactive backends:
```

```
(10)                    agg, cairo, pdf, pgf, ps, svg, template
(8)                 or a string of the form: ``module://my.module.name``.
(8)                 notebook is a synonym for nbAgg.
(8)                 Switching to an interactive backend is not possible if an unrelated
(8)                 event loop has already been started (e.g., switching to GTK3Agg if a
(8)                 TkAgg window has already been opened).  Switching to a non-interactive
(8)                 backend is always possible.
(4)             force : bool, default: True
(8)                 If True (the default), raise an `ImportError` if the backend cannot be
(8)                 set up (either because it fails to import, or because an incompatible
(8)                 GUI interactive framework is already running); if False, silently
(8)                 ignore the failure.
(4)             See Also
(4)             --------
(4)             :ref:`backends`
(4)             matplotlib.get_backend
(4)             matplotlib.pyplot.switch_backend
(4)             """
(4)             name = validate_backend(backend)
(4)             if rcParams._get_backend_or_none() == name:
(8)                 pass
(4)             else:
(8)                 plt = sys.modules.get('matplotlib.pyplot')
(8)                 if plt is not None:
(12)                    try:
(16)                        plt.switch_backend(name)
(12)                    except ImportError:
(16)                        if force:
(20)                            raise
(8)                 else:
(12)                    rcParams['backend'] = backend
(4)             rcParams['backend_fallback'] = False
(0)         if os.environ.get('MPLBACKEND'):
(4)             rcParams['backend'] = os.environ.get('MPLBACKEND')
(0)         def get_backend():
(4)             """
(4)             Return the name of the current backend.
(4)             See Also
(4)             --------
(4)             matplotlib.use
(4)             """
(4)             return rcParams['backend']
(0)         def interactive(b):
(4)             """
(4)             Set whether to redraw after every plotting command (e.g.
`.pyplot.xlabel`).
(4)             """
(4)             rcParams['interactive'] = b
(0)         def is_interactive():
(4)             """
(4)             Return whether to redraw after every plotting command.
(4)             .. note::
(8)                 This function is only intended for use in backends. End users should
(8)                 use `.pyplot.isinteractive` instead.
(4)             """
(4)             return rcParams['interactive']
(0)         def _val_or_rc(val, rc_name):
(4)             """
(4)             If *val* is None, return ``mpl.rcParams[rc_name]``, otherwise return val.
(4)             """
(4)             return val if val is not None else rcParams[rc_name]
(0)         def _init_tests():
(4)             LOCAL_FREETYPE_VERSION = '2.6.1'
(4)             from matplotlib import ft2font
(4)             if (ft2font.__freetype_version__ != LOCAL_FREETYPE_VERSION or
(12)                    ft2font.__freetype_build_type__ != 'local'):
(8)                 _log.warning(
(12)                    "Matplotlib is not built with the correct FreeType version to run
tests.  "
```

```
1007: (12)                      "Rebuild without setting system-freetype=true in Meson setup
options.  "
1008: (12)                      "Expect many image comparison failures below.  "
1009: (12)                      "Expected freetype version %s.  "
1010: (12)                      "Found freetype version %s.  "
1011: (12)                      "Freetype build type is %slocal.",
1012: (12)                      LOCAL_FREETYPE_VERSION,
1013: (12)                      ft2font.__freetype_version__,
1014: (12)                      "" if ft2font.__freetype_build_type__ == 'local' else "not ")
1015: (0)           def _replacer(data, value):
1016: (4)               """
1017: (4)               Either returns ``data[value]`` or passes ``data`` back, converts either to
1018: (4)               a sequence.
1019: (4)               """
1020: (4)               try:
1021: (8)                   if isinstance(value, str):
1022: (12)                      value = data[value]
1023: (4)               except Exception:
1024: (8)                   pass
1025: (4)               return sanitize_sequence(value)
1026: (0)           def _label_from_arg(y, default_name):
1027: (4)               try:
1028: (8)                   return y.name
1029: (4)               except AttributeError:
1030: (8)                   if isinstance(default_name, str):
1031: (12)                      return default_name
1032: (4)               return None
1033: (0)           def _add_data_doc(docstring, replace_names):
1034: (4)               """
1035: (4)               Add documentation for a *data* field to the given docstring.
1036: (4)               Parameters
1037: (4)               ----------
1038: (4)               docstring : str
1039: (8)                   The input docstring.
1040: (4)               replace_names : list of str or None
1041: (8)                   The list of parameter names which arguments should be replaced by
1042: (8)                   ``data[name]`` (if ``data[name]`` does not throw an exception).  If
1043: (8)                   None, replacement is attempted for all arguments.
1044: (4)               Returns
1045: (4)               -------
1046: (4)               str
1047: (8)                   The augmented docstring.
1048: (4)               """
1049: (4)               if (docstring is None
1050: (12)                      or replace_names is not None and len(replace_names) == 0):
1051: (8)                   return docstring
1052: (4)               docstring = inspect.cleandoc(docstring)
1053: (4)               data_doc = ("""\
1054: (4)               If given, all parameters also accept a string ``s``, which is
1055: (4)               interpreted as ``data[s]`` (unless this raises an exception)."""
1056: (16)                          if replace_names is None else f"""\
1057: (4)               If given, the following parameters also accept a string ``s``, which is
1058: (4)               interpreted as ``data[s]`` (unless this raises an exception):
1059: (4)               {', '.join(map('*{}*'.format, replace_names))}""")
1060: (4)               if _log.level <= logging.DEBUG:
1061: (8)                   if "data : indexable object, optional" not in docstring:
1062: (12)                      _log.debug("data parameter docstring error: no data parameter")
1063: (8)                   if 'DATA_PARAMETER_PLACEHOLDER' not in docstring:
1064: (12)                      _log.debug("data parameter docstring error: missing placeholder")
1065: (4)               return docstring.replace('    DATA_PARAMETER_PLACEHOLDER', data_doc)
1066: (0)           def _preprocess_data(func=None, *, replace_names=None, label_namer=None):
1067: (4)               """
1068: (4)               A decorator to add a 'data' kwarg to a function.
1069: (4)               When applied::
1070: (8)                   @_preprocess_data()
1071: (8)                   def func(ax, *args, **kwargs): ...
1072: (4)               the signature is modified to ``decorated(ax, *args, data=None, **kwargs)``
1073: (4)               with the following behavior:
1074: (4)               - if called with ``data=None``, forward the other arguments to ``func``;
```

```
(4)                - otherwise, *data* must be a mapping; for any argument passed in as a
(6)                  string ``name``, replace the argument by ``data[name]`` (if this does
not
(6)                  throw an exception), then forward the arguments to ``func``.
(4)                In either case, any argument that is a `MappingView` is also converted to
a
(4)                list.
(4)                Parameters
(4)                ----------
(4)                replace_names : list of str or None, default: None
(8)                    The list of parameter names for which lookup into *data* should be
(8)                    attempted. If None, replacement is attempted for all arguments.
(4)                label_namer : str, default: None
(8)                    If set e.g. to "namer" (which must be a kwarg in the function's
(8)                    signature -- not as ``**kwargs``), if the *namer* argument passed in
is
(8)                    a (string) key of *data* and no *label* kwarg is passed, then use the
(8)                    (string) value of the *namer* as *label*. ::
(12)                       @_preprocess_data(label_namer="foo")
(12)                       def func(foo, label=None): ...
(12)                       func("key", data={"key": value})
(12)                       func.__wrapped__(value, label="key")
(4)                """
(4)                if func is None:  # Return the actual decorator.
(8)                    return functools.partial(
(12)                       _preprocess_data,
(12)                       replace_names=replace_names, label_namer=label_namer)
(4)                sig = inspect.signature(func)
(4)                varargs_name = None
(4)                varkwargs_name = None
(4)                arg_names = []
(4)                params = list(sig.parameters.values())
(4)                for p in params:
(8)                    if p.kind is Parameter.VAR_POSITIONAL:
(12)                       varargs_name = p.name
(8)                    elif p.kind is Parameter.VAR_KEYWORD:
(12)                       varkwargs_name = p.name
(8)                    else:
(12)                       arg_names.append(p.name)
(4)                data_param = Parameter("data", Parameter.KEYWORD_ONLY, default=None)
(4)                if varkwargs_name:
(8)                    params.insert(-1, data_param)
(4)                else:
(8)                    params.append(data_param)
(4)                new_sig = sig.replace(parameters=params)
(4)                arg_names = arg_names[1:]  # remove the first "ax" / self arg
(4)                assert {*arg_names}.issuperset(replace_names or []) or varkwargs_name, (
(8)                    "Matplotlib internal error: invalid replace_names "
(8)                    f"({replace_names!r}) for {func.__name__!r}")
(4)                assert label_namer is None or label_namer in arg_names, (
(8)                    "Matplotlib internal error: invalid label_namer "
(8)                    f"({label_namer!r}) for {func.__name__!r}")
(4)                @functools.wraps(func)
(4)                def inner(ax, *args, data=None, **kwargs):
(8)                    if data is None:
(12)                       return func(
(16)                           ax,
(16)                           *map(sanitize_sequence, args),
(16)                           **{k: sanitize_sequence(v) for k, v in kwargs.items()})
(8)                    bound = new_sig.bind(ax, *args, **kwargs)
(8)                    auto_label = (bound.arguments.get(label_namer)
(22)                                 or bound.kwargs.get(label_namer))
(8)                    for k, v in bound.arguments.items():
(12)                       if k == varkwargs_name:
(16)                           for k1, v1 in v.items():
(20)                               if replace_names is None or k1 in replace_names:
(24)                                   v[k1] = _replacer(data, v1)
(12)                       elif k == varargs_name:
(16)                           if replace_names is None:
```

```
1141: (20)                                bound.arguments[k] = tuple(_replacer(data, v1) for v1 in
v)
1142: (12)                        else:
1143: (16)                            if replace_names is None or k in replace_names:
1144: (20)                                bound.arguments[k] = _replacer(data, v)
1145: (8)                    new_args = bound.args
1146: (8)                    new_kwargs = bound.kwargs
1147: (8)                    args_and_kwargs = {**bound.arguments, **bound.kwargs}
1148: (8)                    if label_namer and "label" not in args_and_kwargs:
1149: (12)                        new_kwargs["label"] = _label_from_arg(
1150: (16)                            args_and_kwargs.get(label_namer), auto_label)
1151: (8)                    return func(*new_args, **new_kwargs)
1152: (4)                inner.__doc__ = _add_data_doc(inner.__doc__, replace_names)
1153: (4)                inner.__signature__ = new_sig
1154: (4)                return inner
1155: (0)            _log.debug('interactive is %s', is_interactive())
1156: (0)            _log.debug('platform is %s', sys.platform)
1157: (0)            from matplotlib.cm import _colormaps as colormaps  # noqa: E402
1158: (0)            from matplotlib.colors import _color_sequences as color_sequences  # noqa:
E402
```

File 2 - _afm.py:

```
1: (0)              """
2: (0)              A python interface to Adobe Font Metrics Files.
3: (0)              Although a number of other Python implementations exist, and may be more
4: (0)              complete than this, it was decided not to go with them because they were
5: (0)              either:
6: (0)              1) copyrighted or used a non-BSD compatible license
7: (0)              2) had too many dependencies and a free standing lib was needed
8: (0)              3) did more than needed and it was easier to write afresh rather than
9: (3)                 figure out how to get just what was needed.
10: (0)             It is pretty easy to use, and has no external dependencies:
11: (0)             >>> import matplotlib as mpl
12: (0)             >>> from pathlib import Path
13: (0)             >>> afm_path = Path(mpl.get_data_path(), 'fonts', 'afm', 'ptmr8a.afm')
14: (0)             >>>
15: (0)             >>> from matplotlib.afm import AFM
16: (0)             >>> with afm_path.open('rb') as fh:
17: (0)             ...     afm = AFM(fh)
18: (0)             >>> afm.string_width_height('What the heck?')
19: (0)             (6220.0, 694)
20: (0)             >>> afm.get_fontname()
21: (0)             'Times-Roman'
22: (0)             >>> afm.get_kern_dist('A', 'f')
23: (0)             0
24: (0)             >>> afm.get_kern_dist('A', 'y')
25: (0)             -92.0
26: (0)             >>> afm.get_bbox_char('!')
27: (0)             [130, -9, 238, 676]
28: (0)             As in the Adobe Font Metrics File Format Specification, all dimensions
29: (0)             are given in units of 1/1000 of the scale factor (point size) of the font
30: (0)             being used.
31: (0)             """
32: (0)             from collections import namedtuple
33: (0)             import logging
34: (0)             import re
35: (0)             from ._mathtext_data import uni2type1
36: (0)             _log = logging.getLogger(__name__)
37: (0)             def _to_int(x):
38: (4)                 return int(float(x))
39: (0)             def _to_float(x):
40: (4)                 if isinstance(x, bytes):
41: (8)                     x = x.decode('latin-1')
42: (4)                 return float(x.replace(',', '.'))
43: (0)             def _to_str(x):
44: (4)                 return x.decode('utf8')
```

```
45: (0)             def _to_list_of_ints(s):
46: (4)                 s = s.replace(b',', b' ')
47: (4)                 return [_to_int(val) for val in s.split()]
48: (0)             def _to_list_of_floats(s):
49: (4)                 return [_to_float(val) for val in s.split()]
50: (0)             def _to_bool(s):
51: (4)                 if s.lower().strip() in (b'false', b'0', b'no'):
52: (8)                     return False
53: (4)                 else:
54: (8)                     return True
55: (0)             def _parse_header(fh):
56: (4)                 """
57: (4)                 Read the font metrics header (up to the char metrics) and returns
58: (4)                 a dictionary mapping *key* to *val*.  *val* will be converted to the
59: (4)                 appropriate python type as necessary; e.g.:
60: (8)                     * 'False'->False
61: (8)                     * '0'->0
62: (8)                     * '-168 -218 1000 898'-> [-168, -218, 1000, 898]
63: (4)                 Dictionary keys are
64: (6)                   StartFontMetrics, FontName, FullName, FamilyName, Weight,
65: (6)                   ItalicAngle, IsFixedPitch, FontBBox, UnderlinePosition,
66: (6)                   UnderlineThickness, Version, Notice, EncodingScheme, CapHeight,
67: (6)                   XHeight, Ascender, Descender, StartCharMetrics
68: (4)                 """
69: (4)                 header_converters = {
70: (8)                     b'StartFontMetrics': _to_float,
71: (8)                     b'FontName': _to_str,
72: (8)                     b'FullName': _to_str,
73: (8)                     b'FamilyName': _to_str,
74: (8)                     b'Weight': _to_str,
75: (8)                     b'ItalicAngle': _to_float,
76: (8)                     b'IsFixedPitch': _to_bool,
77: (8)                     b'FontBBox': _to_list_of_ints,
78: (8)                     b'UnderlinePosition': _to_float,
79: (8)                     b'UnderlineThickness': _to_float,
80: (8)                     b'Version': _to_str,
81: (8)                     b'Notice': lambda x: x,
82: (8)                     b'EncodingScheme': _to_str,
83: (8)                     b'CapHeight': _to_float,  # Is the second version a mistake, or
84: (8)                     b'Capheight': _to_float,  # do some AFM files contain 'Capheight'? -
JKS
85: (8)                     b'XHeight': _to_float,
86: (8)                     b'Ascender': _to_float,
87: (8)                     b'Descender': _to_float,
88: (8)                     b'StdHW': _to_float,
89: (8)                     b'StdVW': _to_float,
90: (8)                     b'StartCharMetrics': _to_int,
91: (8)                     b'CharacterSet': _to_str,
92: (8)                     b'Characters': _to_int,
93: (4)                 }
94: (4)                 d = {}
95: (4)                 first_line = True
96: (4)                 for line in fh:
97: (8)                     line = line.rstrip()
98: (8)                     if line.startswith(b'Comment'):
99: (12)                        continue
100: (8)                    lst = line.split(b' ', 1)
101: (8)                    key = lst[0]
102: (8)                    if first_line:
103: (12)                       if key != b'StartFontMetrics':
104: (16)                           raise RuntimeError('Not an AFM file')
105: (12)                       first_line = False
106: (8)                    if len(lst) == 2:
107: (12)                       val = lst[1]
108: (8)                    else:
109: (12)                       val = b''
110: (8)                    try:
111: (12)                       converter = header_converters[key]
112: (8)                    except KeyError:
```

```
113: (12)                        _log.error("Found an unknown keyword in AFM header (was %r)", key)
114: (12)                        continue
115: (8)                    try:
116: (12)                        d[key] = converter(val)
117: (8)                    except ValueError:
118: (12)                        _log.error('Value error parsing header in AFM: %s, %s', key, val)
119: (12)                        continue
120: (8)                    if key == b'StartCharMetrics':
121: (12)                        break
122: (4)                else:
123: (8)                    raise RuntimeError('Bad parse')
124: (4)                return d
125: (0)            CharMetrics = namedtuple('CharMetrics', 'width, name, bbox')
126: (0)            CharMetrics.__doc__ = """
127: (4)                Represents the character metrics of a single character.
128: (4)                Notes
129: (4)                -----
130: (4)                The fields do currently only describe a subset of character metrics
131: (4)                information defined in the AFM standard.
132: (4)                """
133: (0)            CharMetrics.width.__doc__ = """The character width (WX)."""
134: (0)            CharMetrics.name.__doc__ = """The character name (N)."""
135: (0)            CharMetrics.bbox.__doc__ = """
136: (4)                The bbox of the character (B) as a tuple (*llx*, *lly*, *urx*, *ury*)."""
137: (0)            def _parse_char_metrics(fh):
138: (4)                """
139: (4)                Parse the given filehandle for character metrics information and return
140: (4)                the information as dicts.
141: (4)                It is assumed that the file cursor is on the line behind
142: (4)                'StartCharMetrics'.
143: (4)                Returns
144: (4)                -------
145: (4)                ascii_d : dict
146: (9)                     A mapping "ASCII num of the character" to `.CharMetrics`.
147: (4)                name_d : dict
148: (9)                     A mapping "character name" to `.CharMetrics`.
149: (4)                Notes
150: (4)                -----
151: (4)                This function is incomplete per the standard, but thus far parses
152: (4)                all the sample afm files tried.
153: (4)                """
154: (4)                required_keys = {'C', 'WX', 'N', 'B'}
155: (4)                ascii_d = {}
156: (4)                name_d = {}
157: (4)                for line in fh:
158: (8)                    line = _to_str(line.rstrip())  # Convert from byte-literal
159: (8)                    if line.startswith('EndCharMetrics'):
160: (12)                       return ascii_d, name_d
161: (8)                    vals = dict(s.strip().split(' ', 1) for s in line.split(';') if s)
162: (8)                    if not required_keys.issubset(vals):
163: (12)                       raise RuntimeError('Bad char metrics line: %s' % line)
164: (8)                    num = _to_int(vals['C'])
165: (8)                    wx = _to_float(vals['WX'])
166: (8)                    name = vals['N']
167: (8)                    bbox = _to_list_of_floats(vals['B'])
168: (8)                    bbox = list(map(int, bbox))
169: (8)                    metrics = CharMetrics(wx, name, bbox)
170: (8)                    if name == 'Euro':
171: (12)                       num = 128
172: (8)                    elif name == 'minus':
173: (12)                       num = ord("\N{MINUS SIGN}")  # 0x2212
174: (8)                    if num != -1:
175: (12)                       ascii_d[num] = metrics
176: (8)                    name_d[name] = metrics
177: (4)                raise RuntimeError('Bad parse')
178: (0)            def _parse_kern_pairs(fh):
179: (4)                """
180: (4)                Return a kern pairs dictionary; keys are (*char1*, *char2*) tuples and
181: (4)                values are the kern pair value.  For example, a kern pairs line like
```

```
182: (4)                ``KPX A y -50``
183: (4)                will be represented as::
184: (6)                  d[ ('A', 'y') ] = -50
185: (4)                """
186: (4)                line = next(fh)
187: (4)                if not line.startswith(b'StartKernPairs'):
188: (8)                    raise RuntimeError('Bad start of kern pairs data: %s' % line)
189: (4)                d = {}
190: (4)                for line in fh:
191: (8)                    line = line.rstrip()
192: (8)                    if not line:
193: (12)                       continue
194: (8)                    if line.startswith(b'EndKernPairs'):
195: (12)                       next(fh)  # EndKernData
196: (12)                       return d
197: (8)                    vals = line.split()
198: (8)                    if len(vals) != 4 or vals[0] != b'KPX':
199: (12)                       raise RuntimeError('Bad kern pairs line: %s' % line)
200: (8)                    c1, c2, val = _to_str(vals[1]), _to_str(vals[2]), _to_float(vals[3])
201: (8)                    d[(c1, c2)] = val
202: (4)                raise RuntimeError('Bad kern pairs parse')
203: (0)            CompositePart = namedtuple('CompositePart', 'name, dx, dy')
204: (0)            CompositePart.__doc__ = """
205: (4)                Represents the information on a composite element of a composite char."""
206: (0)            CompositePart.name.__doc__ = """Name of the part, e.g. 'acute'."""
207: (0)            CompositePart.dx.__doc__ = """x-displacement of the part from the origin."""
208: (0)            CompositePart.dy.__doc__ = """y-displacement of the part from the origin."""
209: (0)            def _parse_composites(fh):
210: (4)                """
211: (4)                Parse the given filehandle for composites information return them as a
212: (4)                dict.
213: (4)                It is assumed that the file cursor is on the line behind
'StartComposites'.
214: (4)                Returns
215: (4)                -------
216: (4)                dict
217: (8)                    A dict mapping composite character names to a parts list. The parts
218: (8)                    list is a list of `.CompositePart` entries describing the parts of
219: (8)                    the composite.
220: (4)                Examples
221: (4)                --------
222: (4)                A composite definition line::
223: (6)                  CC Aacute 2 ; PCC A 0 0 ; PCC acute 160 170 ;
224: (4)                will be represented as::
225: (6)                  composites['Aacute'] = [CompositePart(name='A', dx=0, dy=0),
226: (30)                                          CompositePart(name='acute', dx=160, dy=170)]
227: (4)                """
228: (4)                composites = {}
229: (4)                for line in fh:
230: (8)                    line = line.rstrip()
231: (8)                    if not line:
232: (12)                       continue
233: (8)                    if line.startswith(b'EndComposites'):
234: (12)                       return composites
235: (8)                    vals = line.split(b';')
236: (8)                    cc = vals[0].split()
237: (8)                    name, _num_parts = cc[1], _to_int(cc[2])
238: (8)                    pccParts = []
239: (8)                    for s in vals[1:-1]:
240: (12)                       pcc = s.split()
241: (12)                       part = CompositePart(pcc[1], _to_float(pcc[2]), _to_float(pcc[3]))
242: (12)                       pccParts.append(part)
243: (8)                    composites[name] = pccParts
244: (4)                raise RuntimeError('Bad composites parse')
245: (0)            def _parse_optional(fh):
246: (4)                """
247: (4)                Parse the optional fields for kern pair data and composites.
248: (4)                Returns
249: (4)                -------
```

```
(4)                 kern_data : dict
(8)                     A dict containing kerning information. May be empty.
(8)                     See `._parse_kern_pairs`.
(4)                 composites : dict
(8)                     A dict containing composite information. May be empty.
(8)                     See `._parse_composites`.
(4)                 """
(4)                 optional = {
(8)                     b'StartKernData': _parse_kern_pairs,
(8)                     b'StartComposites':  _parse_composites,
(8)                     }
(4)                 d = {b'StartKernData': {},
(9)                      b'StartComposites': {}}
(4)                 for line in fh:
(8)                     line = line.rstrip()
(8)                     if not line:
(12)                        continue
(8)                     key = line.split()[0]
(8)                     if key in optional:
(12)                        d[key] = optional[key](fh)
(4)                 return d[b'StartKernData'], d[b'StartComposites']
(0)             class AFM:
(4)                 def __init__(self, fh):
(8)                     """Parse the AFM file in file object *fh*."""
(8)                     self._header = _parse_header(fh)
(8)                     self._metrics, self._metrics_by_name = _parse_char_metrics(fh)
(8)                     self._kern, self._composite = _parse_optional(fh)
(4)                 def get_bbox_char(self, c, isord=False):
(8)                     if not isord:
(12)                        c = ord(c)
(8)                     return self._metrics[c].bbox
(4)                 def string_width_height(self, s):
(8)                     """
(8)                     Return the string width (including kerning) and string height
(8)                     as a (*w*, *h*) tuple.
(8)                     """
(8)                     if not len(s):
(12)                        return 0, 0
(8)                     total_width = 0
(8)                     namelast = None
(8)                     miny = 1e9
(8)                     maxy = 0
(8)                     for c in s:
(12)                        if c == '\n':
(16)                            continue
(12)                        wx, name, bbox = self._metrics[ord(c)]
(12)                        total_width += wx + self._kern.get((namelast, name), 0)
(12)                        l, b, w, h = bbox
(12)                        miny = min(miny, b)
(12)                        maxy = max(maxy, b + h)
(12)                        namelast = name
(8)                     return total_width, maxy - miny
(4)                 def get_str_bbox_and_descent(self, s):
(8)                     """Return the string bounding box and the maximal descent."""
(8)                     if not len(s):
(12)                        return 0, 0, 0, 0, 0
(8)                     total_width = 0
(8)                     namelast = None
(8)                     miny = 1e9
(8)                     maxy = 0
(8)                     left = 0
(8)                     if not isinstance(s, str):
(12)                        s = _to_str(s)
(8)                     for c in s:
(12)                        if c == '\n':
(16)                            continue
(12)                        name = uni2type1.get(ord(c), f"uni{ord(c):04X}")
(12)                        try:
(16)                            wx, _, bbox = self._metrics_by_name[name]
```

```
(12)                        except KeyError:
(16)                            name = 'question'
(16)                            wx, _, bbox = self._metrics_by_name[name]
(12)                        total_width += wx + self._kern.get((namelast, name), 0)
(12)                        l, b, w, h = bbox
(12)                        left = min(left, l)
(12)                        miny = min(miny, b)
(12)                        maxy = max(maxy, b + h)
(12)                        namelast = name
(8)                     return left, miny, total_width, maxy - miny, -miny
(4)                 def get_str_bbox(self, s):
(8)                     """Return the string bounding box."""
(8)                     return self.get_str_bbox_and_descent(s)[:4]
(4)                 def get_name_char(self, c, isord=False):
(8)                     """Get the name of the character, i.e., ';' is 'semicolon'."""
(8)                     if not isord:
(12)                        c = ord(c)
(8)                     return self._metrics[c].name
(4)                 def get_width_char(self, c, isord=False):
(8)                     """
(8)                     Get the width of the character from the character metric WX field.
(8)                     """
(8)                     if not isord:
(12)                        c = ord(c)
(8)                     return self._metrics[c].width
(4)                 def get_width_from_char_name(self, name):
(8)                     """Get the width of the character from a type1 character name."""
(8)                     return self._metrics_by_name[name].width
(4)                 def get_height_char(self, c, isord=False):
(8)                     """Get the bounding box (ink) height of character *c* (space is 0)."""
(8)                     if not isord:
(12)                        c = ord(c)
(8)                     return self._metrics[c].bbox[-1]
(4)                 def get_kern_dist(self, c1, c2):
(8)                     """
(8)                     Return the kerning pair distance (possibly 0) for chars *c1* and *c2*.
(8)                     """
(8)                     name1, name2 = self.get_name_char(c1), self.get_name_char(c2)
(8)                     return self.get_kern_dist_from_name(name1, name2)
(4)                 def get_kern_dist_from_name(self, name1, name2):
(8)                     """
(8)                     Return the kerning pair distance (possibly 0) for chars
(8)                     *name1* and *name2*.
(8)                     """
(8)                     return self._kern.get((name1, name2), 0)
(4)                 def get_fontname(self):
(8)                     """Return the font name, e.g., 'Times-Roman'."""
(8)                     return self._header[b'FontName']
(4)                 @property
(4)                 def postscript_name(self):  # For consistency with FT2Font.
(8)                     return self.get_fontname()
(4)                 def get_fullname(self):
(8)                     """Return the font full name, e.g., 'Times-Roman'."""
(8)                     name = self._header.get(b'FullName')
(8)                     if name is None:  # use FontName as a substitute
(12)                        name = self._header[b'FontName']
(8)                     return name
(4)                 def get_familyname(self):
(8)                     """Return the font family name, e.g., 'Times'."""
(8)                     name = self._header.get(b'FamilyName')
(8)                     if name is not None:
(12)                        return name
(8)                     name = self.get_fullname()
(8)                     extras = (r'(?i)([ -](regular|plain|italic|oblique|bold|semibold|'
(18)                              r'light|ultralight|extra|condensed))+$')
(8)                     return re.sub(extras, '', name)
(4)                 @property
(4)                 def family_name(self):
(8)                     """The font family name, e.g., 'Times'."""
```

```
388: (8)                    return self.get_familyname()
389: (4)                def get_weight(self):
390: (8)                    """Return the font weight, e.g., 'Bold' or 'Roman'."""
391: (8)                    return self._header[b'Weight']
392: (4)                def get_angle(self):
393: (8)                    """Return the fontangle as float."""
394: (8)                    return self._header[b'ItalicAngle']
395: (4)                def get_capheight(self):
396: (8)                    """Return the cap height as float."""
397: (8)                    return self._header[b'CapHeight']
398: (4)                def get_xheight(self):
399: (8)                    """Return the xheight as float."""
400: (8)                    return self._header[b'XHeight']
401: (4)                def get_underline_thickness(self):
402: (8)                    """Return the underline thickness as float."""
403: (8)                    return self._header[b'UnderlineThickness']
404: (4)                def get_horizontal_stem_width(self):
405: (8)                    """
406: (8)                    Return the standard horizontal stem width as float, or *None* if
407: (8)                    not specified in AFM file.
408: (8)                    """
409: (8)                    return self._header.get(b'StdHW', None)
410: (4)                def get_vertical_stem_width(self):
411: (8)                    """
412: (8)                    Return the standard vertical stem width as float, or *None* if
413: (8)                    not specified in AFM file.
414: (8)                    """
415: (8)                    return self._header.get(b'StdVW', None)
```

File 3 - _animation_data.py:

```
1: (0)              JS_INCLUDE = """
2: (0)              <link rel="stylesheet"
3: (0)              href="https://maxcdn.bootstrapcdn.com/font-awesome/4.4.0/css/font-
awesome.min.css">
4: (0)              <script language="javascript">
5: (2)                function isInternetExplorer() {
6: (4)                  ua = navigator.userAgent;
7: (4)                  /* MSIE used to detect old browsers and Trident used to newer ones*/
8: (4)                  return ua.indexOf("MSIE ") > -1 || ua.indexOf("Trident/") > -1;
9: (2)                }
10: (2)               /* Define the Animation class */
11: (2)               function Animation(frames, img_id, slider_id, interval, loop_select_id){
12: (4)                 this.img_id = img_id;
13: (4)                 this.slider_id = slider_id;
14: (4)                 this.loop_select_id = loop_select_id;
15: (4)                 this.interval = interval;
16: (4)                 this.current_frame = 0;
17: (4)                 this.direction = 0;
18: (4)                 this.timer = null;
19: (4)                 this.frames = new Array(frames.length);
20: (4)                 for (var i=0; i<frames.length; i++)
21: (4)                 {
22: (5)                  this.frames[i] = new Image();
23: (5)                  this.frames[i].src = frames[i];
24: (4)                 }
25: (4)                 var slider = document.getElementById(this.slider_id);
26: (4)                 slider.max = this.frames.length - 1;
27: (4)                 if (isInternetExplorer()) {
28: (8)                     // switch from oninput to onchange because IE <= 11 does not conform
29: (8)                     // with W3C specification. It ignores oninput and onchange behaves
30: (8)                     // like oninput. In contrast, Microsoft Edge behaves correctly.
31: (8)                     slider.setAttribute('onchange', slider.getAttribute('oninput'));
32: (8)                     slider.setAttribute('oninput', null);
33: (4)                 }
34: (4)                 this.set_frame(this.current_frame);
35: (2)               }
```

```
(2)  Animation.prototype.get_loop_state = function(){
(4)    var button_group = document[this.loop_select_id].state;
(4)    for (var i = 0; i < button_group.length; i++) {
(8)        var button = button_group[i];
(8)        if (button.checked) {
(12)           return button.value;
(8)        }
(4)    }
(4)    return undefined;
(2)  }
(2)  Animation.prototype.set_frame = function(frame){
(4)    this.current_frame = frame;
(4)    document.getElementById(this.img_id).src =
(12)           this.frames[this.current_frame].src;
(4)    document.getElementById(this.slider_id).value = this.current_frame;
(2)  }
(2)  Animation.prototype.next_frame = function()
(2)  {
(4)    this.set_frame(Math.min(this.frames.length - 1, this.current_frame + 1));
(2)  }
(2)  Animation.prototype.previous_frame = function()
(2)  {
(4)    this.set_frame(Math.max(0, this.current_frame - 1));
(2)  }
(2)  Animation.prototype.first_frame = function()
(2)  {
(4)    this.set_frame(0);
(2)  }
(2)  Animation.prototype.last_frame = function()
(2)  {
(4)    this.set_frame(this.frames.length - 1);
(2)  }
(2)  Animation.prototype.slower = function()
(2)  {
(4)    this.interval /= 0.7;
(4)    if(this.direction > 0){this.play_animation();}
(4)    else if(this.direction < 0){this.reverse_animation();}
(2)  }
(2)  Animation.prototype.faster = function()
(2)  {
(4)    this.interval *= 0.7;
(4)    if(this.direction > 0){this.play_animation();}
(4)    else if(this.direction < 0){this.reverse_animation();}
(2)  }
(2)  Animation.prototype.anim_step_forward = function()
(2)  {
(4)    this.current_frame += 1;
(4)    if(this.current_frame < this.frames.length){
(6)      this.set_frame(this.current_frame);
(4)    }else{
(6)      var loop_state = this.get_loop_state();
(6)      if(loop_state == "loop"){
(8)        this.first_frame();
(6)      }else if(loop_state == "reflect"){
(8)        this.last_frame();
(8)        this.reverse_animation();
(6)      }else{
(8)        this.pause_animation();
(8)        this.last_frame();
(6)      }
(4)    }
(2)  }
(2)  Animation.prototype.anim_step_reverse = function()
(2)  {
(4)    this.current_frame -= 1;
(4)    if(this.current_frame >= 0){
(6)      this.set_frame(this.current_frame);
(4)    }else{
(6)      var loop_state = this.get_loop_state();
```

```
(6)                 if(loop_state == "loop"){
(8)                   this.last_frame();
(6)                 }else if(loop_state == "reflect"){
(8)                   this.first_frame();
(8)                   this.play_animation();
(6)                 }else{
(8)                   this.pause_animation();
(8)                   this.first_frame();
(6)                 }
(4)               }
(2)             }
(2)             Animation.prototype.pause_animation = function()
(2)             {
(4)               this.direction = 0;
(4)               if (this.timer){
(6)                 clearInterval(this.timer);
(6)                 this.timer = null;
(4)               }
(2)             }
(2)             Animation.prototype.play_animation = function()
(2)             {
(4)               this.pause_animation();
(4)               this.direction = 1;
(4)               var t = this;
(4)               if (!this.timer) this.timer = setInterval(function() {
(8)                   t.anim_step_forward();
(4)               }, this.interval);
(2)             }
(2)             Animation.prototype.reverse_animation = function()
(2)             {
(4)               this.pause_animation();
(4)               this.direction = -1;
(4)               var t = this;
(4)               if (!this.timer) this.timer = setInterval(function() {
(8)                   t.anim_step_reverse();
(4)               }, this.interval);
(2)             }
(0)           </script>
(0)           """
(0)           STYLE_INCLUDE = """
(0)           <style>
(0)           .animation {
(4)               display: inline-block;
(4)               text-align: center;
(0)           }
(0)           input[type=range].anim-slider {
(4)               width: 374px;
(4)               margin-left: auto;
(4)               margin-right: auto;
(0)           }
(0)           .anim-buttons {
(4)               margin: 8px 0px;
(0)           }
(0)           .anim-buttons button {
(4)               padding: 0;
(4)               width: 36px;
(0)           }
(0)           .anim-state label {
(4)               margin-right: 8px;
(0)           }
(0)           .anim-state input {
(4)               margin: 0;
(4)               vertical-align: middle;
(0)           }
(0)           </style>
(0)           """
(0)           DISPLAY_TEMPLATE = """
(0)           <div class="animation">
(2)             <img id="_anim_img{id}">
```

```
174: (2)              <div class="anim-controls">
175: (4)                <input id="_anim_slider{id}" type="range" class="anim-slider"
176: (11)                      name="points" min="0" max="1" step="1" value="0"
177: (11)                      oninput="anim{id}.set_frame(parseInt(this.value));">
178: (4)                <div class="anim-buttons">
179: (6)                  <button title="Decrease speed" aria-label="Decrease speed"
onclick="anim{id}.slower()">
180: (10)                     <i class="fa fa-minus"></i></button>
181: (6)                  <button title="First frame" aria-label="First frame"
onclick="anim{id}.first_frame()">
182: (8)                    <i class="fa fa-fast-backward"></i></button>
183: (6)                  <button title="Previous frame" aria-label="Previous frame"
onclick="anim{id}.previous_frame()">
184: (10)                     <i class="fa fa-step-backward"></i></button>
185: (6)                  <button title="Play backwards" aria-label="Play backwards"
onclick="anim{id}.reverse_animation()">
186: (10)                     <i class="fa fa-play fa-flip-horizontal"></i></button>
187: (6)                  <button title="Pause" aria-label="Pause"
onclick="anim{id}.pause_animation()">
188: (10)                     <i class="fa fa-pause"></i></button>
189: (6)                  <button title="Play" aria-label="Play"
onclick="anim{id}.play_animation()">
190: (10)                     <i class="fa fa-play"></i></button>
191: (6)                  <button title="Next frame" aria-label="Next frame"
onclick="anim{id}.next_frame()">
192: (10)                     <i class="fa fa-step-forward"></i></button>
193: (6)                  <button title="Last frame" aria-label="Last frame"
onclick="anim{id}.last_frame()">
194: (10)                     <i class="fa fa-fast-forward"></i></button>
195: (6)                  <button title="Increase speed" aria-label="Increase speed"
onclick="anim{id}.faster()">
196: (10)                     <i class="fa fa-plus"></i></button>
197: (4)                </div>
198: (4)                <form title="Repetition mode" aria-label="Repetition mode" action="#n"
name="_anim_loop_select{id}"
199: (10)                     class="anim-state">
200: (6)                  <input type="radio" name="state" value="once" id="_anim_radio1_{id}"
201: (13)                       {once_checked}>
202: (6)                  <label for="_anim_radio1_{id}">Once</label>
203: (6)                  <input type="radio" name="state" value="loop" id="_anim_radio2_{id}"
204: (13)                       {loop_checked}>
205: (6)                  <label for="_anim_radio2_{id}">Loop</label>
206: (6)                  <input type="radio" name="state" value="reflect" id="_anim_radio3_{id}"
207: (13)                       {reflect_checked}>
208: (6)                  <label for="_anim_radio3_{id}">Reflect</label>
209: (4)                </form>
210: (2)              </div>
211: (0)            </div>
212: (0)            <script language="javascript">
213: (2)              /* Instantiate the Animation class. */
214: (2)              /* The IDs given should match those used in the template above. */
215: (2)              (function() {{
216: (4)                var img_id = "_anim_img{id}";
217: (4)                var slider_id = "_anim_slider{id}";
218: (4)                var loop_select_id = "_anim_loop_select{id}";
219: (4)                var frames = new Array({Nframes});
220: (4)                {fill_frames}
221: (4)                /* set a timeout to make sure all the above elements are created before
222: (7)                   the object is initialized. */
223: (4)                setTimeout(function() {{
224: (8)                    anim{id} = new Animation(frames, img_id, slider_id, {interval},
225: (33)                                             loop_select_id);
226: (4)                }}, 0);
227: (2)              }})()
228: (0)            </script>
229: (0)            """  # noqa: E501
230: (0)            INCLUDED_FRAMES = """
231: (2)              for (var i=0; i<{Nframes}; i++){{
232: (4)                frames[i] = "{frame_dir}/frame" + ("0000000" + i).slice(-7) +
```

```
233: (16)                            ".{frame_format}";
234: (2)              }}
235: (0)            """
```

File 4 - _blocking_input.py:

```
1: (0)              def blocking_input_loop(figure, event_names, timeout, handler):
2: (4)                  """
3: (4)                  Run *figure*'s event loop while listening to interactive events.
4: (4)                  The events listed in *event_names* are passed to *handler*.
5: (4)                  This function is used to implement `.Figure.waitforbuttonpress`,
6: (4)                  `.Figure.ginput`, and `.Axes.clabel`.
7: (4)                  Parameters
8: (4)                  ----------
9: (4)                  figure : `~matplotlib.figure.Figure`
10: (4)                 event_names : list of str
11: (8)                     The names of the events passed to *handler*.
12: (4)                 timeout : float
13: (8)                     If positive, the event loop is stopped after *timeout* seconds.
14: (4)                 handler : Callable[[Event], Any]
15: (8)                     Function called for each event; it can force an early exit of the
event
16: (8)                     loop by calling ``canvas.stop_event_loop()``.
17: (4)                 """
18: (4)                 if figure.canvas.manager:
19: (8)                     figure.show()  # Ensure that the figure is shown if we are managing
it.
20: (4)                 cids = [figure.canvas.mpl_connect(name, handler) for name in event_names]
21: (4)                 try:
22: (8)                     figure.canvas.start_event_loop(timeout)  # Start event loop.
23: (4)                 finally:  # Run even on exception like ctrl-c.
24: (8)                     for cid in cids:
25: (12)                        figure.canvas.mpl_disconnect(cid)
```

File 5 - _cm.py:

```
1: (0)              """
2: (0)              Nothing here but dictionaries for generating LinearSegmentedColormaps,
3: (0)              and a dictionary of these dictionaries.
4: (0)              Documentation for each is in pyplot.colormaps().  Please update this
5: (0)              with the purpose and type of your colormap if you add data for one here.
6: (0)              """
7: (0)              from functools import partial
8: (0)              import numpy as np
9: (0)              _binary_data = {
10: (4)                 'red':    ((0., 1., 1.), (1., 0., 0.)),
11: (4)                 'green':  ((0., 1., 1.), (1., 0., 0.)),
12: (4)                 'blue':   ((0., 1., 1.), (1., 0., 0.))
13: (4)                 }
14: (0)             _autumn_data = {'red':   ((0., 1.0, 1.0), (1.0, 1.0, 1.0)),
15: (16)                            'green': ((0., 0., 0.), (1.0, 1.0, 1.0)),
16: (16)                            'blue':  ((0., 0., 0.), (1.0, 0., 0.))}
17: (0)             _bone_data = {'red':   ((0., 0., 0.),
18: (24)                                    (0.746032, 0.652778, 0.652778),
19: (24)                                    (1.0, 1.0, 1.0)),
20: (14)                          'green': ((0., 0., 0.),
21: (24)                                    (0.365079, 0.319444, 0.319444),
22: (24)                                    (0.746032, 0.777778, 0.777778),
23: (24)                                    (1.0, 1.0, 1.0)),
24: (14)                          'blue':  ((0., 0., 0.),
25: (24)                                    (0.365079, 0.444444, 0.444444),
26: (24)                                    (1.0, 1.0, 1.0))}
27: (0)             _cool_data = {'red':   ((0., 0., 0.), (1.0, 1.0, 1.0)),
28: (14)                          'green': ((0., 1., 1.), (1.0, 0.,  0.)),
29: (14)                          'blue':  ((0., 1., 1.), (1.0, 1.,  1.))}
```

```
30: (0)             _copper_data = {'red':   ((0., 0., 0.),
31: (26)                                      (0.809524, 1.000000, 1.000000),
32: (26)                                      (1.0, 1.0, 1.0)),
33: (16)                          'green': ((0., 0., 0.),
34: (26)                                      (1.0, 0.7812, 0.7812)),
35: (16)                          'blue':  ((0., 0., 0.),
36: (26)                                      (1.0, 0.4975, 0.4975))}
37: (0)             def _flag_red(x): return 0.75 * np.sin((x * 31.5 + 0.25) * np.pi) + 0.5
38: (0)             def _flag_green(x): return np.sin(x * 31.5 * np.pi)
39: (0)             def _flag_blue(x): return 0.75 * np.sin((x * 31.5 - 0.25) * np.pi) + 0.5
40: (0)             _flag_data = {'red': _flag_red, 'green': _flag_green, 'blue': _flag_blue}
41: (0)             def _prism_red(x): return 0.75 * np.sin((x * 20.9 + 0.25) * np.pi) + 0.67
42: (0)             def _prism_green(x): return 0.75 * np.sin((x * 20.9 - 0.25) * np.pi) + 0.33
43: (0)             def _prism_blue(x): return -1.1 * np.sin((x * 20.9) * np.pi)
44: (0)             _prism_data = {'red': _prism_red, 'green': _prism_green, 'blue': _prism_blue}
45: (0)             def _ch_helper(gamma, s, r, h, p0, p1, x):
46: (4)                 """Helper function for generating picklable cubehelix colormaps."""
47: (4)                 xg = x ** gamma
48: (4)                 a = h * xg * (1 - xg) / 2
49: (4)                 phi = 2 * np.pi * (s / 3 + r * x)
50: (4)                 return xg + a * (p0 * np.cos(phi) + p1 * np.sin(phi))
51: (0)             def cubehelix(gamma=1.0, s=0.5, r=-1.5, h=1.0):
52: (4)                 """
53: (4)                 Return custom data dictionary of (r, g, b) conversion functions, which can
54: (4)                 be used with `.ColormapRegistry.register`, for the cubehelix color scheme.
55: (4)                 Unlike most other color schemes cubehelix was designed by D.A. Green to
56: (4)                 be monotonically increasing in terms of perceived brightness.
57: (4)                 Also, when printed on a black and white postscript printer, the scheme
58: (4)                 results in a greyscale with monotonically increasing brightness.
59: (4)                 This color scheme is named cubehelix because the (r, g, b) values produced
60: (4)                 can be visualised as a squashed helix around the diagonal in the
61: (4)                 (r, g, b) color cube.
62: (4)                 For a unit color cube (i.e. 3D coordinates for (r, g, b) each in the
63: (4)                 range 0 to 1) the color scheme starts at (r, g, b) = (0, 0, 0), i.e.
black,
64: (4)                 and finishes at (r, g, b) = (1, 1, 1), i.e. white. For some fraction *x*,
65: (4)                 between 0 and 1, the color is the corresponding grey value at that
66: (4)                 fraction along the black to white diagonal (x, x, x) plus a color
67: (4)                 element. This color element is calculated in a plane of constant
68: (4)                 perceived intensity and controlled by the following parameters.
69: (4)                 Parameters
70: (4)                 ----------
71: (4)                 gamma : float, default: 1
72: (8)                     Gamma factor emphasizing either low intensity values (gamma < 1), or
73: (8)                     high intensity values (gamma > 1).
74: (4)                 s : float, default: 0.5 (purple)
75: (8)                     The starting color.
76: (4)                 r : float, default: -1.5
77: (8)                     The number of r, g, b rotations in color that are made from the start
78: (8)                     to the end of the color scheme.  The default of -1.5 corresponds to ->
79: (8)                     B -> G -> R -> B.
80: (4)                 h : float, default: 1
81: (8)                     The hue, i.e. how saturated the colors are. If this parameter is zero
82: (8)                     then the color scheme is purely a greyscale.
83: (4)                 """
84: (4)                 return {'red': partial(_ch_helper, gamma, s, r, h, -0.14861, 1.78277),
85: (12)                        'green': partial(_ch_helper, gamma, s, r, h, -0.29227, -0.90649),
86: (12)                        'blue': partial(_ch_helper, gamma, s, r, h, 1.97294, 0.0)}
87: (0)             _cubehelix_data = cubehelix()
88: (0)             _bwr_data = ((0.0, 0.0, 1.0), (1.0, 1.0, 1.0), (1.0, 0.0, 0.0))
89: (0)             _brg_data = ((0.0, 0.0, 1.0), (1.0, 0.0, 0.0), (0.0, 1.0, 0.0))
90: (0)             def _g0(x): return 0
91: (0)             def _g1(x): return 0.5
92: (0)             def _g2(x): return 1
93: (0)             def _g3(x): return x
94: (0)             def _g4(x): return x ** 2
95: (0)             def _g5(x): return x ** 3
96: (0)             def _g6(x): return x ** 4
97: (0)             def _g7(x): return np.sqrt(x)
```

```
(0)             def _g8(x): return np.sqrt(np.sqrt(x))
(0)             def _g9(x): return np.sin(x * np.pi / 2)
(0)             def _g10(x): return np.cos(x * np.pi / 2)
(0)             def _g11(x): return np.abs(x - 0.5)
(0)             def _g12(x): return (2 * x - 1) ** 2
(0)             def _g13(x): return np.sin(x * np.pi)
(0)             def _g14(x): return np.abs(np.cos(x * np.pi))
(0)             def _g15(x): return np.sin(x * 2 * np.pi)
(0)             def _g16(x): return np.cos(x * 2 * np.pi)
(0)             def _g17(x): return np.abs(np.sin(x * 2 * np.pi))
(0)             def _g18(x): return np.abs(np.cos(x * 2 * np.pi))
(0)             def _g19(x): return np.abs(np.sin(x * 4 * np.pi))
(0)             def _g20(x): return np.abs(np.cos(x * 4 * np.pi))
(0)             def _g21(x): return 3 * x
(0)             def _g22(x): return 3 * x - 1
(0)             def _g23(x): return 3 * x - 2
(0)             def _g24(x): return np.abs(3 * x - 1)
(0)             def _g25(x): return np.abs(3 * x - 2)
(0)             def _g26(x): return (3 * x - 1) / 2
(0)             def _g27(x): return (3 * x - 2) / 2
(0)             def _g28(x): return np.abs((3 * x - 1) / 2)
(0)             def _g29(x): return np.abs((3 * x - 2) / 2)
(0)             def _g30(x): return x / 0.32 - 0.78125
(0)             def _g31(x): return 2 * x - 0.84
(0)             def _g32(x):
(4)                 ret = np.zeros(len(x))
(4)                 m = (x < 0.25)
(4)                 ret[m] = 4 * x[m]
(4)                 m = (x >= 0.25) & (x < 0.92)
(4)                 ret[m] = -2 * x[m] + 1.84
(4)                 m = (x >= 0.92)
(4)                 ret[m] = x[m] / 0.08 - 11.5
(4)                 return ret
(0)             def _g33(x): return np.abs(2 * x - 0.5)
(0)             def _g34(x): return 2 * x
(0)             def _g35(x): return 2 * x - 0.5
(0)             def _g36(x): return 2 * x - 1
(0)             gfunc = {i: globals()[f"_g{i}"] for i in range(37)}
(0)             _gnuplot_data = {
(8)                     'red': gfunc[7],
(8)                     'green': gfunc[5],
(8)                     'blue': gfunc[15],
(0)             }
(0)             _gnuplot2_data = {
(8)                     'red': gfunc[30],
(8)                     'green': gfunc[31],
(8)                     'blue': gfunc[32],
(0)             }
(0)             _ocean_data = {
(8)                     'red': gfunc[23],
(8)                     'green': gfunc[28],
(8)                     'blue': gfunc[3],
(0)             }
(0)             _afmhot_data = {
(8)                     'red': gfunc[34],
(8)                     'green': gfunc[35],
(8)                     'blue': gfunc[36],
(0)             }
(0)             _rainbow_data = {
(8)                     'red': gfunc[33],
(8)                     'green': gfunc[13],
(8)                     'blue': gfunc[10],
(0)             }
(0)             _seismic_data = (
(8)                     (0.0, 0.0, 0.3), (0.0, 0.0, 1.0),
(8)                     (1.0, 1.0, 1.0), (1.0, 0.0, 0.0),
(8)                     (0.5, 0.0, 0.0))
(0)             _terrain_data = (
(8)                     (0.00, (0.2, 0.2, 0.6)),
```

```
(8)                     (0.15, (0.0, 0.6, 1.0)),
(8)                     (0.25, (0.0, 0.8, 0.4)),
(8)                     (0.50, (1.0, 1.0, 0.6)),
(8)                     (0.75, (0.5, 0.36, 0.33)),
(8)                     (1.00, (1.0, 1.0, 1.0)))
(0)             _gray_data = {'red':   ((0., 0, 0), (1., 1, 1)),
(14)                          'green': ((0., 0, 0), (1., 1, 1)),
(14)                          'blue':  ((0., 0, 0), (1., 1, 1))}
(0)             _hot_data = {'red':   ((0., 0.0416, 0.0416),
(23)                                   (0.365079, 1.000000, 1.000000),
(23)                                   (1.0, 1.0, 1.0)),
(13)                         'green': ((0., 0., 0.),
(23)                                   (0.365079, 0.000000, 0.000000),
(23)                                   (0.746032, 1.000000, 1.000000),
(23)                                   (1.0, 1.0, 1.0)),
(13)                         'blue':  ((0., 0., 0.),
(23)                                   (0.746032, 0.000000, 0.000000),
(23)                                   (1.0, 1.0, 1.0))}
(0)             _hsv_data = {'red':   ((0., 1., 1.),
(23)                                   (0.158730, 1.000000, 1.000000),
(23)                                   (0.174603, 0.968750, 0.968750),
(23)                                   (0.333333, 0.031250, 0.031250),
(23)                                   (0.349206, 0.000000, 0.000000),
(23)                                   (0.666667, 0.000000, 0.000000),
(23)                                   (0.682540, 0.031250, 0.031250),
(23)                                   (0.841270, 0.968750, 0.968750),
(23)                                   (0.857143, 1.000000, 1.000000),
(23)                                   (1.0, 1.0, 1.0)),
(13)                         'green': ((0., 0., 0.),
(23)                                   (0.158730, 0.937500, 0.937500),
(23)                                   (0.174603, 1.000000, 1.000000),
(23)                                   (0.507937, 1.000000, 1.000000),
(23)                                   (0.666667, 0.062500, 0.062500),
(23)                                   (0.682540, 0.000000, 0.000000),
(23)                                   (1.0, 0., 0.)),
(13)                         'blue':  ((0., 0., 0.),
(23)                                   (0.333333, 0.000000, 0.000000),
(23)                                   (0.349206, 0.062500, 0.062500),
(23)                                   (0.507937, 1.000000, 1.000000),
(23)                                   (0.841270, 1.000000, 1.000000),
(23)                                   (0.857143, 0.937500, 0.937500),
(23)                                   (1.0, 0.09375, 0.09375))}
(0)             _jet_data = {'red':   ((0.00, 0, 0),
(23)                                   (0.35, 0, 0),
(23)                                   (0.66, 1, 1),
(23)                                   (0.89, 1, 1),
(23)                                   (1.00, 0.5, 0.5)),
(13)                         'green': ((0.000, 0, 0),
(23)                                   (0.125, 0, 0),
(23)                                   (0.375, 1, 1),
(23)                                   (0.640, 1, 1),
(23)                                   (0.910, 0, 0),
(23)                                   (1.000, 0, 0)),
(13)                         'blue':  ((0.00, 0.5, 0.5),
(23)                                   (0.11, 1, 1),
(23)                                   (0.34, 1, 1),
(23)                                   (0.65, 0, 0),
(23)                                   (1.00, 0, 0))}
(0)             _pink_data = {'red':   ((0., 0.1178, 0.1178), (0.015873, 0.195857, 0.195857),
(24)                                    (0.031746, 0.250661, 0.250661),
(24)                                    (0.047619, 0.295468, 0.295468),
(24)                                    (0.063492, 0.334324, 0.334324),
(24)                                    (0.079365, 0.369112, 0.369112),
(24)                                    (0.095238, 0.400892, 0.400892),
(24)                                    (0.111111, 0.430331, 0.430331),
(24)                                    (0.126984, 0.457882, 0.457882),
(24)                                    (0.142857, 0.483867, 0.483867),
(24)                                    (0.158730, 0.508525, 0.508525),
(24)                                    (0.174603, 0.532042, 0.532042),
```

```
236: (24)                                   (0.190476, 0.554563, 0.554563),
237: (24)                                   (0.206349, 0.576204, 0.576204),
238: (24)                                   (0.222222, 0.597061, 0.597061),
239: (24)                                   (0.238095, 0.617213, 0.617213),
240: (24)                                   (0.253968, 0.636729, 0.636729),
241: (24)                                   (0.269841, 0.655663, 0.655663),
242: (24)                                   (0.285714, 0.674066, 0.674066),
243: (24)                                   (0.301587, 0.691980, 0.691980),
244: (24)                                   (0.317460, 0.709441, 0.709441),
245: (24)                                   (0.333333, 0.726483, 0.726483),
246: (24)                                   (0.349206, 0.743134, 0.743134),
247: (24)                                   (0.365079, 0.759421, 0.759421),
248: (24)                                   (0.380952, 0.766356, 0.766356),
249: (24)                                   (0.396825, 0.773229, 0.773229),
250: (24)                                   (0.412698, 0.780042, 0.780042),
251: (24)                                   (0.428571, 0.786796, 0.786796),
252: (24)                                   (0.444444, 0.793492, 0.793492),
253: (24)                                   (0.460317, 0.800132, 0.800132),
254: (24)                                   (0.476190, 0.806718, 0.806718),
255: (24)                                   (0.492063, 0.813250, 0.813250),
256: (24)                                   (0.507937, 0.819730, 0.819730),
257: (24)                                   (0.523810, 0.826160, 0.826160),
258: (24)                                   (0.539683, 0.832539, 0.832539),
259: (24)                                   (0.555556, 0.838870, 0.838870),
260: (24)                                   (0.571429, 0.845154, 0.845154),
261: (24)                                   (0.587302, 0.851392, 0.851392),
262: (24)                                   (0.603175, 0.857584, 0.857584),
263: (24)                                   (0.619048, 0.863731, 0.863731),
264: (24)                                   (0.634921, 0.869835, 0.869835),
265: (24)                                   (0.650794, 0.875897, 0.875897),
266: (24)                                   (0.666667, 0.881917, 0.881917),
267: (24)                                   (0.682540, 0.887896, 0.887896),
268: (24)                                   (0.698413, 0.893835, 0.893835),
269: (24)                                   (0.714286, 0.899735, 0.899735),
270: (24)                                   (0.730159, 0.905597, 0.905597),
271: (24)                                   (0.746032, 0.911421, 0.911421),
272: (24)                                   (0.761905, 0.917208, 0.917208),
273: (24)                                   (0.777778, 0.922958, 0.922958),
274: (24)                                   (0.793651, 0.928673, 0.928673),
275: (24)                                   (0.809524, 0.934353, 0.934353),
276: (24)                                   (0.825397, 0.939999, 0.939999),
277: (24)                                   (0.841270, 0.945611, 0.945611),
278: (24)                                   (0.857143, 0.951190, 0.951190),
279: (24)                                   (0.873016, 0.956736, 0.956736),
280: (24)                                   (0.888889, 0.962250, 0.962250),
281: (24)                                   (0.904762, 0.967733, 0.967733),
282: (24)                                   (0.920635, 0.973185, 0.973185),
283: (24)                                   (0.936508, 0.978607, 0.978607),
284: (24)                                   (0.952381, 0.983999, 0.983999),
285: (24)                                   (0.968254, 0.989361, 0.989361),
286: (24)                                   (0.984127, 0.994695, 0.994695), (1.0, 1.0, 1.0)),
287: (14)                         'green': ((0., 0., 0.), (0.015873, 0.102869, 0.102869),
288: (24)                                   (0.031746, 0.145479, 0.145479),
289: (24)                                   (0.047619, 0.178174, 0.178174),
290: (24)                                   (0.063492, 0.205738, 0.205738),
291: (24)                                   (0.079365, 0.230022, 0.230022),
292: (24)                                   (0.095238, 0.251976, 0.251976),
293: (24)                                   (0.111111, 0.272166, 0.272166),
294: (24)                                   (0.126984, 0.290957, 0.290957),
295: (24)                                   (0.142857, 0.308607, 0.308607),
296: (24)                                   (0.158730, 0.325300, 0.325300),
297: (24)                                   (0.174603, 0.341178, 0.341178),
298: (24)                                   (0.190476, 0.356348, 0.356348),
299: (24)                                   (0.206349, 0.370899, 0.370899),
300: (24)                                   (0.222222, 0.384900, 0.384900),
301: (24)                                   (0.238095, 0.398410, 0.398410),
302: (24)                                   (0.253968, 0.411476, 0.411476),
303: (24)                                   (0.269841, 0.424139, 0.424139),
304: (24)                                   (0.285714, 0.436436, 0.436436),
```

```
305: (24)                                   (0.301587, 0.448395, 0.448395),
306: (24)                                   (0.317460, 0.460044, 0.460044),
307: (24)                                   (0.333333, 0.471405, 0.471405),
308: (24)                                   (0.349206, 0.482498, 0.482498),
309: (24)                                   (0.365079, 0.493342, 0.493342),
310: (24)                                   (0.380952, 0.517549, 0.517549),
311: (24)                                   (0.396825, 0.540674, 0.540674),
312: (24)                                   (0.412698, 0.562849, 0.562849),
313: (24)                                   (0.428571, 0.584183, 0.584183),
314: (24)                                   (0.444444, 0.604765, 0.604765),
315: (24)                                   (0.460317, 0.624669, 0.624669),
316: (24)                                   (0.476190, 0.643958, 0.643958),
317: (24)                                   (0.492063, 0.662687, 0.662687),
318: (24)                                   (0.507937, 0.680900, 0.680900),
319: (24)                                   (0.523810, 0.698638, 0.698638),
320: (24)                                   (0.539683, 0.715937, 0.715937),
321: (24)                                   (0.555556, 0.732828, 0.732828),
322: (24)                                   (0.571429, 0.749338, 0.749338),
323: (24)                                   (0.587302, 0.765493, 0.765493),
324: (24)                                   (0.603175, 0.781313, 0.781313),
325: (24)                                   (0.619048, 0.796819, 0.796819),
326: (24)                                   (0.634921, 0.812029, 0.812029),
327: (24)                                   (0.650794, 0.826960, 0.826960),
328: (24)                                   (0.666667, 0.841625, 0.841625),
329: (24)                                   (0.682540, 0.856040, 0.856040),
330: (24)                                   (0.698413, 0.870216, 0.870216),
331: (24)                                   (0.714286, 0.884164, 0.884164),
332: (24)                                   (0.730159, 0.897896, 0.897896),
333: (24)                                   (0.746032, 0.911421, 0.911421),
334: (24)                                   (0.761905, 0.917208, 0.917208),
335: (24)                                   (0.777778, 0.922958, 0.922958),
336: (24)                                   (0.793651, 0.928673, 0.928673),
337: (24)                                   (0.809524, 0.934353, 0.934353),
338: (24)                                   (0.825397, 0.939999, 0.939999),
339: (24)                                   (0.841270, 0.945611, 0.945611),
340: (24)                                   (0.857143, 0.951190, 0.951190),
341: (24)                                   (0.873016, 0.956736, 0.956736),
342: (24)                                   (0.888889, 0.962250, 0.962250),
343: (24)                                   (0.904762, 0.967733, 0.967733),
344: (24)                                   (0.920635, 0.973185, 0.973185),
345: (24)                                   (0.936508, 0.978607, 0.978607),
346: (24)                                   (0.952381, 0.983999, 0.983999),
347: (24)                                   (0.968254, 0.989361, 0.989361),
348: (24)                                   (0.984127, 0.994695, 0.994695), (1.0, 1.0, 1.0)),
349: (14)                         'blue':  ((0., 0., 0.), (0.015873, 0.102869, 0.102869),
350: (24)                                   (0.031746, 0.145479, 0.145479),
351: (24)                                   (0.047619, 0.178174, 0.178174),
352: (24)                                   (0.063492, 0.205738, 0.205738),
353: (24)                                   (0.079365, 0.230022, 0.230022),
354: (24)                                   (0.095238, 0.251976, 0.251976),
355: (24)                                   (0.111111, 0.272166, 0.272166),
356: (24)                                   (0.126984, 0.290957, 0.290957),
357: (24)                                   (0.142857, 0.308607, 0.308607),
358: (24)                                   (0.158730, 0.325300, 0.325300),
359: (24)                                   (0.174603, 0.341178, 0.341178),
360: (24)                                   (0.190476, 0.356348, 0.356348),
361: (24)                                   (0.206349, 0.370899, 0.370899),
362: (24)                                   (0.222222, 0.384900, 0.384900),
363: (24)                                   (0.238095, 0.398410, 0.398410),
364: (24)                                   (0.253968, 0.411476, 0.411476),
365: (24)                                   (0.269841, 0.424139, 0.424139),
366: (24)                                   (0.285714, 0.436436, 0.436436),
367: (24)                                   (0.301587, 0.448395, 0.448395),
368: (24)                                   (0.317460, 0.460044, 0.460044),
369: (24)                                   (0.333333, 0.471405, 0.471405),
370: (24)                                   (0.349206, 0.482498, 0.482498),
371: (24)                                   (0.365079, 0.493342, 0.493342),
372: (24)                                   (0.380952, 0.503953, 0.503953),
373: (24)                                   (0.396825, 0.514344, 0.514344),
```

```
(24)                                    (0.412698, 0.524531, 0.524531),
(24)                                    (0.428571, 0.534522, 0.534522),
(24)                                    (0.444444, 0.544331, 0.544331),
(24)                                    (0.460317, 0.553966, 0.553966),
(24)                                    (0.476190, 0.563436, 0.563436),
(24)                                    (0.492063, 0.572750, 0.572750),
(24)                                    (0.507937, 0.581914, 0.581914),
(24)                                    (0.523810, 0.590937, 0.590937),
(24)                                    (0.539683, 0.599824, 0.599824),
(24)                                    (0.555556, 0.608581, 0.608581),
(24)                                    (0.571429, 0.617213, 0.617213),
(24)                                    (0.587302, 0.625727, 0.625727),
(24)                                    (0.603175, 0.634126, 0.634126),
(24)                                    (0.619048, 0.642416, 0.642416),
(24)                                    (0.634921, 0.650600, 0.650600),
(24)                                    (0.650794, 0.658682, 0.658682),
(24)                                    (0.666667, 0.666667, 0.666667),
(24)                                    (0.682540, 0.674556, 0.674556),
(24)                                    (0.698413, 0.682355, 0.682355),
(24)                                    (0.714286, 0.690066, 0.690066),
(24)                                    (0.730159, 0.697691, 0.697691),
(24)                                    (0.746032, 0.705234, 0.705234),
(24)                                    (0.761905, 0.727166, 0.727166),
(24)                                    (0.777778, 0.748455, 0.748455),
(24)                                    (0.793651, 0.769156, 0.769156),
(24)                                    (0.809524, 0.789314, 0.789314),
(24)                                    (0.825397, 0.808969, 0.808969),
(24)                                    (0.841270, 0.828159, 0.828159),
(24)                                    (0.857143, 0.846913, 0.846913),
(24)                                    (0.873016, 0.865261, 0.865261),
(24)                                    (0.888889, 0.883229, 0.883229),
(24)                                    (0.904762, 0.900837, 0.900837),
(24)                                    (0.920635, 0.918109, 0.918109),
(24)                                    (0.936508, 0.935061, 0.935061),
(24)                                    (0.952381, 0.951711, 0.951711),
(24)                                    (0.968254, 0.968075, 0.968075),
(24)                                    (0.984127, 0.984167, 0.984167), (1.0, 1.0, 1.0))}
(0)             _spring_data = {'red':   ((0., 1., 1.), (1.0, 1.0, 1.0)),
(16)                            'green': ((0., 0., 0.), (1.0, 1.0, 1.0)),
(16)                            'blue':  ((0., 1., 1.), (1.0, 0.0, 0.0))}
(0)             _summer_data = {'red':   ((0., 0., 0.), (1.0, 1.0, 1.0)),
(16)                            'green': ((0., 0.5, 0.5), (1.0, 1.0, 1.0)),
(16)                            'blue':  ((0., 0.4, 0.4), (1.0, 0.4, 0.4))}
(0)             _winter_data = {'red':   ((0., 0., 0.), (1.0, 0.0, 0.0)),
(16)                            'green': ((0., 0., 0.), (1.0, 1.0, 1.0)),
(16)                            'blue':  ((0., 1., 1.), (1.0, 0.5, 0.5))}
(0)             _nipy_spectral_data = {
(4)                 'red': [
(8)                     (0.0, 0.0, 0.0), (0.05, 0.4667, 0.4667),
(8)                     (0.10, 0.5333, 0.5333), (0.15, 0.0, 0.0),
(8)                     (0.20, 0.0, 0.0), (0.25, 0.0, 0.0),
(8)                     (0.30, 0.0, 0.0), (0.35, 0.0, 0.0),
(8)                     (0.40, 0.0, 0.0), (0.45, 0.0, 0.0),
(8)                     (0.50, 0.0, 0.0), (0.55, 0.0, 0.0),
(8)                     (0.60, 0.0, 0.0), (0.65, 0.7333, 0.7333),
(8)                     (0.70, 0.9333, 0.9333), (0.75, 1.0, 1.0),
(8)                     (0.80, 1.0, 1.0), (0.85, 1.0, 1.0),
(8)                     (0.90, 0.8667, 0.8667), (0.95, 0.80, 0.80),
(8)                     (1.0, 0.80, 0.80),
(4)                 ],
(4)                 'green': [
(8)                     (0.0, 0.0, 0.0), (0.05, 0.0, 0.0),
(8)                     (0.10, 0.0, 0.0), (0.15, 0.0, 0.0),
(8)                     (0.20, 0.0, 0.0), (0.25, 0.4667, 0.4667),
(8)                     (0.30, 0.6000, 0.6000), (0.35, 0.6667, 0.6667),
(8)                     (0.40, 0.6667, 0.6667), (0.45, 0.6000, 0.6000),
(8)                     (0.50, 0.7333, 0.7333), (0.55, 0.8667, 0.8667),
(8)                     (0.60, 1.0, 1.0), (0.65, 1.0, 1.0),
(8)                     (0.70, 0.9333, 0.9333), (0.75, 0.8000, 0.8000),
```

```
(8)                 (0.80, 0.6000, 0.6000), (0.85, 0.0, 0.0),
(8)                 (0.90, 0.0, 0.0), (0.95, 0.0, 0.0),
(8)                 (1.0, 0.80, 0.80),
(4)             ],
(4)             'blue': [
(8)                 (0.0, 0.0, 0.0), (0.05, 0.5333, 0.5333),
(8)                 (0.10, 0.6000, 0.6000), (0.15, 0.6667, 0.6667),
(8)                 (0.20, 0.8667, 0.8667), (0.25, 0.8667, 0.8667),
(8)                 (0.30, 0.8667, 0.8667), (0.35, 0.6667, 0.6667),
(8)                 (0.40, 0.5333, 0.5333), (0.45, 0.0, 0.0),
(8)                 (0.5, 0.0, 0.0), (0.55, 0.0, 0.0),
(8)                 (0.60, 0.0, 0.0), (0.65, 0.0, 0.0),
(8)                 (0.70, 0.0, 0.0), (0.75, 0.0, 0.0),
(8)                 (0.80, 0.0, 0.0), (0.85, 0.0, 0.0),
(8)                 (0.90, 0.0, 0.0), (0.95, 0.0, 0.0),
(8)                 (1.0, 0.80, 0.80),
(4)             ],
(0)         }
(0)         _Blues_data = (
(4)             (0.96862745098039216,  0.98431372549019602,  1.0                ),
(4)             (0.87058823529411766,  0.92156862745098034,  0.96862745098039216),
(4)             (0.77647058823529413,  0.85882352941176465,  0.93725490196078431),
(4)             (0.61960784313725492,  0.792156862745098  ,  0.88235294117647056),
(4)             (0.41960784313725491,  0.68235294117647061,  0.83921568627450982),
(4)             (0.25882352941176473,  0.5725490196078431 ,  0.77647058823529413),
(4)             (0.12941176470588237,  0.44313725490196076,  0.70980392156862748),
(4)             (0.03137254901960784,  0.31764705882352939,  0.61176470588235299),
(4)             (0.03137254901960784,  0.18823529411764706,  0.41960784313725491)
(4)             )
(0)         _BrBG_data = (
(4)             (0.32941176470588235,  0.18823529411764706,  0.0196078431372549 ),
(4)             (0.5490196078431373 ,  0.31764705882352939,  0.0392156862745098 ),
(4)             (0.74901960784313726,  0.50588235294117645,  0.17647058823529413),
(4)             (0.87450980392156863,  0.76078431372549016,  0.49019607843137253),
(4)             (0.96470588235294119,  0.90980392156862744,  0.76470588235294112),
(4)             (0.96078431372549022,  0.96078431372549022,  0.96078431372549022),
(4)             (0.7803921568627451 ,  0.91764705882352937,  0.89803921568627454),
(4)             (0.50196078431372548,  0.80392156862745101,  0.75686274509803919),
(4)             (0.20784313725490197,  0.59215686274509804,  0.5607843137254902 ),
(4)             (0.00392156862745098,  0.4                ,  0.36862745098039218),
(4)             (0.0                ,  0.23529411764705882,  0.18823529411764706)
(4)             )
(0)         _BuGn_data = (
(4)             (0.96862745098039216,  0.9882352941176471 ,  0.99215686274509807),
(4)             (0.89803921568627454,  0.96078431372549022,  0.97647058823529409),
(4)             (0.8                ,  0.92549019607843142,  0.90196078431372551),
(4)             (0.6                ,  0.84705882352941175,  0.78823529411764703),
(4)             (0.4                ,  0.76078431372549016,  0.64313725490196083),
(4)             (0.25490196078431371,  0.68235294117647061,  0.46274509803921571),
(4)             (0.13725490196078433,  0.54509803921568623,  0.27058823529411763),
(4)             (0.0                ,  0.42745098039215684,  0.17254901960784313),
(4)             (0.0                ,  0.26666666666666666,  0.10588235294117647)
(4)             )
(0)         _BuPu_data = (
(4)             (0.96862745098039216,  0.9882352941176471 ,  0.99215686274509807),
(4)             (0.8784313725490196 ,  0.92549019607843142,  0.95686274509803926),
(4)             (0.74901960784313726,  0.82745098039215681,  0.90196078431372551),
(4)             (0.61960784313725492,  0.73725490196078436,  0.85490196078431369),
(4)             (0.5490196078431373 ,  0.58823529411764708,  0.77647058823529413),
(4)             (0.5490196078431373 ,  0.41960784313725491,  0.69411764705882351),
(4)             (0.53333333333333333,  0.25490196078431371,  0.61568627450980395),
(4)             (0.50588235294117645,  0.05882352941176471,  0.48627450980392156),
(4)             (0.30196078431372547,  0.0                ,  0.29411764705882354)
(4)             )
(0)         _GnBu_data = (
(4)             (0.96862745098039216,  0.9882352941176471 ,  0.94117647058823528),
(4)             (0.8784313725490196 ,  0.95294117647058818,  0.85882352941176465),
(4)             (0.8                ,  0.92156862745098034,  0.77254901960784317),
(4)             (0.6588235294117647 ,  0.8666666666666667 ,  0.70980392156862748),
```

```
(4)                 (0.4823529411764706 ,  0.8                 ,  0.7686274509803922 ),
(4)                 (0.30588235294117649,  0.70196078431372544,  0.82745098039215681),
(4)                 (0.16862745098039217,  0.5490196078431373 ,  0.74509803921568629),
(4)                 (0.03137254901960784,  0.40784313725490196,  0.67450980392156867),
(4)                 (0.03137254901960784,  0.25098039215686274,  0.50588235294117645)
(4)                 )
(0)             _Greens_data = (
(4)                 (0.96862745098039216,  0.9882352941176471 ,  0.96078431372549022),
(4)                 (0.89803921568627454,  0.96078431372549022,  0.8784313725490196 ),
(4)                 (0.7803921568627451 ,  0.9137254901960784 ,  0.75294117647058822),
(4)                 (0.63137254901960782,  0.85098039215686272,  0.60784313725490191),
(4)                 (0.45490196078431372,  0.7686274509803922 ,  0.46274509803921571),
(4)                 (0.25490196078431371,  0.6705882352941176 ,  0.36470588235294116),
(4)                 (0.13725490196078433,  0.54509803921568623,  0.27058823529411763),
(4)                 (0.0                ,  0.42745098039215684,  0.17254901960784313),
(4)                 (0.0                ,  0.26666666666666666,  0.10588235294117647)
(4)                 )
(0)             _Greys_data = (
(4)                 (1.0                ,  1.0                ,  1.0                ),
(4)                 (0.94117647058823528,  0.94117647058823528,  0.94117647058823528),
(4)                 (0.85098039215686272,  0.85098039215686272,  0.85098039215686272),
(4)                 (0.74117647058823533,  0.74117647058823533,  0.74117647058823533),
(4)                 (0.58823529411764708,  0.58823529411764708,  0.58823529411764708),
(4)                 (0.45098039215686275,  0.45098039215686275,  0.45098039215686275),
(4)                 (0.32156862745098042,  0.32156862745098042,  0.32156862745098042),
(4)                 (0.14509803921568629,  0.14509803921568629,  0.14509803921568629),
(4)                 (0.0                ,  0.0                ,  0.0                )
(4)                 )
(0)             _Oranges_data = (
(4)                 (1.0                ,  0.96078431372549022,  0.92156862745098034),
(4)                 (0.99607843137254903,  0.90196078431372551,  0.80784313725490198),
(4)                 (0.99215686274509807,  0.81568627450980391,  0.63529411764705879),
(4)                 (0.99215686274509807,  0.68235294117647061,  0.41960784313725491),
(4)                 (0.99215686274509807,  0.55294117647058827,  0.23529411764705882),
(4)                 (0.94509803921568625,  0.41176470588235292,  0.07450980392156863),
(4)                 (0.85098039215686272,  0.28235294117647058,  0.00392156862745098),
(4)                 (0.65098039215686276,  0.21176470588235294,  0.01176470588235294),
(4)                 (0.49803921568627452,  0.15294117647058825,  0.01568627450980392)
(4)                 )
(0)             _OrRd_data = (
(4)                 (1.0                ,  0.96862745098039216,  0.92549019607843142),
(4)                 (0.99607843137254903,  0.90980392156862744,  0.78431372549019607),
(4)                 (0.99215686274509807,  0.83137254901960789,  0.61960784313725492),
(4)                 (0.99215686274509807,  0.73333333333333328,  0.51764705882352946),
(4)                 (0.9882352941176471 ,  0.55294117647058827,  0.34901960784313724),
(4)                 (0.93725490196078431,  0.396078431372549  ,  0.28235294117647058),
(4)                 (0.84313725490196079,  0.18823529411764706,  0.12156862745098039),
(4)                 (0.70196078431372544,  0.0                ,  0.0                ),
(4)                 (0.49803921568627452,  0.0                ,  0.0                )
(4)                 )
(0)             _PiYG_data = (
(4)                 (0.55686274509803924,  0.00392156862745098,  0.32156862745098042),
(4)                 (0.77254901960784317,  0.10588235294117647,  0.49019607843137253),
(4)                 (0.87058823529411766,  0.46666666666666667,  0.68235294117647061),
(4)                 (0.94509803921568625,  0.71372549019607845,  0.85490196078431369),
(4)                 (0.99215686274509807,  0.8784313725490196 ,  0.93725490196078431),
(4)                 (0.96862745098039216,  0.96862745098039216,  0.96862745098039216),
(4)                 (0.90196078431372551,  0.96078431372549022,  0.81568627450980391),
(4)                 (0.72156862745098038,  0.88235294117647056,  0.52549019607843139),
(4)                 (0.49803921568627452,  0.73725490196078436,  0.25490196078431371),
(4)                 (0.30196078431372547,  0.5725490196078431 ,  0.12941176470588237),
(4)                 (0.15294117647058825,  0.39215686274509803,  0.09803921568627451)
(4)                 )
(0)             _PRGn_data = (
(4)                 (0.25098039215686274,  0.0                ,  0.29411764705882354),
(4)                 (0.46274509803921571,  0.16470588235294117,  0.51372549019607838),
(4)                 (0.6                ,  0.4392156862745098 ,  0.6705882352941176 ),
(4)                 (0.76078431372549016,  0.6470588235294118 ,  0.81176470588235294),
(4)                 (0.90588235294117647,  0.83137254901960789,  0.90980392156862744),
```

```
(4)                 (0.96862745098039216,  0.96862745098039216,  0.96862745098039216),
(4)                 (0.85098039215686272,  0.94117647058823528,  0.82745098039215681),
(4)                 (0.65098039215686276,  0.85882352941176465,  0.62745098039215685),
(4)                 (0.35294117647058826,  0.68235294117647061,  0.38039215686274508),
(4)                 (0.10588235294117647,  0.47058823529411764,  0.21568627450980393),
(4)                 (0.0                ,  0.26666666666666666,  0.10588235294117647)
(4)                 )
(0)             _PuBu_data = (
(4)                 (1.0                ,  0.96862745098039216,  0.98431372549019602),
(4)                 (0.92549019607843142,  0.90588235294117647,  0.94901960784313721),
(4)                 (0.81568627450980391,  0.81960784313725488,  0.90196078431372551),
(4)                 (0.65098039215686276,  0.74117647058823533,  0.85882352941176465),
(4)                 (0.45490196078431372,  0.66274509803921566,  0.81176470588235294),
(4)                 (0.21176470588235294,  0.56470588235294117,  0.75294117647058822),
(4)                 (0.0196078431372549 ,  0.4392156862745098 ,  0.69019607843137254),
(4)                 (0.01568627450980392,  0.35294117647058826,  0.55294117647058827),
(4)                 (0.00784313725490196,  0.2196078431372549 ,  0.34509803921568627)
(4)                 )
(0)             _PuBuGn_data = (
(4)                 (1.0                ,  0.96862745098039216,  0.98431372549019602),
(4)                 (0.92549019607843142,  0.88627450980392153,  0.94117647058823528),
(4)                 (0.81568627450980391,  0.81960784313725488,  0.90196078431372551),
(4)                 (0.65098039215686276,  0.74117647058823533,  0.85882352941176465),
(4)                 (0.40392156862745099,  0.66274509803921566,  0.81176470588235294),
(4)                 (0.21176470588235294,  0.56470588235294117,  0.75294117647058822),
(4)                 (0.00784313725490196,  0.50588235294117645,  0.54117647058823526),
(4)                 (0.00392156862745098,  0.42352941176470588,  0.34901960784313724),
(4)                 (0.00392156862745098,  0.27450980392156865,  0.21176470588235294)
(4)                 )
(0)             _PuOr_data = (
(4)                 (0.49803921568627452,  0.23137254901960785,  0.03137254901960784),
(4)                 (0.70196078431372544,  0.34509803921568627,  0.02352941176470588),
(4)                 (0.8784313725490196 ,  0.50980392156862742,  0.07843137254901961),
(4)                 (0.99215686274509807,  0.72156862745098038,  0.38823529411764707),
(4)                 (0.99607843137254903,  0.8784313725490196 ,  0.71372549019607845),
(4)                 (0.96862745098039216,  0.96862745098039216,  0.96862745098039216),
(4)                 (0.84705882352941175,  0.85490196078431369,  0.92156862745098034),
(4)                 (0.69803921568627447,  0.6705882352941176 ,  0.82352941176470584),
(4)                 (0.50196078431372548,  0.45098039215686275,  0.67450980392156867),
(4)                 (0.32941176470588235,  0.15294117647058825,  0.53333333333333333),
(4)                 (0.17647058823529413,  0.0                ,  0.29411764705882354)
(4)                 )
(0)             _PuRd_data = (
(4)                 (0.96862745098039216,  0.95686274509803926,  0.97647058823529409),
(4)                 (0.90588235294117647,  0.88235294117647056,  0.93725490196078431),
(4)                 (0.83137254901960789,  0.72549019607843135,  0.85490196078431369),
(4)                 (0.78823529411764703,  0.58039215686274515,  0.7803921568627451 ),
(4)                 (0.87450980392156863,  0.396078431372549  ,  0.69019607843137254),
(4)                 (0.90588235294117647,  0.16078431372549021,  0.54117647058823526),
(4)                 (0.80784313725490198,  0.07058823529411765,  0.33725490196078434),
(4)                 (0.59607843137254901,  0.0                ,  0.2627450980392157 ),
(4)                 (0.40392156862745099,  0.0                ,  0.12156862745098039)
(4)                 )
(0)             _Purples_data = (
(4)                 (0.9882352941176471 ,  0.98431372549019602,  0.99215686274509807),
(4)                 (0.93725490196078431,  0.92941176470588238,  0.96078431372549022),
(4)                 (0.85490196078431369,  0.85490196078431369,  0.92156862745098034),
(4)                 (0.73725490196078436,  0.74117647058823533,  0.86274509803921573),
(4)                 (0.61960784313725492,  0.60392156862745094,  0.78431372549019607),
(4)                 (0.50196078431372548,  0.49019607843137253,  0.72941176470588232),
(4)                 (0.41568627450980394,  0.31764705882352939,  0.63921568627450975),
(4)                 (0.32941176470588235,  0.15294117647058825,  0.5607843137254902 ),
(4)                 (0.24705882352941178,  0.0                ,  0.49019607843137253)
(4)                 )
(0)             _RdBu_data = (
(4)                 (0.40392156862745099,  0.0                ,  0.12156862745098039),
(4)                 (0.69803921568627447,  0.09411764705882353,  0.16862745098039217),
(4)                 (0.83921568627450982,  0.37647058823529411,  0.30196078431372547),
(4)                 (0.95686274509803926,  0.6470588235294118 ,  0.50980392156862742),
```

```
(4)                 (0.99215686274509807,  0.85882352941176465,  0.7803921568627451 ),
(4)                 (0.96862745098039216,  0.96862745098039216,  0.96862745098039216),
(4)                 (0.81960784313725488,  0.89803921568627454,  0.94117647058823528),
(4)                 (0.5725490196078431 ,  0.77254901960784317,  0.87058823529411766),
(4)                 (0.2627450980392157 ,  0.57647058823529407,  0.76470588235294112),
(4)                 (0.12941176470588237,  0.4                ,  0.67450980392156867),
(4)                 (0.0196078431372549 ,  0.18823529411764706,  0.38039215686274508)
(4)                 )
(0)             _RdGy_data = (
(4)                 (0.40392156862745099,  0.0                ,  0.12156862745098039),
(4)                 (0.69803921568627447,  0.09411764705882353,  0.16862745098039217),
(4)                 (0.83921568627450982,  0.37647058823529411,  0.30196078431372547),
(4)                 (0.95686274509803926,  0.6470588235294118 ,  0.50980392156862742),
(4)                 (0.99215686274509807,  0.85882352941176465,  0.7803921568627451 ),
(4)                 (1.0                ,  1.0                ,  1.0                ),
(4)                 (0.8784313725490196 ,  0.8784313725490196 ,  0.8784313725490196 ),
(4)                 (0.72941176470588232,  0.72941176470588232,  0.72941176470588232),
(4)                 (0.52941176470588236,  0.52941176470588236,  0.52941176470588236),
(4)                 (0.30196078431372547,  0.30196078431372547,  0.30196078431372547),
(4)                 (0.10196078431372549,  0.10196078431372549,  0.10196078431372549)
(4)                 )
(0)             _RdPu_data = (
(4)                 (1.0                ,  0.96862745098039216,  0.95294117647058818),
(4)                 (0.99215686274509807,  0.8784313725490196 ,  0.86666666666666667),
(4)                 (0.9882352941176471 ,  0.77254901960784317,  0.75294117647058822),
(4)                 (0.98039215686274506,  0.62352941176470589,  0.70980392156862748),
(4)                 (0.96862745098039216,  0.40784313725490196,  0.63137254901960782),
(4)                 (0.86666666666666667,  0.20392156862745098,  0.59215686274509804),
(4)                 (0.68235294117647061,  0.00392156862745098,  0.49411764705882355),
(4)                 (0.47843137254901963,  0.00392156862745098,  0.46666666666666667),
(4)                 (0.28627450980392155,  0.0                ,  0.41568627450980394)
(4)                 )
(0)             _RdYlBu_data = (
(4)                 (0.6470588235294118 , 0.0                 , 0.14901960784313725),
(4)                 (0.84313725490196079, 0.18823529411764706 , 0.15294117647058825),
(4)                 (0.95686274509803926, 0.42745098039215684 , 0.2627450980392157 ),
(4)                 (0.99215686274509807, 0.68235294117647061 , 0.38039215686274508),
(4)                 (0.99607843137254903, 0.8784313725490196  , 0.56470588235294117),
(4)                 (1.0                , 1.0                 , 0.74901960784313726),
(4)                 (0.8784313725490196 , 0.95294117647058818 , 0.97254901960784312),
(4)                 (0.6705882352941176 , 0.85098039215686272 , 0.9137254901960784 ),
(4)                 (0.45490196078431372, 0.67843137254901964 , 0.81960784313725488),
(4)                 (0.27058823529411763, 0.45882352941176469 , 0.70588235294117652),
(4)                 (0.19215686274509805, 0.21176470588235294 , 0.58431372549019611)
(4)                 )
(0)             _RdYlGn_data = (
(4)                 (0.6470588235294118 , 0.0                 , 0.14901960784313725),
(4)                 (0.84313725490196079, 0.18823529411764706 , 0.15294117647058825),
(4)                 (0.95686274509803926, 0.42745098039215684 , 0.2627450980392157 ),
(4)                 (0.99215686274509807, 0.68235294117647061 , 0.38039215686274508),
(4)                 (0.99607843137254903, 0.8784313725490196  , 0.54509803921568623),
(4)                 (1.0                , 1.0                 , 0.74901960784313726),
(4)                 (0.85098039215686272, 0.93725490196078431 , 0.54509803921568623),
(4)                 (0.65098039215686276, 0.85098039215686272 , 0.41568627450980394),
(4)                 (0.4                , 0.74117647058823533 , 0.38823529411764707),
(4)                 (0.10196078431372549, 0.59607843137254901 , 0.31372549019607843),
(4)                 (0.0                , 0.40784313725490196 , 0.21568627450980393)
(4)                 )
(0)             _Reds_data = (
(4)                 (1.0                , 0.96078431372549022 , 0.94117647058823528),
(4)                 (0.99607843137254903, 0.8784313725490196  , 0.82352941176470584),
(4)                 (0.9882352941176471 , 0.73333333333333328 , 0.63137254901960782),
(4)                 (0.9882352941176471 , 0.5725490196078431  , 0.44705882352941179),
(4)                 (0.98431372549019602, 0.41568627450980394 , 0.29019607843137257),
(4)                 (0.93725490196078431, 0.23137254901960785 , 0.17254901960784313),
(4)                 (0.79607843137254897, 0.09411764705882352 8, 0.11372549019607843),
(4)                 (0.6470588235294118 , 0.058823529411764705, 0.08235294117647058),
(4)                 (0.40392156862745099, 0.0                 , 0.05098039215686274)
(4)                 )
```

```
(0)             _Spectral_data = (
(4)                 (0.61960784313725492, 0.00392156862745098 0, 0.25882352941176473),
(4)                 (0.83529411764705885, 0.24313725490196078 , 0.30980392156862746),
(4)                 (0.95686274509803926, 0.42745098039215684 , 0.2627450980392157 ),
(4)                 (0.99215686274509807, 0.68235294117647061 , 0.38039215686274508),
(4)                 (0.99607843137254903, 0.8784313725490196  , 0.54509803921568623),
(4)                 (1.0                , 1.0                 , 0.74901960784313726),
(4)                 (0.90196078431372551, 0.96078431372549022 , 0.59607843137254901),
(4)                 (0.6705882352941176 , 0.8666666666666667  , 0.64313725490196083),
(4)                 (0.4                , 0.76078431372549016 , 0.6470588235294118 ),
(4)                 (0.19607843137254902, 0.53333333333333333 , 0.74117647058823533),
(4)                 (0.36862745098039218, 0.30980392156862746 , 0.63529411764705879)
(4)                 )
(0)             _YlGn_data = (
(4)                 (1.0                , 1.0                 , 0.89803921568627454),
(4)                 (0.96862745098039216, 0.9882352941176471  , 0.72549019607843135),
(4)                 (0.85098039215686272, 0.94117647058823528 , 0.63921568627450975),
(4)                 (0.67843137254901964, 0.8666666666666667  , 0.55686274509803924),
(4)                 (0.47058823529411764, 0.77647058823529413 , 0.47450980392156861),
(4)                 (0.25490196078431371, 0.6705882352941176  , 0.36470588235294116),
(4)                 (0.13725490196078433, 0.51764705882352946 , 0.2627450980392157 ),
(4)                 (0.0                , 0.40784313725490196 , 0.21568627450980393),
(4)                 (0.0                , 0.27058823529411763 , 0.16078431372549021)
(4)                 )
(0)             _YlGnBu_data = (
(4)                 (1.0                , 1.0                 , 0.85098039215686272),
(4)                 (0.92941176470588238, 0.97254901960784312 , 0.69411764705882351),
(4)                 (0.7803921568627451 , 0.9137254901960784  , 0.70588235294117652),
(4)                 (0.49803921568627452, 0.80392156862745101 , 0.73333333333333328),
(4)                 (0.25490196078431371, 0.71372549019607845 , 0.7686274509803922 ),
(4)                 (0.11372549019607843, 0.56862745098039214 , 0.75294117647058822),
(4)                 (0.13333333333333333, 0.36862745098039218 , 0.6588235294117647 ),
(4)                 (0.14509803921568629, 0.20392156862745098 , 0.58039215686274515),
(4)                 (0.03137254901960784, 0.11372549019607843 , 0.34509803921568627)
(4)                 )
(0)             _YlOrBr_data = (
(4)                 (1.0                , 1.0                 , 0.89803921568627454),
(4)                 (1.0                , 0.96862745098039216 , 0.73725490196078436),
(4)                 (0.99607843137254903, 0.8901960784313725  , 0.56862745098039214),
(4)                 (0.99607843137254903, 0.7686274509803922  , 0.30980392156862746),
(4)                 (0.99607843137254903, 0.6                 , 0.16078431372549021),
(4)                 (0.92549019607843142, 0.4392156862745098  , 0.07843137254901961),
(4)                 (0.8                , 0.29803921568627451 , 0.00784313725490196),
(4)                 (0.6                , 0.20392156862745098 , 0.01568627450980392),
(4)                 (0.4                , 0.14509803921568629 , 0.02352941176470588)
(4)                 )
(0)             _YlOrRd_data = (
(4)                 (1.0                , 1.0                 , 0.8                ),
(4)                 (1.0                , 0.92941176470588238 , 0.62745098039215685),
(4)                 (0.99607843137254903, 0.85098039215686272 , 0.46274509803921571),
(4)                 (0.99607843137254903, 0.69803921568627447 , 0.29803921568627451),
(4)                 (0.99215686274509807, 0.55294117647058827 , 0.23529411764705882),
(4)                 (0.9882352941176471 , 0.30588235294117649 , 0.16470588235294117),
(4)                 (0.8901960784313725 , 0.10196078431372549 , 0.10980392156862745),
(4)                 (0.74117647058823533, 0.0                 , 0.14901960784313725),
(4)                 (0.50196078431372548, 0.0                 , 0.14901960784313725)
(4)                 )
(0)             _Accent_data = (
(4)                 (0.49803921568627452, 0.78823529411764703, 0.49803921568627452),
(4)                 (0.74509803921568629, 0.68235294117647061, 0.83137254901960789),
(4)                 (0.99215686274509807, 0.75294117647058822, 0.52549019607843139),
(4)                 (1.0,                 1.0,                 0.6                ),
(4)                 (0.2196078431372549,  0.42352941176470588, 0.69019607843137254),
(4)                 (0.94117647058823528, 0.00784313725490196, 0.49803921568627452),
(4)                 (0.74901960784313726, 0.35686274509803922, 0.09019607843137254),
(4)                 (0.4,                 0.4,                 0.4                ),
(4)                 )
(0)             _Dark2_data = (
(4)                 (0.10588235294117647, 0.61960784313725492, 0.46666666666666667),
```

```
(4)                 (0.85098039215686272, 0.37254901960784315, 0.00784313725490196),
(4)                 (0.45882352941176469, 0.4392156862745098,  0.70196078431372544),
(4)                 (0.90588235294117647, 0.16078431372549021, 0.54117647058823526),
(4)                 (0.4,                 0.65098039215686276, 0.11764705882352941),
(4)                 (0.90196078431372551, 0.6705882352941176,  0.00784313725490196),
(4)                 (0.65098039215686276, 0.46274509803921571, 0.11372549019607843),
(4)                 (0.4,                 0.4,                 0.4                ),
(4)                 )
(0)             _Paired_data = (
(4)                 (0.65098039215686276, 0.80784313725490198, 0.8901960784313725 ),
(4)                 (0.12156862745098039, 0.47058823529411764, 0.70588235294117652),
(4)                 (0.69803921568627447, 0.87450980392156863, 0.54117647058823526),
(4)                 (0.2,                 0.62745098039215685, 0.17254901960784313),
(4)                 (0.98431372549019602, 0.60392156862745094, 0.6                ),
(4)                 (0.8901960784313725,  0.10196078431372549, 0.10980392156862745),
(4)                 (0.99215686274509807, 0.74901960784313726, 0.43529411764705883),
(4)                 (1.0,                 0.49803921568627452, 0.0                ),
(4)                 (0.792156862745098,   0.69803921568627447, 0.83921568627450982),
(4)                 (0.41568627450980394, 0.23921568627450981, 0.60392156862745094),
(4)                 (1.0,                 1.0,                 0.6                ),
(4)                 (0.69411764705882351, 0.34901960784313724, 0.15686274509803921),
(4)                 )
(0)             _Pastel1_data = (
(4)                 (0.98431372549019602, 0.70588235294117652, 0.68235294117647061),
(4)                 (0.70196078431372544, 0.80392156862745101, 0.8901960784313725 ),
(4)                 (0.8,                 0.92156862745098034, 0.77254901960784317),
(4)                 (0.87058823529411766, 0.79607843137254897, 0.89411764705882357),
(4)                 (0.99607843137254903, 0.85098039215686272, 0.65098039215686276),
(4)                 (1.0,                 1.0,                 0.8                ),
(4)                 (0.89803921568627454, 0.84705882352941175, 0.74117647058823533),
(4)                 (0.99215686274509807, 0.85490196078431369, 0.92549019607843142),
(4)                 (0.94901960784313721, 0.94901960784313721, 0.94901960784313721),
(4)                 )
(0)             _Pastel2_data = (
(4)                 (0.70196078431372544, 0.88627450980392153, 0.80392156862745101),
(4)                 (0.99215686274509807, 0.80392156862745101, 0.67450980392156867),
(4)                 (0.79607843137254897, 0.83529411764705885, 0.90980392156862744),
(4)                 (0.95686274509803926, 0.792156862745098,   0.89411764705882357),
(4)                 (0.90196078431372551, 0.96078431372549022, 0.78823529411764703),
(4)                 (1.0,                 0.94901960784313721, 0.68235294117647061),
(4)                 (0.94509803921568625, 0.88627450980392153, 0.8                ),
(4)                 (0.8,                 0.8,                 0.8                ),
(4)                 )
(0)             _Set1_data = (
(4)                 (0.89411764705882357, 0.10196078431372549, 0.10980392156862745),
(4)                 (0.21568627450980393, 0.49411764705882355, 0.72156862745098038),
(4)                 (0.30196078431372547, 0.68627450980392157, 0.29019607843137257),
(4)                 (0.59607843137254901, 0.30588235294117649, 0.63921568627450975),
(4)                 (1.0,                 0.49803921568627452, 0.0                ),
(4)                 (1.0,                 1.0,                 0.2                ),
(4)                 (0.65098039215686276, 0.33725490196078434, 0.15686274509803921),
(4)                 (0.96862745098039216, 0.50588235294117645, 0.74901960784313726),
(4)                 (0.6,                 0.6,                 0.6),
(4)                 )
(0)             _Set2_data = (
(4)                 (0.4,                 0.76078431372549016, 0.6470588235294118 ),
(4)                 (0.9882352941176471,  0.55294117647058827, 0.3843137254901961 ),
(4)                 (0.55294117647058827, 0.62745098039215685, 0.79607843137254897),
(4)                 (0.90588235294117647, 0.54117647058823526, 0.76470588235294112),
(4)                 (0.65098039215686276, 0.84705882352941175, 0.32941176470588235),
(4)                 (1.0,                 0.85098039215686272, 0.18431372549019609),
(4)                 (0.89803921568627454, 0.7686274509803922,  0.58039215686274515),
(4)                 (0.70196078431372544, 0.70196078431372544, 0.70196078431372544),
(4)                 )
(0)             _Set3_data = (
(4)                 (0.55294117647058827, 0.82745098039215681, 0.7803921568627451 ),
(4)                 (1.0,                 1.0,                 0.70196078431372544),
(4)                 (0.74509803921568629, 0.72941176470588232, 0.85490196078431369),
(4)                 (0.98431372549019602, 0.50196078431372548, 0.44705882352941179),
```

```
857: (4)                (0.50196078431372548, 0.69411764705882351, 0.82745098039215681),
858: (4)                (0.99215686274509807, 0.70588235294117652, 0.3843137254901961 ),
859: (4)                (0.70196078431372544, 0.87058823529411766, 0.41176470588235292),
860: (4)                (0.9882352941176471,  0.80392156862745101, 0.89803921568627454),
861: (4)                (0.85098039215686272, 0.85098039215686272, 0.85098039215686272),
862: (4)                (0.73725490196078436, 0.50196078431372548, 0.74117647058823533),
863: (4)                (0.8,                 0.92156862745098034, 0.77254901960784317),
864: (4)                (1.0,                 0.92941176470588238, 0.43529411764705883),
865: (4)                )
866: (0)            _gist_earth_data = {
867: (4)                'red': (
868: (8)                    (0.0, 0.0, 0.0000),
869: (8)                    (0.2824, 0.1882, 0.1882),
870: (8)                    (0.4588, 0.2714, 0.2714),
871: (8)                    (0.5490, 0.4719, 0.4719),
872: (8)                    (0.6980, 0.7176, 0.7176),
873: (8)                    (0.7882, 0.7553, 0.7553),
874: (8)                    (1.0000, 0.9922, 0.9922),
875: (4)                ),
876: (4)                'green': (
877: (8)                    (0.0, 0.0, 0.0000),
878: (8)                    (0.0275, 0.0000, 0.0000),
879: (8)                    (0.1098, 0.1893, 0.1893),
880: (8)                    (0.1647, 0.3035, 0.3035),
881: (8)                    (0.2078, 0.3841, 0.3841),
882: (8)                    (0.2824, 0.5020, 0.5020),
883: (8)                    (0.5216, 0.6397, 0.6397),
884: (8)                    (0.6980, 0.7171, 0.7171),
885: (8)                    (0.7882, 0.6392, 0.6392),
886: (8)                    (0.7922, 0.6413, 0.6413),
887: (8)                    (0.8000, 0.6447, 0.6447),
888: (8)                    (0.8078, 0.6481, 0.6481),
889: (8)                    (0.8157, 0.6549, 0.6549),
890: (8)                    (0.8667, 0.6991, 0.6991),
891: (8)                    (0.8745, 0.7103, 0.7103),
892: (8)                    (0.8824, 0.7216, 0.7216),
893: (8)                    (0.8902, 0.7323, 0.7323),
894: (8)                    (0.8980, 0.7430, 0.7430),
895: (8)                    (0.9412, 0.8275, 0.8275),
896: (8)                    (0.9569, 0.8635, 0.8635),
897: (8)                    (0.9647, 0.8816, 0.8816),
898: (8)                    (0.9961, 0.9733, 0.9733),
899: (8)                    (1.0000, 0.9843, 0.9843),
900: (4)                ),
901: (4)                'blue': (
902: (8)                    (0.0, 0.0, 0.0000),
903: (8)                    (0.0039, 0.1684, 0.1684),
904: (8)                    (0.0078, 0.2212, 0.2212),
905: (8)                    (0.0275, 0.4329, 0.4329),
906: (8)                    (0.0314, 0.4549, 0.4549),
907: (8)                    (0.2824, 0.5004, 0.5004),
908: (8)                    (0.4667, 0.2748, 0.2748),
909: (8)                    (0.5451, 0.3205, 0.3205),
910: (8)                    (0.7843, 0.3961, 0.3961),
911: (8)                    (0.8941, 0.6651, 0.6651),
912: (8)                    (1.0000, 0.9843, 0.9843),
913: (4)                )
914: (0)            }
915: (0)            _gist_gray_data = {
916: (8)                    'red': gfunc[3],
917: (8)                    'green': gfunc[3],
918: (8)                    'blue': gfunc[3],
919: (0)            }
920: (0)            def _gist_heat_red(x): return 1.5 * x
921: (0)            def _gist_heat_green(x): return 2 * x - 1
922: (0)            def _gist_heat_blue(x): return 4 * x - 3
923: (0)            _gist_heat_data = {
924: (4)                'red': _gist_heat_red, 'green': _gist_heat_green, 'blue': _gist_heat_blue}
925: (0)            _gist_ncar_data = {
```

```
(4)                 'red': (
(8)                     (0.0, 0.0, 0.0000),
(8)                     (0.3098, 0.0000, 0.0000),
(8)                     (0.3725, 0.3993, 0.3993),
(8)                     (0.4235, 0.5003, 0.5003),
(8)                     (0.5333, 1.0000, 1.0000),
(8)                     (0.7922, 1.0000, 1.0000),
(8)                     (0.8471, 0.6218, 0.6218),
(8)                     (0.8980, 0.9235, 0.9235),
(8)                     (1.0000, 0.9961, 0.9961),
(4)                 ),
(4)                 'green': (
(8)                     (0.0, 0.0, 0.0000),
(8)                     (0.0510, 0.3722, 0.3722),
(8)                     (0.1059, 0.0000, 0.0000),
(8)                     (0.1569, 0.7202, 0.7202),
(8)                     (0.1608, 0.7537, 0.7537),
(8)                     (0.1647, 0.7752, 0.7752),
(8)                     (0.2157, 1.0000, 1.0000),
(8)                     (0.2588, 0.9804, 0.9804),
(8)                     (0.2706, 0.9804, 0.9804),
(8)                     (0.3176, 1.0000, 1.0000),
(8)                     (0.3686, 0.8081, 0.8081),
(8)                     (0.4275, 1.0000, 1.0000),
(8)                     (0.5216, 1.0000, 1.0000),
(8)                     (0.6314, 0.7292, 0.7292),
(8)                     (0.6863, 0.2796, 0.2796),
(8)                     (0.7451, 0.0000, 0.0000),
(8)                     (0.7922, 0.0000, 0.0000),
(8)                     (0.8431, 0.1753, 0.1753),
(8)                     (0.8980, 0.5000, 0.5000),
(8)                     (1.0000, 0.9725, 0.9725),
(4)                 ),
(4)                 'blue': (
(8)                     (0.0, 0.5020, 0.5020),
(8)                     (0.0510, 0.0222, 0.0222),
(8)                     (0.1098, 1.0000, 1.0000),
(8)                     (0.2039, 1.0000, 1.0000),
(8)                     (0.2627, 0.6145, 0.6145),
(8)                     (0.3216, 0.0000, 0.0000),
(8)                     (0.4157, 0.0000, 0.0000),
(8)                     (0.4745, 0.2342, 0.2342),
(8)                     (0.5333, 0.0000, 0.0000),
(8)                     (0.5804, 0.0000, 0.0000),
(8)                     (0.6314, 0.0549, 0.0549),
(8)                     (0.6902, 0.0000, 0.0000),
(8)                     (0.7373, 0.0000, 0.0000),
(8)                     (0.7922, 0.9738, 0.9738),
(8)                     (0.8000, 1.0000, 1.0000),
(8)                     (0.8431, 1.0000, 1.0000),
(8)                     (0.8980, 0.9341, 0.9341),
(8)                     (1.0000, 0.9961, 0.9961),
(4)                 )
(0)             }
(0)             _gist_rainbow_data = (
(8)                     (0.000, (1.00, 0.00, 0.16)),
(8)                     (0.030, (1.00, 0.00, 0.00)),
(8)                     (0.215, (1.00, 1.00, 0.00)),
(8)                     (0.400, (0.00, 1.00, 0.00)),
(8)                     (0.586, (0.00, 1.00, 1.00)),
(8)                     (0.770, (0.00, 0.00, 1.00)),
(8)                     (0.954, (1.00, 0.00, 1.00)),
(8)                     (1.000, (1.00, 0.00, 0.75))
(0)             )
(0)             _gist_stern_data = {
(8)                     'red': (
(12)                        (0.000, 0.000, 0.000), (0.0547, 1.000, 1.000),
(12)                        (0.250, 0.027, 0.250),  # (0.2500, 0.250, 0.250),
(12)                        (1.000, 1.000, 1.000)),
```

```
995: (8)                    'green': ((0, 0, 0), (1, 1, 1)),
996: (8)                    'blue': (
997: (12)                       (0.000, 0.000, 0.000), (0.500, 1.000, 1.000),
998: (12)                       (0.735, 0.000, 0.000), (1.000, 1.000, 1.000))
999: (0)            }
1000: (0)           def _gist_yarg(x): return 1 - x
1001: (0)           _gist_yarg_data = {'red': _gist_yarg, 'green': _gist_yarg, 'blue': _gist_yarg}
1002: (0)           _coolwarm_data = {
1003: (4)               'red': [
1004: (8)                   (0.0, 0.2298057, 0.2298057),
1005: (8)                   (0.03125, 0.26623388, 0.26623388),
1006: (8)                   (0.0625, 0.30386891, 0.30386891),
1007: (8)                   (0.09375, 0.342804478, 0.342804478),
1008: (8)                   (0.125, 0.38301334, 0.38301334),
1009: (8)                   (0.15625, 0.424369608, 0.424369608),
1010: (8)                   (0.1875, 0.46666708, 0.46666708),
1011: (8)                   (0.21875, 0.509635204, 0.509635204),
1012: (8)                   (0.25, 0.552953156, 0.552953156),
1013: (8)                   (0.28125, 0.596262162, 0.596262162),
1014: (8)                   (0.3125, 0.639176211, 0.639176211),
1015: (8)                   (0.34375, 0.681291281, 0.681291281),
1016: (8)                   (0.375, 0.722193294, 0.722193294),
1017: (8)                   (0.40625, 0.761464949, 0.761464949),
1018: (8)                   (0.4375, 0.798691636, 0.798691636),
1019: (8)                   (0.46875, 0.833466556, 0.833466556),
1020: (8)                   (0.5, 0.865395197, 0.865395197),
1021: (8)                   (0.53125, 0.897787179, 0.897787179),
1022: (8)                   (0.5625, 0.924127593, 0.924127593),
1023: (8)                   (0.59375, 0.944468518, 0.944468518),
1024: (8)                   (0.625, 0.958852946, 0.958852946),
1025: (8)                   (0.65625, 0.96732803, 0.96732803),
1026: (8)                   (0.6875, 0.969954137, 0.969954137),
1027: (8)                   (0.71875, 0.966811177, 0.966811177),
1028: (8)                   (0.75, 0.958003065, 0.958003065),
1029: (8)                   (0.78125, 0.943660866, 0.943660866),
1030: (8)                   (0.8125, 0.923944917, 0.923944917),
1031: (8)                   (0.84375, 0.89904617, 0.89904617),
1032: (8)                   (0.875, 0.869186849, 0.869186849),
1033: (8)                   (0.90625, 0.834620542, 0.834620542),
1034: (8)                   (0.9375, 0.795631745, 0.795631745),
1035: (8)                   (0.96875, 0.752534934, 0.752534934),
1036: (8)                   (1.0, 0.705673158, 0.705673158)],
1037: (4)               'green': [
1038: (8)                   (0.0, 0.298717966, 0.298717966),
1039: (8)                   (0.03125, 0.353094838, 0.353094838),
1040: (8)                   (0.0625, 0.406535296, 0.406535296),
1041: (8)                   (0.09375, 0.458757618, 0.458757618),
1042: (8)                   (0.125, 0.50941904, 0.50941904),
1043: (8)                   (0.15625, 0.558148092, 0.558148092),
1044: (8)                   (0.1875, 0.604562568, 0.604562568),
1045: (8)                   (0.21875, 0.648280772, 0.648280772),
1046: (8)                   (0.25, 0.688929332, 0.688929332),
1047: (8)                   (0.28125, 0.726149107, 0.726149107),
1048: (8)                   (0.3125, 0.759599947, 0.759599947),
1049: (8)                   (0.34375, 0.788964712, 0.788964712),
1050: (8)                   (0.375, 0.813952739, 0.813952739),
1051: (8)                   (0.40625, 0.834302879, 0.834302879),
1052: (8)                   (0.4375, 0.849786142, 0.849786142),
1053: (8)                   (0.46875, 0.860207984, 0.860207984),
1054: (8)                   (0.5, 0.86541021, 0.86541021),
1055: (8)                   (0.53125, 0.848937047, 0.848937047),
1056: (8)                   (0.5625, 0.827384882, 0.827384882),
1057: (8)                   (0.59375, 0.800927443, 0.800927443),
1058: (8)                   (0.625, 0.769767752, 0.769767752),
1059: (8)                   (0.65625, 0.734132809, 0.734132809),
1060: (8)                   (0.6875, 0.694266682, 0.694266682),
1061: (8)                   (0.71875, 0.650421156, 0.650421156),
1062: (8)                   (0.75, 0.602842431, 0.602842431),
1063: (8)                   (0.78125, 0.551750968, 0.551750968),
```

```
(8)                     (0.8125, 0.49730856, 0.49730856),
(8)                     (0.84375, 0.439559467, 0.439559467),
(8)                     (0.875, 0.378313092, 0.378313092),
(8)                     (0.90625, 0.312874446, 0.312874446),
(8)                     (0.9375, 0.24128379, 0.24128379),
(8)                     (0.96875, 0.157246067, 0.157246067),
(8)                     (1.0, 0.01555616, 0.01555616)],
(4)                 'blue': [
(8)                     (0.0, 0.753683153, 0.753683153),
(8)                     (0.03125, 0.801466763, 0.801466763),
(8)                     (0.0625, 0.84495867, 0.84495867),
(8)                     (0.09375, 0.883725899, 0.883725899),
(8)                     (0.125, 0.917387822, 0.917387822),
(8)                     (0.15625, 0.945619588, 0.945619588),
(8)                     (0.1875, 0.968154911, 0.968154911),
(8)                     (0.21875, 0.98478814, 0.98478814),
(8)                     (0.25, 0.995375608, 0.995375608),
(8)                     (0.28125, 0.999836203, 0.999836203),
(8)                     (0.3125, 0.998151185, 0.998151185),
(8)                     (0.34375, 0.990363227, 0.990363227),
(8)                     (0.375, 0.976574709, 0.976574709),
(8)                     (0.40625, 0.956945269, 0.956945269),
(8)                     (0.4375, 0.931688648, 0.931688648),
(8)                     (0.46875, 0.901068838, 0.901068838),
(8)                     (0.5, 0.865395561, 0.865395561),
(8)                     (0.53125, 0.820880546, 0.820880546),
(8)                     (0.5625, 0.774508472, 0.774508472),
(8)                     (0.59375, 0.726736146, 0.726736146),
(8)                     (0.625, 0.678007945, 0.678007945),
(8)                     (0.65625, 0.628751763, 0.628751763),
(8)                     (0.6875, 0.579375448, 0.579375448),
(8)                     (0.71875, 0.530263762, 0.530263762),
(8)                     (0.75, 0.481775914, 0.481775914),
(8)                     (0.78125, 0.434243684, 0.434243684),
(8)                     (0.8125, 0.387970225, 0.387970225),
(8)                     (0.84375, 0.343229596, 0.343229596),
(8)                     (0.875, 0.300267182, 0.300267182),
(8)                     (0.90625, 0.259301199, 0.259301199),
(8)                     (0.9375, 0.220525627, 0.220525627),
(8)                     (0.96875, 0.184115123, 0.184115123),
(8)                     (1.0, 0.150232812, 0.150232812)]
(4)                 }
(0)             _CMRmap_data = {'red':    ((0.000, 0.00, 0.00),
(27)                                       (0.125, 0.15, 0.15),
(27)                                       (0.250, 0.30, 0.30),
(27)                                       (0.375, 0.60, 0.60),
(27)                                       (0.500, 1.00, 1.00),
(27)                                       (0.625, 0.90, 0.90),
(27)                                       (0.750, 0.90, 0.90),
(27)                                       (0.875, 0.90, 0.90),
(27)                                       (1.000, 1.00, 1.00)),
(16)                            'green':  ((0.000, 0.00, 0.00),
(27)                                       (0.125, 0.15, 0.15),
(27)                                       (0.250, 0.15, 0.15),
(27)                                       (0.375, 0.20, 0.20),
(27)                                       (0.500, 0.25, 0.25),
(27)                                       (0.625, 0.50, 0.50),
(27)                                       (0.750, 0.75, 0.75),
(27)                                       (0.875, 0.90, 0.90),
(27)                                       (1.000, 1.00, 1.00)),
(16)                            'blue':   ((0.000, 0.00, 0.00),
(27)                                       (0.125, 0.50, 0.50),
(27)                                       (0.250, 0.75, 0.75),
(27)                                       (0.375, 0.50, 0.50),
(27)                                       (0.500, 0.15, 0.15),
(27)                                       (0.625, 0.00, 0.00),
(27)                                       (0.750, 0.10, 0.10),
(27)                                       (0.875, 0.50, 0.50),
(27)                                       (1.000, 1.00, 1.00))}
```

```
(0)          _wistia_data = {
(4)              'red': [(0.0, 0.8941176470588236, 0.8941176470588236),
(12)                     (0.25, 1.0, 1.0),
(12)                     (0.5, 1.0, 1.0),
(12)                     (0.75, 1.0, 1.0),
(12)                     (1.0, 0.9882352941176471, 0.9882352941176471)],
(4)              'green': [(0.0, 1.0, 1.0),
(14)                       (0.25, 0.9098039215686274, 0.9098039215686274),
(14)                       (0.5, 0.7411764705882353, 0.7411764705882353),
(14)                       (0.75, 0.6274509803921569, 0.6274509803921569),
(14)                       (1.0, 0.4980392156862745, 0.4980392156862745)],
(4)              'blue': [(0.0, 0.47843137254901963, 0.47843137254901963),
(13)                      (0.25, 0.10196078431372549, 0.10196078431372549),
(13)                      (0.5, 0.0, 0.0),
(13)                      (0.75, 0.0, 0.0),
(13)                      (1.0, 0.0, 0.0)],
(0)          }
(0)          _tab10_data = (
(4)              (0.12156862745098039, 0.4666666666666667,  0.7058823529411765  ),  #
1f77b4
(4)              (1.0,                 0.4980392156862745,  0.054901960784313725),  #
ff7f0e
(4)              (0.17254901960784313, 0.6274509803921569,  0.17254901960784313 ),  #
2ca02c
(4)              (0.8392156862745098,  0.15294117647058825, 0.1568627450980392  ),  #
d62728
(4)              (0.5803921568627451,  0.403921568627451,   0.7411764705882353  ),  #
9467bd
(4)              (0.5490196078431373,  0.33725490196078434, 0.29411764705882354 ),  #
8c564b
(4)              (0.8901960784313725,  0.4666666666666667,  0.7607843137254902  ),  #
e377c2
(4)              (0.4980392156862745,  0.4980392156862745,  0.4980392156862745  ),  #
7f7f7f
(4)              (0.7372549019607844,  0.7411764705882353,  0.13333333333333333 ),  #
bcbd22
(4)              (0.09019607843137255, 0.7450980392156863,  0.8117647058823529),    #
17becf
(0)          )
(0)          _tab20_data = (
(4)              (0.12156862745098039, 0.4666666666666667,  0.7058823529411765  ),  #
1f77b4
(4)              (0.6823529411764706,  0.7803921568627451,  0.9098039215686274  ),  #
aec7e8
(4)              (1.0,                 0.4980392156862745,  0.054901960784313725),  #
ff7f0e
(4)              (1.0,                 0.7333333333333333,  0.47058823529411764 ),  #
ffbb78
(4)              (0.17254901960784313, 0.6274509803921569,  0.17254901960784313 ),  #
2ca02c
(4)              (0.596078431372549,   0.8745098039215686,  0.5411764705882353  ),  #
98df8a
(4)              (0.8392156862745098,  0.15294117647058825, 0.1568627450980392  ),  #
d62728
(4)              (1.0,                 0.596078431372549,   0.5882352941176471  ),  #
ff9896
(4)              (0.5803921568627451,  0.403921568627451,   0.7411764705882353  ),  #
9467bd
(4)              (0.7725490196078432,  0.6901960784313725,  0.8352941176470589  ),  #
c5b0d5
(4)              (0.5490196078431373,  0.33725490196078434, 0.29411764705882354 ),  #
8c564b
(4)              (0.7686274509803922,  0.611764705882353,   0.5803921568627451  ),  #
c49c94
(4)              (0.8901960784313725,  0.4666666666666667,  0.7607843137254902  ),  #
e377c2
(4)              (0.9686274509803922,  0.7137254901960784,  0.8235294117647058  ),  #
f7b6d2
(4)              (0.4980392156862745,  0.4980392156862745,  0.4980392156862745  ),  #
```

```
7f7f7f
1178: (4)               (0.7803921568627451,  0.7803921568627451,  0.7803921568627451  ),  #
c7c7c7
1179: (4)               (0.7372549019607844,  0.7411764705882353,  0.13333333333333333 ),  #
bcbd22
1180: (4)               (0.8588235294117647,  0.8588235294117647,  0.5529411764705883  ),  #
dbdb8d
1181: (4)               (0.09019607843137255, 0.7450980392156863,  0.8117647058823529  ),  #
17becf
1182: (4)               (0.6196078431372549,  0.8549019607843137,  0.8980392156862745),    #
9edae5
1183: (0)           )
1184: (0)           _tab20b_data = (
1185: (4)               (0.2235294117647059,  0.23137254901960785, 0.4745098039215686 ),  # 393b79
1186: (4)               (0.3215686274509804,  0.32941176470588235, 0.6392156862745098 ),  # 5254a3
1187: (4)               (0.4196078431372549,  0.43137254901960786, 0.8117647058823529 ),  # 6b6ecf
1188: (4)               (0.611764705882353,   0.6196078431372549,  0.8705882352941177 ),  # 9c9ede
1189: (4)               (0.38823529411764707, 0.4745098039215686,  0.2235294117647059 ),  # 637939
1190: (4)               (0.5490196078431373,  0.6352941176470588,  0.3215686274509804 ),  # 8ca252
1191: (4)               (0.7098039215686275,  0.8117647058823529,  0.4196078431372549 ),  # b5cf6b
1192: (4)               (0.807843137254902,   0.8588235294117647,  0.611764705882353  ),  # cedb9c
1193: (4)               (0.5490196078431373,  0.42745098039215684, 0.19215686274509805),  # 8c6d31
1194: (4)               (0.7411764705882353,  0.6196078431372549,  0.2235294117647059 ),  # bd9e39
1195: (4)               (0.9058823529411765,  0.7294117647058823,  0.3215686274509804 ),  # e7ba52
1196: (4)               (0.9058823529411765,  0.796078431372549,   0.5803921568627451 ),  # e7cb94
1197: (4)               (0.5176470588235295,  0.23529411764705882, 0.2235294117647059 ),  # 843c39
1198: (4)               (0.6784313725490196,  0.28627450980392155, 0.2901960784313726 ),  # ad494a
1199: (4)               (0.8392156862745098,  0.3803921568627451,  0.4196078431372549 ),  # d6616b
1200: (4)               (0.9058823529411765,  0.5882352941176471,  0.611764705882353  ),  # e7969c
1201: (4)               (0.4823529411764706,  0.2549019607843137,  0.45098039215686275),  # 7b4173
1202: (4)               (0.6470588235294118,  0.3176470588235294,  0.5803921568627451 ),  # a55194
1203: (4)               (0.807843137254902,   0.42745098039215684, 0.7411764705882353 ),  # ce6dbd
1204: (4)               (0.8705882352941177,  0.6196078431372549,  0.8392156862745098 ),  # de9ed6
1205: (0)           )
1206: (0)           _tab20c_data = (
1207: (4)               (0.19215686274509805, 0.5098039215686274,  0.7411764705882353  ),  #
3182bd
1208: (4)               (0.4196078431372549,  0.6823529411764706,  0.8392156862745098  ),  #
6baed6
1209: (4)               (0.6196078431372549,  0.792156862745098,   0.8823529411764706  ),  #
9ecae1
1210: (4)               (0.7764705882352941,  0.8588235294117647,  0.9372549019607843  ),  #
c6dbef
1211: (4)               (0.9019607843137255,  0.3333333333333333,  0.050980392156862744),  #
e6550d
1212: (4)               (0.9921568627450981,  0.5529411764705883,  0.23529411764705882 ),  #
fd8d3c
1213: (4)               (0.9921568627450981,  0.6823529411764706,  0.4196078431372549  ),  #
fdae6b
1214: (4)               (0.9921568627450981,  0.8156862745098039,  0.6352941176470588  ),  #
fdd0a2
1215: (4)               (0.19215686274509805, 0.6392156862745098,  0.32941176470588235 ),  #
31a354
1216: (4)               (0.4549019607843137,  0.7686274509803922,  0.4627450980392157  ),  #
74c476
1217: (4)               (0.6313725490196078,  0.8509803921568627,  0.6078431372549019  ),  #
a1d99b
1218: (4)               (0.7803921568627451,  0.9137254901960784,  0.7529411764705882  ),  #
c7e9c0
1219: (4)               (0.4588235294117647,  0.4196078431372549,  0.6941176470588235  ),  #
756bb1
1220: (4)               (0.6196078431372549,  0.6039215686274509,  0.7843137254901961  ),  #
9e9ac8
1221: (4)               (0.7372549019607844,  0.7411764705882353,  0.8627450980392157  ),  #
bcbddc
1222: (4)               (0.8549019607843137,  0.8549019607843137,  0.9215686274509803  ),  #
dadaeb
1223: (4)               (0.38823529411764707, 0.38823529411764707, 0.38823529411764707 ),  #
636363
```

```
(4)                (0.5882352941176471,  0.5882352941176471,  0.5882352941176471  ),  #
969696
(4)                (0.7411764705882353,  0.7411764705882353,  0.7411764705882353  ),  #
bdbdbd
(4)                (0.8509803921568627,  0.8509803921568627,  0.8509803921568627  ),  #
d9d9d9
(0)            )
(0)            datad = {
(4)                'Blues': _Blues_data,
(4)                'BrBG': _BrBG_data,
(4)                'BuGn': _BuGn_data,
(4)                'BuPu': _BuPu_data,
(4)                'CMRmap': _CMRmap_data,
(4)                'GnBu': _GnBu_data,
(4)                'Greens': _Greens_data,
(4)                'Greys': _Greys_data,
(4)                'OrRd': _OrRd_data,
(4)                'Oranges': _Oranges_data,
(4)                'PRGn': _PRGn_data,
(4)                'PiYG': _PiYG_data,
(4)                'PuBu': _PuBu_data,
(4)                'PuBuGn': _PuBuGn_data,
(4)                'PuOr': _PuOr_data,
(4)                'PuRd': _PuRd_data,
(4)                'Purples': _Purples_data,
(4)                'RdBu': _RdBu_data,
(4)                'RdGy': _RdGy_data,
(4)                'RdPu': _RdPu_data,
(4)                'RdYlBu': _RdYlBu_data,
(4)                'RdYlGn': _RdYlGn_data,
(4)                'Reds': _Reds_data,
(4)                'Spectral': _Spectral_data,
(4)                'Wistia': _wistia_data,
(4)                'YlGn': _YlGn_data,
(4)                'YlGnBu': _YlGnBu_data,
(4)                'YlOrBr': _YlOrBr_data,
(4)                'YlOrRd': _YlOrRd_data,
(4)                'afmhot': _afmhot_data,
(4)                'autumn': _autumn_data,
(4)                'binary': _binary_data,
(4)                'bone': _bone_data,
(4)                'brg': _brg_data,
(4)                'bwr': _bwr_data,
(4)                'cool': _cool_data,
(4)                'coolwarm': _coolwarm_data,
(4)                'copper': _copper_data,
(4)                'cubehelix': _cubehelix_data,
(4)                'flag': _flag_data,
(4)                'gist_earth': _gist_earth_data,
(4)                'gist_gray': _gist_gray_data,
(4)                'gist_heat': _gist_heat_data,
(4)                'gist_ncar': _gist_ncar_data,
(4)                'gist_rainbow': _gist_rainbow_data,
(4)                'gist_stern': _gist_stern_data,
(4)                'gist_yarg': _gist_yarg_data,
(4)                'gnuplot': _gnuplot_data,
(4)                'gnuplot2': _gnuplot2_data,
(4)                'gray': _gray_data,
(4)                'hot': _hot_data,
(4)                'hsv': _hsv_data,
(4)                'jet': _jet_data,
(4)                'nipy_spectral': _nipy_spectral_data,
(4)                'ocean': _ocean_data,
(4)                'pink': _pink_data,
(4)                'prism': _prism_data,
(4)                'rainbow': _rainbow_data,
(4)                'seismic': _seismic_data,
(4)                'spring': _spring_data,
(4)                'summer': _summer_data,
```

```
1290: (4)                'terrain': _terrain_data,
1291: (4)                'winter': _winter_data,
1292: (4)                'Accent': {'listed': _Accent_data},
1293: (4)                'Dark2': {'listed': _Dark2_data},
1294: (4)                'Paired': {'listed': _Paired_data},
1295: (4)                'Pastel1': {'listed': _Pastel1_data},
1296: (4)                'Pastel2': {'listed': _Pastel2_data},
1297: (4)                'Set1': {'listed': _Set1_data},
1298: (4)                'Set2': {'listed': _Set2_data},
1299: (4)                'Set3': {'listed': _Set3_data},
1300: (4)                'tab10': {'listed': _tab10_data},
1301: (4)                'tab20': {'listed': _tab20_data},
1302: (4)                'tab20b': {'listed': _tab20b_data},
1303: (4)                'tab20c': {'listed': _tab20c_data},
1304: (0)            }

----------------------------------------

File 6 - _cm_listed.py:

1: (0)               from .colors import ListedColormap
2: (0)               _magma_data = [[0.001462, 0.000466, 0.013866],
3: (15)                             [0.002258, 0.001295, 0.018331],
4: (15)                             [0.003279, 0.002305, 0.023708],
5: (15)                             [0.004512, 0.003490, 0.029965],
6: (15)                             [0.005950, 0.004843, 0.037130],
7: (15)                             [0.007588, 0.006356, 0.044973],
8: (15)                             [0.009426, 0.008022, 0.052844],
9: (15)                             [0.011465, 0.009828, 0.060750],
10: (15)                            [0.013708, 0.011771, 0.068667],
11: (15)                            [0.016156, 0.013840, 0.076603],
12: (15)                            [0.018815, 0.016026, 0.084584],
13: (15)                            [0.021692, 0.018320, 0.092610],
14: (15)                            [0.024792, 0.020715, 0.100676],
15: (15)                            [0.028123, 0.023201, 0.108787],
16: (15)                            [0.031696, 0.025765, 0.116965],
17: (15)                            [0.035520, 0.028397, 0.125209],
18: (15)                            [0.039608, 0.031090, 0.133515],
19: (15)                            [0.043830, 0.033830, 0.141886],
20: (15)                            [0.048062, 0.036607, 0.150327],
21: (15)                            [0.052320, 0.039407, 0.158841],
22: (15)                            [0.056615, 0.042160, 0.167446],
23: (15)                            [0.060949, 0.044794, 0.176129],
24: (15)                            [0.065330, 0.047318, 0.184892],
25: (15)                            [0.069764, 0.049726, 0.193735],
26: (15)                            [0.074257, 0.052017, 0.202660],
27: (15)                            [0.078815, 0.054184, 0.211667],
28: (15)                            [0.083446, 0.056225, 0.220755],
29: (15)                            [0.088155, 0.058133, 0.229922],
30: (15)                            [0.092949, 0.059904, 0.239164],
31: (15)                            [0.097833, 0.061531, 0.248477],
32: (15)                            [0.102815, 0.063010, 0.257854],
33: (15)                            [0.107899, 0.064335, 0.267289],
34: (15)                            [0.113094, 0.065492, 0.276784],
35: (15)                            [0.118405, 0.066479, 0.286321],
36: (15)                            [0.123833, 0.067295, 0.295879],
37: (15)                            [0.129380, 0.067935, 0.305443],
38: (15)                            [0.135053, 0.068391, 0.315000],
39: (15)                            [0.140858, 0.068654, 0.324538],
40: (15)                            [0.146785, 0.068738, 0.334011],
41: (15)                            [0.152839, 0.068637, 0.343404],
42: (15)                            [0.159018, 0.068354, 0.352688],
43: (15)                            [0.165308, 0.067911, 0.361816],
44: (15)                            [0.171713, 0.067305, 0.370771],
45: (15)                            [0.178212, 0.066576, 0.379497],
46: (15)                            [0.184801, 0.065732, 0.387973],
47: (15)                            [0.191460, 0.064818, 0.396152],
48: (15)                            [0.198177, 0.063862, 0.404009],
49: (15)                            [0.204935, 0.062907, 0.411514],
```

```
50: (15)                                [0.211718, 0.061992, 0.418647],
51: (15)                                [0.218512, 0.061158, 0.425392],
52: (15)                                [0.225302, 0.060445, 0.431742],
53: (15)                                [0.232077, 0.059889, 0.437695],
54: (15)                                [0.238826, 0.059517, 0.443256],
55: (15)                                [0.245543, 0.059352, 0.448436],
56: (15)                                [0.252220, 0.059415, 0.453248],
57: (15)                                [0.258857, 0.059706, 0.457710],
58: (15)                                [0.265447, 0.060237, 0.461840],
59: (15)                                [0.271994, 0.060994, 0.465660],
60: (15)                                [0.278493, 0.061978, 0.469190],
61: (15)                                [0.284951, 0.063168, 0.472451],
62: (15)                                [0.291366, 0.064553, 0.475462],
63: (15)                                [0.297740, 0.066117, 0.478243],
64: (15)                                [0.304081, 0.067835, 0.480812],
65: (15)                                [0.310382, 0.069702, 0.483186],
66: (15)                                [0.316654, 0.071690, 0.485380],
67: (15)                                [0.322899, 0.073782, 0.487408],
68: (15)                                [0.329114, 0.075972, 0.489287],
69: (15)                                [0.335308, 0.078236, 0.491024],
70: (15)                                [0.341482, 0.080564, 0.492631],
71: (15)                                [0.347636, 0.082946, 0.494121],
72: (15)                                [0.353773, 0.085373, 0.495501],
73: (15)                                [0.359898, 0.087831, 0.496778],
74: (15)                                [0.366012, 0.090314, 0.497960],
75: (15)                                [0.372116, 0.092816, 0.499053],
76: (15)                                [0.378211, 0.095332, 0.500067],
77: (15)                                [0.384299, 0.097855, 0.501002],
78: (15)                                [0.390384, 0.100379, 0.501864],
79: (15)                                [0.396467, 0.102902, 0.502658],
80: (15)                                [0.402548, 0.105420, 0.503386],
81: (15)                                [0.408629, 0.107930, 0.504052],
82: (15)                                [0.414709, 0.110431, 0.504662],
83: (15)                                [0.420791, 0.112920, 0.505215],
84: (15)                                [0.426877, 0.115395, 0.505714],
85: (15)                                [0.432967, 0.117855, 0.506160],
86: (15)                                [0.439062, 0.120298, 0.506555],
87: (15)                                [0.445163, 0.122724, 0.506901],
88: (15)                                [0.451271, 0.125132, 0.507198],
89: (15)                                [0.457386, 0.127522, 0.507448],
90: (15)                                [0.463508, 0.129893, 0.507652],
91: (15)                                [0.469640, 0.132245, 0.507809],
92: (15)                                [0.475780, 0.134577, 0.507921],
93: (15)                                [0.481929, 0.136891, 0.507989],
94: (15)                                [0.488088, 0.139186, 0.508011],
95: (15)                                [0.494258, 0.141462, 0.507988],
96: (15)                                [0.500438, 0.143719, 0.507920],
97: (15)                                [0.506629, 0.145958, 0.507806],
98: (15)                                [0.512831, 0.148179, 0.507648],
99: (15)                                [0.519045, 0.150383, 0.507443],
100: (15)                               [0.525270, 0.152569, 0.507192],
101: (15)                               [0.531507, 0.154739, 0.506895],
102: (15)                               [0.537755, 0.156894, 0.506551],
103: (15)                               [0.544015, 0.159033, 0.506159],
104: (15)                               [0.550287, 0.161158, 0.505719],
105: (15)                               [0.556571, 0.163269, 0.505230],
106: (15)                               [0.562866, 0.165368, 0.504692],
107: (15)                               [0.569172, 0.167454, 0.504105],
108: (15)                               [0.575490, 0.169530, 0.503466],
109: (15)                               [0.581819, 0.171596, 0.502777],
110: (15)                               [0.588158, 0.173652, 0.502035],
111: (15)                               [0.594508, 0.175701, 0.501241],
112: (15)                               [0.600868, 0.177743, 0.500394],
113: (15)                               [0.607238, 0.179779, 0.499492],
114: (15)                               [0.613617, 0.181811, 0.498536],
115: (15)                               [0.620005, 0.183840, 0.497524],
116: (15)                               [0.626401, 0.185867, 0.496456],
117: (15)                               [0.632805, 0.187893, 0.495332],
118: (15)                               [0.639216, 0.189921, 0.494150],
```

```
119: (15)                          [0.645633, 0.191952, 0.492910],
120: (15)                          [0.652056, 0.193986, 0.491611],
121: (15)                          [0.658483, 0.196027, 0.490253],
122: (15)                          [0.664915, 0.198075, 0.488836],
123: (15)                          [0.671349, 0.200133, 0.487358],
124: (15)                          [0.677786, 0.202203, 0.485819],
125: (15)                          [0.684224, 0.204286, 0.484219],
126: (15)                          [0.690661, 0.206384, 0.482558],
127: (15)                          [0.697098, 0.208501, 0.480835],
128: (15)                          [0.703532, 0.210638, 0.479049],
129: (15)                          [0.709962, 0.212797, 0.477201],
130: (15)                          [0.716387, 0.214982, 0.475290],
131: (15)                          [0.722805, 0.217194, 0.473316],
132: (15)                          [0.729216, 0.219437, 0.471279],
133: (15)                          [0.735616, 0.221713, 0.469180],
134: (15)                          [0.742004, 0.224025, 0.467018],
135: (15)                          [0.748378, 0.226377, 0.464794],
136: (15)                          [0.754737, 0.228772, 0.462509],
137: (15)                          [0.761077, 0.231214, 0.460162],
138: (15)                          [0.767398, 0.233705, 0.457755],
139: (15)                          [0.773695, 0.236249, 0.455289],
140: (15)                          [0.779968, 0.238851, 0.452765],
141: (15)                          [0.786212, 0.241514, 0.450184],
142: (15)                          [0.792427, 0.244242, 0.447543],
143: (15)                          [0.798608, 0.247040, 0.444848],
144: (15)                          [0.804752, 0.249911, 0.442102],
145: (15)                          [0.810855, 0.252861, 0.439305],
146: (15)                          [0.816914, 0.255895, 0.436461],
147: (15)                          [0.822926, 0.259016, 0.433573],
148: (15)                          [0.828886, 0.262229, 0.430644],
149: (15)                          [0.834791, 0.265540, 0.427671],
150: (15)                          [0.840636, 0.268953, 0.424666],
151: (15)                          [0.846416, 0.272473, 0.421631],
152: (15)                          [0.852126, 0.276106, 0.418573],
153: (15)                          [0.857763, 0.279857, 0.415496],
154: (15)                          [0.863320, 0.283729, 0.412403],
155: (15)                          [0.868793, 0.287728, 0.409303],
156: (15)                          [0.874176, 0.291859, 0.406205],
157: (15)                          [0.879464, 0.296125, 0.403118],
158: (15)                          [0.884651, 0.300530, 0.400047],
159: (15)                          [0.889731, 0.305079, 0.397002],
160: (15)                          [0.894700, 0.309773, 0.393995],
161: (15)                          [0.899552, 0.314616, 0.391037],
162: (15)                          [0.904281, 0.319610, 0.388137],
163: (15)                          [0.908884, 0.324755, 0.385308],
164: (15)                          [0.913354, 0.330052, 0.382563],
165: (15)                          [0.917689, 0.335500, 0.379915],
166: (15)                          [0.921884, 0.341098, 0.377376],
167: (15)                          [0.925937, 0.346844, 0.374959],
168: (15)                          [0.929845, 0.352734, 0.372677],
169: (15)                          [0.933606, 0.358764, 0.370541],
170: (15)                          [0.937221, 0.364929, 0.368567],
171: (15)                          [0.940687, 0.371224, 0.366762],
172: (15)                          [0.944006, 0.377643, 0.365136],
173: (15)                          [0.947180, 0.384178, 0.363701],
174: (15)                          [0.950210, 0.390820, 0.362468],
175: (15)                          [0.953099, 0.397563, 0.361438],
176: (15)                          [0.955849, 0.404400, 0.360619],
177: (15)                          [0.958464, 0.411324, 0.360014],
178: (15)                          [0.960949, 0.418323, 0.359630],
179: (15)                          [0.963310, 0.425390, 0.359469],
180: (15)                          [0.965549, 0.432519, 0.359529],
181: (15)                          [0.967671, 0.439703, 0.359810],
182: (15)                          [0.969680, 0.446936, 0.360311],
183: (15)                          [0.971582, 0.454210, 0.361030],
184: (15)                          [0.973381, 0.461520, 0.361965],
185: (15)                          [0.975082, 0.468861, 0.363111],
186: (15)                          [0.976690, 0.476226, 0.364466],
187: (15)                          [0.978210, 0.483612, 0.366025],
```

```
188: (15)                          [0.979645, 0.491014, 0.367783],
189: (15)                          [0.981000, 0.498428, 0.369734],
190: (15)                          [0.982279, 0.505851, 0.371874],
191: (15)                          [0.983485, 0.513280, 0.374198],
192: (15)                          [0.984622, 0.520713, 0.376698],
193: (15)                          [0.985693, 0.528148, 0.379371],
194: (15)                          [0.986700, 0.535582, 0.382210],
195: (15)                          [0.987646, 0.543015, 0.385210],
196: (15)                          [0.988533, 0.550446, 0.388365],
197: (15)                          [0.989363, 0.557873, 0.391671],
198: (15)                          [0.990138, 0.565296, 0.395122],
199: (15)                          [0.990871, 0.572706, 0.398714],
200: (15)                          [0.991558, 0.580107, 0.402441],
201: (15)                          [0.992196, 0.587502, 0.406299],
202: (15)                          [0.992785, 0.594891, 0.410283],
203: (15)                          [0.993326, 0.602275, 0.414390],
204: (15)                          [0.993834, 0.609644, 0.418613],
205: (15)                          [0.994309, 0.616999, 0.422950],
206: (15)                          [0.994738, 0.624350, 0.427397],
207: (15)                          [0.995122, 0.631696, 0.431951],
208: (15)                          [0.995480, 0.639027, 0.436607],
209: (15)                          [0.995810, 0.646344, 0.441361],
210: (15)                          [0.996096, 0.653659, 0.446213],
211: (15)                          [0.996341, 0.660969, 0.451160],
212: (15)                          [0.996580, 0.668256, 0.456192],
213: (15)                          [0.996775, 0.675541, 0.461314],
214: (15)                          [0.996925, 0.682828, 0.466526],
215: (15)                          [0.997077, 0.690088, 0.471811],
216: (15)                          [0.997186, 0.697349, 0.477182],
217: (15)                          [0.997254, 0.704611, 0.482635],
218: (15)                          [0.997325, 0.711848, 0.488154],
219: (15)                          [0.997351, 0.719089, 0.493755],
220: (15)                          [0.997351, 0.726324, 0.499428],
221: (15)                          [0.997341, 0.733545, 0.505167],
222: (15)                          [0.997285, 0.740772, 0.510983],
223: (15)                          [0.997228, 0.747981, 0.516859],
224: (15)                          [0.997138, 0.755190, 0.522806],
225: (15)                          [0.997019, 0.762398, 0.528821],
226: (15)                          [0.996898, 0.769591, 0.534892],
227: (15)                          [0.996727, 0.776795, 0.541039],
228: (15)                          [0.996571, 0.783977, 0.547233],
229: (15)                          [0.996369, 0.791167, 0.553499],
230: (15)                          [0.996162, 0.798348, 0.559820],
231: (15)                          [0.995932, 0.805527, 0.566202],
232: (15)                          [0.995680, 0.812706, 0.572645],
233: (15)                          [0.995424, 0.819875, 0.579140],
234: (15)                          [0.995131, 0.827052, 0.585701],
235: (15)                          [0.994851, 0.834213, 0.592307],
236: (15)                          [0.994524, 0.841387, 0.598983],
237: (15)                          [0.994222, 0.848540, 0.605696],
238: (15)                          [0.993866, 0.855711, 0.612482],
239: (15)                          [0.993545, 0.862859, 0.619299],
240: (15)                          [0.993170, 0.870024, 0.626189],
241: (15)                          [0.992831, 0.877168, 0.633109],
242: (15)                          [0.992440, 0.884330, 0.640099],
243: (15)                          [0.992089, 0.891470, 0.647116],
244: (15)                          [0.991688, 0.898627, 0.654202],
245: (15)                          [0.991332, 0.905763, 0.661309],
246: (15)                          [0.990930, 0.912915, 0.668481],
247: (15)                          [0.990570, 0.920049, 0.675675],
248: (15)                          [0.990175, 0.927196, 0.682926],
249: (15)                          [0.989815, 0.934329, 0.690198],
250: (15)                          [0.989434, 0.941470, 0.697519],
251: (15)                          [0.989077, 0.948604, 0.704863],
252: (15)                          [0.988717, 0.955742, 0.712242],
253: (15)                          [0.988367, 0.962878, 0.719649],
254: (15)                          [0.988033, 0.970012, 0.727077],
255: (15)                          [0.987691, 0.977154, 0.734536],
256: (15)                          [0.987387, 0.984288, 0.742002],
```

```
(15)                            [0.987053, 0.991438, 0.749504]]
(0)             _inferno_data = [[0.001462, 0.000466, 0.013866],
(17)                             [0.002267, 0.001270, 0.018570],
(17)                             [0.003299, 0.002249, 0.024239],
(17)                             [0.004547, 0.003392, 0.030909],
(17)                             [0.006006, 0.004692, 0.038558],
(17)                             [0.007676, 0.006136, 0.046836],
(17)                             [0.009561, 0.007713, 0.055143],
(17)                             [0.011663, 0.009417, 0.063460],
(17)                             [0.013995, 0.011225, 0.071862],
(17)                             [0.016561, 0.013136, 0.080282],
(17)                             [0.019373, 0.015133, 0.088767],
(17)                             [0.022447, 0.017199, 0.097327],
(17)                             [0.025793, 0.019331, 0.105930],
(17)                             [0.029432, 0.021503, 0.114621],
(17)                             [0.033385, 0.023702, 0.123397],
(17)                             [0.037668, 0.025921, 0.132232],
(17)                             [0.042253, 0.028139, 0.141141],
(17)                             [0.046915, 0.030324, 0.150164],
(17)                             [0.051644, 0.032474, 0.159254],
(17)                             [0.056449, 0.034569, 0.168414],
(17)                             [0.061340, 0.036590, 0.177642],
(17)                             [0.066331, 0.038504, 0.186962],
(17)                             [0.071429, 0.040294, 0.196354],
(17)                             [0.076637, 0.041905, 0.205799],
(17)                             [0.081962, 0.043328, 0.215289],
(17)                             [0.087411, 0.044556, 0.224813],
(17)                             [0.092990, 0.045583, 0.234358],
(17)                             [0.098702, 0.046402, 0.243904],
(17)                             [0.104551, 0.047008, 0.253430],
(17)                             [0.110536, 0.047399, 0.262912],
(17)                             [0.116656, 0.047574, 0.272321],
(17)                             [0.122908, 0.047536, 0.281624],
(17)                             [0.129285, 0.047293, 0.290788],
(17)                             [0.135778, 0.046856, 0.299776],
(17)                             [0.142378, 0.046242, 0.308553],
(17)                             [0.149073, 0.045468, 0.317085],
(17)                             [0.155850, 0.044559, 0.325338],
(17)                             [0.162689, 0.043554, 0.333277],
(17)                             [0.169575, 0.042489, 0.340874],
(17)                             [0.176493, 0.041402, 0.348111],
(17)                             [0.183429, 0.040329, 0.354971],
(17)                             [0.190367, 0.039309, 0.361447],
(17)                             [0.197297, 0.038400, 0.367535],
(17)                             [0.204209, 0.037632, 0.373238],
(17)                             [0.211095, 0.037030, 0.378563],
(17)                             [0.217949, 0.036615, 0.383522],
(17)                             [0.224763, 0.036405, 0.388129],
(17)                             [0.231538, 0.036405, 0.392400],
(17)                             [0.238273, 0.036621, 0.396353],
(17)                             [0.244967, 0.037055, 0.400007],
(17)                             [0.251620, 0.037705, 0.403378],
(17)                             [0.258234, 0.038571, 0.406485],
(17)                             [0.264810, 0.039647, 0.409345],
(17)                             [0.271347, 0.040922, 0.411976],
(17)                             [0.277850, 0.042353, 0.414392],
(17)                             [0.284321, 0.043933, 0.416608],
(17)                             [0.290763, 0.045644, 0.418637],
(17)                             [0.297178, 0.047470, 0.420491],
(17)                             [0.303568, 0.049396, 0.422182],
(17)                             [0.309935, 0.051407, 0.423721],
(17)                             [0.316282, 0.053490, 0.425116],
(17)                             [0.322610, 0.055634, 0.426377],
(17)                             [0.328921, 0.057827, 0.427511],
(17)                             [0.335217, 0.060060, 0.428524],
(17)                             [0.341500, 0.062325, 0.429425],
(17)                             [0.347771, 0.064616, 0.430217],
(17)                             [0.354032, 0.066925, 0.430906],
(17)                             [0.360284, 0.069247, 0.431497],
```

```
326: (17)                                [0.366529, 0.071579, 0.431994],
327: (17)                                [0.372768, 0.073915, 0.432400],
328: (17)                                [0.379001, 0.076253, 0.432719],
329: (17)                                [0.385228, 0.078591, 0.432955],
330: (17)                                [0.391453, 0.080927, 0.433109],
331: (17)                                [0.397674, 0.083257, 0.433183],
332: (17)                                [0.403894, 0.085580, 0.433179],
333: (17)                                [0.410113, 0.087896, 0.433098],
334: (17)                                [0.416331, 0.090203, 0.432943],
335: (17)                                [0.422549, 0.092501, 0.432714],
336: (17)                                [0.428768, 0.094790, 0.432412],
337: (17)                                [0.434987, 0.097069, 0.432039],
338: (17)                                [0.441207, 0.099338, 0.431594],
339: (17)                                [0.447428, 0.101597, 0.431080],
340: (17)                                [0.453651, 0.103848, 0.430498],
341: (17)                                [0.459875, 0.106089, 0.429846],
342: (17)                                [0.466100, 0.108322, 0.429125],
343: (17)                                [0.472328, 0.110547, 0.428334],
344: (17)                                [0.478558, 0.112764, 0.427475],
345: (17)                                [0.484789, 0.114974, 0.426548],
346: (17)                                [0.491022, 0.117179, 0.425552],
347: (17)                                [0.497257, 0.119379, 0.424488],
348: (17)                                [0.503493, 0.121575, 0.423356],
349: (17)                                [0.509730, 0.123769, 0.422156],
350: (17)                                [0.515967, 0.125960, 0.420887],
351: (17)                                [0.522206, 0.128150, 0.419549],
352: (17)                                [0.528444, 0.130341, 0.418142],
353: (17)                                [0.534683, 0.132534, 0.416667],
354: (17)                                [0.540920, 0.134729, 0.415123],
355: (17)                                [0.547157, 0.136929, 0.413511],
356: (17)                                [0.553392, 0.139134, 0.411829],
357: (17)                                [0.559624, 0.141346, 0.410078],
358: (17)                                [0.565854, 0.143567, 0.408258],
359: (17)                                [0.572081, 0.145797, 0.406369],
360: (17)                                [0.578304, 0.148039, 0.404411],
361: (17)                                [0.584521, 0.150294, 0.402385],
362: (17)                                [0.590734, 0.152563, 0.400290],
363: (17)                                [0.596940, 0.154848, 0.398125],
364: (17)                                [0.603139, 0.157151, 0.395891],
365: (17)                                [0.609330, 0.159474, 0.393589],
366: (17)                                [0.615513, 0.161817, 0.391219],
367: (17)                                [0.621685, 0.164184, 0.388781],
368: (17)                                [0.627847, 0.166575, 0.386276],
369: (17)                                [0.633998, 0.168992, 0.383704],
370: (17)                                [0.640135, 0.171438, 0.381065],
371: (17)                                [0.646260, 0.173914, 0.378359],
372: (17)                                [0.652369, 0.176421, 0.375586],
373: (17)                                [0.658463, 0.178962, 0.372748],
374: (17)                                [0.664540, 0.181539, 0.369846],
375: (17)                                [0.670599, 0.184153, 0.366879],
376: (17)                                [0.676638, 0.186807, 0.363849],
377: (17)                                [0.682656, 0.189501, 0.360757],
378: (17)                                [0.688653, 0.192239, 0.357603],
379: (17)                                [0.694627, 0.195021, 0.354388],
380: (17)                                [0.700576, 0.197851, 0.351113],
381: (17)                                [0.706500, 0.200728, 0.347777],
382: (17)                                [0.712396, 0.203656, 0.344383],
383: (17)                                [0.718264, 0.206636, 0.340931],
384: (17)                                [0.724103, 0.209670, 0.337424],
385: (17)                                [0.729909, 0.212759, 0.333861],
386: (17)                                [0.735683, 0.215906, 0.330245],
387: (17)                                [0.741423, 0.219112, 0.326576],
388: (17)                                [0.747127, 0.222378, 0.322856],
389: (17)                                [0.752794, 0.225706, 0.319085],
390: (17)                                [0.758422, 0.229097, 0.315266],
391: (17)                                [0.764010, 0.232554, 0.311399],
392: (17)                                [0.769556, 0.236077, 0.307485],
393: (17)                                [0.775059, 0.239667, 0.303526],
394: (17)                                [0.780517, 0.243327, 0.299523],
```

```
395: (17)                                [0.785929, 0.247056, 0.295477],
396: (17)                                [0.791293, 0.250856, 0.291390],
397: (17)                                [0.796607, 0.254728, 0.287264],
398: (17)                                [0.801871, 0.258674, 0.283099],
399: (17)                                [0.807082, 0.262692, 0.278898],
400: (17)                                [0.812239, 0.266786, 0.274661],
401: (17)                                [0.817341, 0.270954, 0.270390],
402: (17)                                [0.822386, 0.275197, 0.266085],
403: (17)                                [0.827372, 0.279517, 0.261750],
404: (17)                                [0.832299, 0.283913, 0.257383],
405: (17)                                [0.837165, 0.288385, 0.252988],
406: (17)                                [0.841969, 0.292933, 0.248564],
407: (17)                                [0.846709, 0.297559, 0.244113],
408: (17)                                [0.851384, 0.302260, 0.239636],
409: (17)                                [0.855992, 0.307038, 0.235133],
410: (17)                                [0.860533, 0.311892, 0.230606],
411: (17)                                [0.865006, 0.316822, 0.226055],
412: (17)                                [0.869409, 0.321827, 0.221482],
413: (17)                                [0.873741, 0.326906, 0.216886],
414: (17)                                [0.878001, 0.332060, 0.212268],
415: (17)                                [0.882188, 0.337287, 0.207628],
416: (17)                                [0.886302, 0.342586, 0.202968],
417: (17)                                [0.890341, 0.347957, 0.198286],
418: (17)                                [0.894305, 0.353399, 0.193584],
419: (17)                                [0.898192, 0.358911, 0.188860],
420: (17)                                [0.902003, 0.364492, 0.184116],
421: (17)                                [0.905735, 0.370140, 0.179350],
422: (17)                                [0.909390, 0.375856, 0.174563],
423: (17)                                [0.912966, 0.381636, 0.169755],
424: (17)                                [0.916462, 0.387481, 0.164924],
425: (17)                                [0.919879, 0.393389, 0.160070],
426: (17)                                [0.923215, 0.399359, 0.155193],
427: (17)                                [0.926470, 0.405389, 0.150292],
428: (17)                                [0.929644, 0.411479, 0.145367],
429: (17)                                [0.932737, 0.417627, 0.140417],
430: (17)                                [0.935747, 0.423831, 0.135440],
431: (17)                                [0.938675, 0.430091, 0.130438],
432: (17)                                [0.941521, 0.436405, 0.125409],
433: (17)                                [0.944285, 0.442772, 0.120354],
434: (17)                                [0.946965, 0.449191, 0.115272],
435: (17)                                [0.949562, 0.455660, 0.110164],
436: (17)                                [0.952075, 0.462178, 0.105031],
437: (17)                                [0.954506, 0.468744, 0.099874],
438: (17)                                [0.956852, 0.475356, 0.094695],
439: (17)                                [0.959114, 0.482014, 0.089499],
440: (17)                                [0.961293, 0.488716, 0.084289],
441: (17)                                [0.963387, 0.495462, 0.079073],
442: (17)                                [0.965397, 0.502249, 0.073859],
443: (17)                                [0.967322, 0.509078, 0.068659],
444: (17)                                [0.969163, 0.515946, 0.063488],
445: (17)                                [0.970919, 0.522853, 0.058367],
446: (17)                                [0.972590, 0.529798, 0.053324],
447: (17)                                [0.974176, 0.536780, 0.048392],
448: (17)                                [0.975677, 0.543798, 0.043618],
449: (17)                                [0.977092, 0.550850, 0.039050],
450: (17)                                [0.978422, 0.557937, 0.034931],
451: (17)                                [0.979666, 0.565057, 0.031409],
452: (17)                                [0.980824, 0.572209, 0.028508],
453: (17)                                [0.981895, 0.579392, 0.026250],
454: (17)                                [0.982881, 0.586606, 0.024661],
455: (17)                                [0.983779, 0.593849, 0.023770],
456: (17)                                [0.984591, 0.601122, 0.023606],
457: (17)                                [0.985315, 0.608422, 0.024202],
458: (17)                                [0.985952, 0.615750, 0.025592],
459: (17)                                [0.986502, 0.623105, 0.027814],
460: (17)                                [0.986964, 0.630485, 0.030908],
461: (17)                                [0.987337, 0.637890, 0.034916],
462: (17)                                [0.987622, 0.645320, 0.039886],
463: (17)                                [0.987819, 0.652773, 0.045581],
```

```
464: (17)                                [0.987926, 0.660250, 0.051750],
465: (17)                                [0.987945, 0.667748, 0.058329],
466: (17)                                [0.987874, 0.675267, 0.065257],
467: (17)                                [0.987714, 0.682807, 0.072489],
468: (17)                                [0.987464, 0.690366, 0.079990],
469: (17)                                [0.987124, 0.697944, 0.087731],
470: (17)                                [0.986694, 0.705540, 0.095694],
471: (17)                                [0.986175, 0.713153, 0.103863],
472: (17)                                [0.985566, 0.720782, 0.112229],
473: (17)                                [0.984865, 0.728427, 0.120785],
474: (17)                                [0.984075, 0.736087, 0.129527],
475: (17)                                [0.983196, 0.743758, 0.138453],
476: (17)                                [0.982228, 0.751442, 0.147565],
477: (17)                                [0.981173, 0.759135, 0.156863],
478: (17)                                [0.980032, 0.766837, 0.166353],
479: (17)                                [0.978806, 0.774545, 0.176037],
480: (17)                                [0.977497, 0.782258, 0.185923],
481: (17)                                [0.976108, 0.789974, 0.196018],
482: (17)                                [0.974638, 0.797692, 0.206332],
483: (17)                                [0.973088, 0.805409, 0.216877],
484: (17)                                [0.971468, 0.813122, 0.227658],
485: (17)                                [0.969783, 0.820825, 0.238686],
486: (17)                                [0.968041, 0.828515, 0.249972],
487: (17)                                [0.966243, 0.836191, 0.261534],
488: (17)                                [0.964394, 0.843848, 0.273391],
489: (17)                                [0.962517, 0.851476, 0.285546],
490: (17)                                [0.960626, 0.859069, 0.298010],
491: (17)                                [0.958720, 0.866624, 0.310820],
492: (17)                                [0.956834, 0.874129, 0.323974],
493: (17)                                [0.954997, 0.881569, 0.337475],
494: (17)                                [0.953215, 0.888942, 0.351369],
495: (17)                                [0.951546, 0.896226, 0.365627],
496: (17)                                [0.950018, 0.903409, 0.380271],
497: (17)                                [0.948683, 0.910473, 0.395289],
498: (17)                                [0.947594, 0.917399, 0.410665],
499: (17)                                [0.946809, 0.924168, 0.426373],
500: (17)                                [0.946392, 0.930761, 0.442367],
501: (17)                                [0.946403, 0.937159, 0.458592],
502: (17)                                [0.946903, 0.943348, 0.474970],
503: (17)                                [0.947937, 0.949318, 0.491426],
504: (17)                                [0.949545, 0.955063, 0.507860],
505: (17)                                [0.951740, 0.960587, 0.524203],
506: (17)                                [0.954529, 0.965896, 0.540361],
507: (17)                                [0.957896, 0.971003, 0.556275],
508: (17)                                [0.961812, 0.975924, 0.571925],
509: (17)                                [0.966249, 0.980678, 0.587206],
510: (17)                                [0.971162, 0.985282, 0.602154],
511: (17)                                [0.976511, 0.989753, 0.616760],
512: (17)                                [0.982257, 0.994109, 0.631017],
513: (17)                                [0.988362, 0.998364, 0.644924]]
514: (0)             _plasma_data = [[0.050383, 0.029803, 0.527975],
515: (16)                             [0.063536, 0.028426, 0.533124],
516: (16)                             [0.075353, 0.027206, 0.538007],
517: (16)                             [0.086222, 0.026125, 0.542658],
518: (16)                             [0.096379, 0.025165, 0.547103],
519: (16)                             [0.105980, 0.024309, 0.551368],
520: (16)                             [0.115124, 0.023556, 0.555468],
521: (16)                             [0.123903, 0.022878, 0.559423],
522: (16)                             [0.132381, 0.022258, 0.563250],
523: (16)                             [0.140603, 0.021687, 0.566959],
524: (16)                             [0.148607, 0.021154, 0.570562],
525: (16)                             [0.156421, 0.020651, 0.574065],
526: (16)                             [0.164070, 0.020171, 0.577478],
527: (16)                             [0.171574, 0.019706, 0.580806],
528: (16)                             [0.178950, 0.019252, 0.584054],
529: (16)                             [0.186213, 0.018803, 0.587228],
530: (16)                             [0.193374, 0.018354, 0.590330],
531: (16)                             [0.200445, 0.017902, 0.593364],
532: (16)                             [0.207435, 0.017442, 0.596333],
```

```
533: (16)                                [0.214350, 0.016973, 0.599239],
534: (16)                                [0.221197, 0.016497, 0.602083],
535: (16)                                [0.227983, 0.016007, 0.604867],
536: (16)                                [0.234715, 0.015502, 0.607592],
537: (16)                                [0.241396, 0.014979, 0.610259],
538: (16)                                [0.248032, 0.014439, 0.612868],
539: (16)                                [0.254627, 0.013882, 0.615419],
540: (16)                                [0.261183, 0.013308, 0.617911],
541: (16)                                [0.267703, 0.012716, 0.620346],
542: (16)                                [0.274191, 0.012109, 0.622722],
543: (16)                                [0.280648, 0.011488, 0.625038],
544: (16)                                [0.287076, 0.010855, 0.627295],
545: (16)                                [0.293478, 0.010213, 0.629490],
546: (16)                                [0.299855, 0.009561, 0.631624],
547: (16)                                [0.306210, 0.008902, 0.633694],
548: (16)                                [0.312543, 0.008239, 0.635700],
549: (16)                                [0.318856, 0.007576, 0.637640],
550: (16)                                [0.325150, 0.006915, 0.639512],
551: (16)                                [0.331426, 0.006261, 0.641316],
552: (16)                                [0.337683, 0.005618, 0.643049],
553: (16)                                [0.343925, 0.004991, 0.644710],
554: (16)                                [0.350150, 0.004382, 0.646298],
555: (16)                                [0.356359, 0.003798, 0.647810],
556: (16)                                [0.362553, 0.003243, 0.649245],
557: (16)                                [0.368733, 0.002724, 0.650601],
558: (16)                                [0.374897, 0.002245, 0.651876],
559: (16)                                [0.381047, 0.001814, 0.653068],
560: (16)                                [0.387183, 0.001434, 0.654177],
561: (16)                                [0.393304, 0.001114, 0.655199],
562: (16)                                [0.399411, 0.000859, 0.656133],
563: (16)                                [0.405503, 0.000678, 0.656977],
564: (16)                                [0.411580, 0.000577, 0.657730],
565: (16)                                [0.417642, 0.000564, 0.658390],
566: (16)                                [0.423689, 0.000646, 0.658956],
567: (16)                                [0.429719, 0.000831, 0.659425],
568: (16)                                [0.435734, 0.001127, 0.659797],
569: (16)                                [0.441732, 0.001540, 0.660069],
570: (16)                                [0.447714, 0.002080, 0.660240],
571: (16)                                [0.453677, 0.002755, 0.660310],
572: (16)                                [0.459623, 0.003574, 0.660277],
573: (16)                                [0.465550, 0.004545, 0.660139],
574: (16)                                [0.471457, 0.005678, 0.659897],
575: (16)                                [0.477344, 0.006980, 0.659549],
576: (16)                                [0.483210, 0.008460, 0.659095],
577: (16)                                [0.489055, 0.010127, 0.658534],
578: (16)                                [0.494877, 0.011990, 0.657865],
579: (16)                                [0.500678, 0.014055, 0.657088],
580: (16)                                [0.506454, 0.016333, 0.656202],
581: (16)                                [0.512206, 0.018833, 0.655209],
582: (16)                                [0.517933, 0.021563, 0.654109],
583: (16)                                [0.523633, 0.024532, 0.652901],
584: (16)                                [0.529306, 0.027747, 0.651586],
585: (16)                                [0.534952, 0.031217, 0.650165],
586: (16)                                [0.540570, 0.034950, 0.648640],
587: (16)                                [0.546157, 0.038954, 0.647010],
588: (16)                                [0.551715, 0.043136, 0.645277],
589: (16)                                [0.557243, 0.047331, 0.643443],
590: (16)                                [0.562738, 0.051545, 0.641509],
591: (16)                                [0.568201, 0.055778, 0.639477],
592: (16)                                [0.573632, 0.060028, 0.637349],
593: (16)                                [0.579029, 0.064296, 0.635126],
594: (16)                                [0.584391, 0.068579, 0.632812],
595: (16)                                [0.589719, 0.072878, 0.630408],
596: (16)                                [0.595011, 0.077190, 0.627917],
597: (16)                                [0.600266, 0.081516, 0.625342],
598: (16)                                [0.605485, 0.085854, 0.622686],
599: (16)                                [0.610667, 0.090204, 0.619951],
600: (16)                                [0.615812, 0.094564, 0.617140],
601: (16)                                [0.620919, 0.098934, 0.614257],
```

```
602: (16)                                [0.625987, 0.103312, 0.611305],
603: (16)                                [0.631017, 0.107699, 0.608287],
604: (16)                                [0.636008, 0.112092, 0.605205],
605: (16)                                [0.640959, 0.116492, 0.602065],
606: (16)                                [0.645872, 0.120898, 0.598867],
607: (16)                                [0.650746, 0.125309, 0.595617],
608: (16)                                [0.655580, 0.129725, 0.592317],
609: (16)                                [0.660374, 0.134144, 0.588971],
610: (16)                                [0.665129, 0.138566, 0.585582],
611: (16)                                [0.669845, 0.142992, 0.582154],
612: (16)                                [0.674522, 0.147419, 0.578688],
613: (16)                                [0.679160, 0.151848, 0.575189],
614: (16)                                [0.683758, 0.156278, 0.571660],
615: (16)                                [0.688318, 0.160709, 0.568103],
616: (16)                                [0.692840, 0.165141, 0.564522],
617: (16)                                [0.697324, 0.169573, 0.560919],
618: (16)                                [0.701769, 0.174005, 0.557296],
619: (16)                                [0.706178, 0.178437, 0.553657],
620: (16)                                [0.710549, 0.182868, 0.550004],
621: (16)                                [0.714883, 0.187299, 0.546338],
622: (16)                                [0.719181, 0.191729, 0.542663],
623: (16)                                [0.723444, 0.196158, 0.538981],
624: (16)                                [0.727670, 0.200586, 0.535293],
625: (16)                                [0.731862, 0.205013, 0.531601],
626: (16)                                [0.736019, 0.209439, 0.527908],
627: (16)                                [0.740143, 0.213864, 0.524216],
628: (16)                                [0.744232, 0.218288, 0.520524],
629: (16)                                [0.748289, 0.222711, 0.516834],
630: (16)                                [0.752312, 0.227133, 0.513149],
631: (16)                                [0.756304, 0.231555, 0.509468],
632: (16)                                [0.760264, 0.235976, 0.505794],
633: (16)                                [0.764193, 0.240396, 0.502126],
634: (16)                                [0.768090, 0.244817, 0.498465],
635: (16)                                [0.771958, 0.249237, 0.494813],
636: (16)                                [0.775796, 0.253658, 0.491171],
637: (16)                                [0.779604, 0.258078, 0.487539],
638: (16)                                [0.783383, 0.262500, 0.483918],
639: (16)                                [0.787133, 0.266922, 0.480307],
640: (16)                                [0.790855, 0.271345, 0.476706],
641: (16)                                [0.794549, 0.275770, 0.473117],
642: (16)                                [0.798216, 0.280197, 0.469538],
643: (16)                                [0.801855, 0.284626, 0.465971],
644: (16)                                [0.805467, 0.289057, 0.462415],
645: (16)                                [0.809052, 0.293491, 0.458870],
646: (16)                                [0.812612, 0.297928, 0.455338],
647: (16)                                [0.816144, 0.302368, 0.451816],
648: (16)                                [0.819651, 0.306812, 0.448306],
649: (16)                                [0.823132, 0.311261, 0.444806],
650: (16)                                [0.826588, 0.315714, 0.441316],
651: (16)                                [0.830018, 0.320172, 0.437836],
652: (16)                                [0.833422, 0.324635, 0.434366],
653: (16)                                [0.836801, 0.329105, 0.430905],
654: (16)                                [0.840155, 0.333580, 0.427455],
655: (16)                                [0.843484, 0.338062, 0.424013],
656: (16)                                [0.846788, 0.342551, 0.420579],
657: (16)                                [0.850066, 0.347048, 0.417153],
658: (16)                                [0.853319, 0.351553, 0.413734],
659: (16)                                [0.856547, 0.356066, 0.410322],
660: (16)                                [0.859750, 0.360588, 0.406917],
661: (16)                                [0.862927, 0.365119, 0.403519],
662: (16)                                [0.866078, 0.369660, 0.400126],
663: (16)                                [0.869203, 0.374212, 0.396738],
664: (16)                                [0.872303, 0.378774, 0.393355],
665: (16)                                [0.875376, 0.383347, 0.389976],
666: (16)                                [0.878423, 0.387932, 0.386600],
667: (16)                                [0.881443, 0.392529, 0.383229],
668: (16)                                [0.884436, 0.397139, 0.379860],
669: (16)                                [0.887402, 0.401762, 0.376494],
670: (16)                                [0.890340, 0.406398, 0.373130],
```

```
671: (16)                                [0.893250, 0.411048, 0.369768],
672: (16)                                [0.896131, 0.415712, 0.366407],
673: (16)                                [0.898984, 0.420392, 0.363047],
674: (16)                                [0.901807, 0.425087, 0.359688],
675: (16)                                [0.904601, 0.429797, 0.356329],
676: (16)                                [0.907365, 0.434524, 0.352970],
677: (16)                                [0.910098, 0.439268, 0.349610],
678: (16)                                [0.912800, 0.444029, 0.346251],
679: (16)                                [0.915471, 0.448807, 0.342890],
680: (16)                                [0.918109, 0.453603, 0.339529],
681: (16)                                [0.920714, 0.458417, 0.336166],
682: (16)                                [0.923287, 0.463251, 0.332801],
683: (16)                                [0.925825, 0.468103, 0.329435],
684: (16)                                [0.928329, 0.472975, 0.326067],
685: (16)                                [0.930798, 0.477867, 0.322697],
686: (16)                                [0.933232, 0.482780, 0.319325],
687: (16)                                [0.935630, 0.487712, 0.315952],
688: (16)                                [0.937990, 0.492667, 0.312575],
689: (16)                                [0.940313, 0.497642, 0.309197],
690: (16)                                [0.942598, 0.502639, 0.305816],
691: (16)                                [0.944844, 0.507658, 0.302433],
692: (16)                                [0.947051, 0.512699, 0.299049],
693: (16)                                [0.949217, 0.517763, 0.295662],
694: (16)                                [0.951344, 0.522850, 0.292275],
695: (16)                                [0.953428, 0.527960, 0.288883],
696: (16)                                [0.955470, 0.533093, 0.285490],
697: (16)                                [0.957469, 0.538250, 0.282096],
698: (16)                                [0.959424, 0.543431, 0.278701],
699: (16)                                [0.961336, 0.548636, 0.275305],
700: (16)                                [0.963203, 0.553865, 0.271909],
701: (16)                                [0.965024, 0.559118, 0.268513],
702: (16)                                [0.966798, 0.564396, 0.265118],
703: (16)                                [0.968526, 0.569700, 0.261721],
704: (16)                                [0.970205, 0.575028, 0.258325],
705: (16)                                [0.971835, 0.580382, 0.254931],
706: (16)                                [0.973416, 0.585761, 0.251540],
707: (16)                                [0.974947, 0.591165, 0.248151],
708: (16)                                [0.976428, 0.596595, 0.244767],
709: (16)                                [0.977856, 0.602051, 0.241387],
710: (16)                                [0.979233, 0.607532, 0.238013],
711: (16)                                [0.980556, 0.613039, 0.234646],
712: (16)                                [0.981826, 0.618572, 0.231287],
713: (16)                                [0.983041, 0.624131, 0.227937],
714: (16)                                [0.984199, 0.629718, 0.224595],
715: (16)                                [0.985301, 0.635330, 0.221265],
716: (16)                                [0.986345, 0.640969, 0.217948],
717: (16)                                [0.987332, 0.646633, 0.214648],
718: (16)                                [0.988260, 0.652325, 0.211364],
719: (16)                                [0.989128, 0.658043, 0.208100],
720: (16)                                [0.989935, 0.663787, 0.204859],
721: (16)                                [0.990681, 0.669558, 0.201642],
722: (16)                                [0.991365, 0.675355, 0.198453],
723: (16)                                [0.991985, 0.681179, 0.195295],
724: (16)                                [0.992541, 0.687030, 0.192170],
725: (16)                                [0.993032, 0.692907, 0.189084],
726: (16)                                [0.993456, 0.698810, 0.186041],
727: (16)                                [0.993814, 0.704741, 0.183043],
728: (16)                                [0.994103, 0.710698, 0.180097],
729: (16)                                [0.994324, 0.716681, 0.177208],
730: (16)                                [0.994474, 0.722691, 0.174381],
731: (16)                                [0.994553, 0.728728, 0.171622],
732: (16)                                [0.994561, 0.734791, 0.168938],
733: (16)                                [0.994495, 0.740880, 0.166335],
734: (16)                                [0.994355, 0.746995, 0.163821],
735: (16)                                [0.994141, 0.753137, 0.161404],
736: (16)                                [0.993851, 0.759304, 0.159092],
737: (16)                                [0.993482, 0.765499, 0.156891],
738: (16)                                [0.993033, 0.771720, 0.154808],
739: (16)                                [0.992505, 0.777967, 0.152855],
```

```
740: (16)                            [0.991897, 0.784239, 0.151042],
741: (16)                            [0.991209, 0.790537, 0.149377],
742: (16)                            [0.990439, 0.796859, 0.147870],
743: (16)                            [0.989587, 0.803205, 0.146529],
744: (16)                            [0.988648, 0.809579, 0.145357],
745: (16)                            [0.987621, 0.815978, 0.144363],
746: (16)                            [0.986509, 0.822401, 0.143557],
747: (16)                            [0.985314, 0.828846, 0.142945],
748: (16)                            [0.984031, 0.835315, 0.142528],
749: (16)                            [0.982653, 0.841812, 0.142303],
750: (16)                            [0.981190, 0.848329, 0.142279],
751: (16)                            [0.979644, 0.854866, 0.142453],
752: (16)                            [0.977995, 0.861432, 0.142808],
753: (16)                            [0.976265, 0.868016, 0.143351],
754: (16)                            [0.974443, 0.874622, 0.144061],
755: (16)                            [0.972530, 0.881250, 0.144923],
756: (16)                            [0.970533, 0.887896, 0.145919],
757: (16)                            [0.968443, 0.894564, 0.147014],
758: (16)                            [0.966271, 0.901249, 0.148180],
759: (16)                            [0.964021, 0.907950, 0.149370],
760: (16)                            [0.961681, 0.914672, 0.150520],
761: (16)                            [0.959276, 0.921407, 0.151566],
762: (16)                            [0.956808, 0.928152, 0.152409],
763: (16)                            [0.954287, 0.934908, 0.152921],
764: (16)                            [0.951726, 0.941671, 0.152925],
765: (16)                            [0.949151, 0.948435, 0.152178],
766: (16)                            [0.946602, 0.955190, 0.150328],
767: (16)                            [0.944152, 0.961916, 0.146861],
768: (16)                            [0.941896, 0.968590, 0.140956],
769: (16)                            [0.940015, 0.975158, 0.131326]]
770: (0)            _viridis_data = [[0.267004, 0.004874, 0.329415],
771: (17)                             [0.268510, 0.009605, 0.335427],
772: (17)                             [0.269944, 0.014625, 0.341379],
773: (17)                             [0.271305, 0.019942, 0.347269],
774: (17)                             [0.272594, 0.025563, 0.353093],
775: (17)                             [0.273809, 0.031497, 0.358853],
776: (17)                             [0.274952, 0.037752, 0.364543],
777: (17)                             [0.276022, 0.044167, 0.370164],
778: (17)                             [0.277018, 0.050344, 0.375715],
779: (17)                             [0.277941, 0.056324, 0.381191],
780: (17)                             [0.278791, 0.062145, 0.386592],
781: (17)                             [0.279566, 0.067836, 0.391917],
782: (17)                             [0.280267, 0.073417, 0.397163],
783: (17)                             [0.280894, 0.078907, 0.402329],
784: (17)                             [0.281446, 0.084320, 0.407414],
785: (17)                             [0.281924, 0.089666, 0.412415],
786: (17)                             [0.282327, 0.094955, 0.417331],
787: (17)                             [0.282656, 0.100196, 0.422160],
788: (17)                             [0.282910, 0.105393, 0.426902],
789: (17)                             [0.283091, 0.110553, 0.431554],
790: (17)                             [0.283197, 0.115680, 0.436115],
791: (17)                             [0.283229, 0.120777, 0.440584],
792: (17)                             [0.283187, 0.125848, 0.444960],
793: (17)                             [0.283072, 0.130895, 0.449241],
794: (17)                             [0.282884, 0.135920, 0.453427],
795: (17)                             [0.282623, 0.140926, 0.457517],
796: (17)                             [0.282290, 0.145912, 0.461510],
797: (17)                             [0.281887, 0.150881, 0.465405],
798: (17)                             [0.281412, 0.155834, 0.469201],
799: (17)                             [0.280868, 0.160771, 0.472899],
800: (17)                             [0.280255, 0.165693, 0.476498],
801: (17)                             [0.279574, 0.170599, 0.479997],
802: (17)                             [0.278826, 0.175490, 0.483397],
803: (17)                             [0.278012, 0.180367, 0.486697],
804: (17)                             [0.277134, 0.185228, 0.489898],
805: (17)                             [0.276194, 0.190074, 0.493001],
806: (17)                             [0.275191, 0.194905, 0.496005],
807: (17)                             [0.274128, 0.199721, 0.498911],
808: (17)                             [0.273006, 0.204520, 0.501721],
```

```
809: (17)                                [0.271828, 0.209303, 0.504434],
810: (17)                                [0.270595, 0.214069, 0.507052],
811: (17)                                [0.269308, 0.218818, 0.509577],
812: (17)                                [0.267968, 0.223549, 0.512008],
813: (17)                                [0.266580, 0.228262, 0.514349],
814: (17)                                [0.265145, 0.232956, 0.516599],
815: (17)                                [0.263663, 0.237631, 0.518762],
816: (17)                                [0.262138, 0.242286, 0.520837],
817: (17)                                [0.260571, 0.246922, 0.522828],
818: (17)                                [0.258965, 0.251537, 0.524736],
819: (17)                                [0.257322, 0.256130, 0.526563],
820: (17)                                [0.255645, 0.260703, 0.528312],
821: (17)                                [0.253935, 0.265254, 0.529983],
822: (17)                                [0.252194, 0.269783, 0.531579],
823: (17)                                [0.250425, 0.274290, 0.533103],
824: (17)                                [0.248629, 0.278775, 0.534556],
825: (17)                                [0.246811, 0.283237, 0.535941],
826: (17)                                [0.244972, 0.287675, 0.537260],
827: (17)                                [0.243113, 0.292092, 0.538516],
828: (17)                                [0.241237, 0.296485, 0.539709],
829: (17)                                [0.239346, 0.300855, 0.540844],
830: (17)                                [0.237441, 0.305202, 0.541921],
831: (17)                                [0.235526, 0.309527, 0.542944],
832: (17)                                [0.233603, 0.313828, 0.543914],
833: (17)                                [0.231674, 0.318106, 0.544834],
834: (17)                                [0.229739, 0.322361, 0.545706],
835: (17)                                [0.227802, 0.326594, 0.546532],
836: (17)                                [0.225863, 0.330805, 0.547314],
837: (17)                                [0.223925, 0.334994, 0.548053],
838: (17)                                [0.221989, 0.339161, 0.548752],
839: (17)                                [0.220057, 0.343307, 0.549413],
840: (17)                                [0.218130, 0.347432, 0.550038],
841: (17)                                [0.216210, 0.351535, 0.550627],
842: (17)                                [0.214298, 0.355619, 0.551184],
843: (17)                                [0.212395, 0.359683, 0.551710],
844: (17)                                [0.210503, 0.363727, 0.552206],
845: (17)                                [0.208623, 0.367752, 0.552675],
846: (17)                                [0.206756, 0.371758, 0.553117],
847: (17)                                [0.204903, 0.375746, 0.553533],
848: (17)                                [0.203063, 0.379716, 0.553925],
849: (17)                                [0.201239, 0.383670, 0.554294],
850: (17)                                [0.199430, 0.387607, 0.554642],
851: (17)                                [0.197636, 0.391528, 0.554969],
852: (17)                                [0.195860, 0.395433, 0.555276],
853: (17)                                [0.194100, 0.399323, 0.555565],
854: (17)                                [0.192357, 0.403199, 0.555836],
855: (17)                                [0.190631, 0.407061, 0.556089],
856: (17)                                [0.188923, 0.410910, 0.556326],
857: (17)                                [0.187231, 0.414746, 0.556547],
858: (17)                                [0.185556, 0.418570, 0.556753],
859: (17)                                [0.183898, 0.422383, 0.556944],
860: (17)                                [0.182256, 0.426184, 0.557120],
861: (17)                                [0.180629, 0.429975, 0.557282],
862: (17)                                [0.179019, 0.433756, 0.557430],
863: (17)                                [0.177423, 0.437527, 0.557565],
864: (17)                                [0.175841, 0.441290, 0.557685],
865: (17)                                [0.174274, 0.445044, 0.557792],
866: (17)                                [0.172719, 0.448791, 0.557885],
867: (17)                                [0.171176, 0.452530, 0.557965],
868: (17)                                [0.169646, 0.456262, 0.558030],
869: (17)                                [0.168126, 0.459988, 0.558082],
870: (17)                                [0.166617, 0.463708, 0.558119],
871: (17)                                [0.165117, 0.467423, 0.558141],
872: (17)                                [0.163625, 0.471133, 0.558148],
873: (17)                                [0.162142, 0.474838, 0.558140],
874: (17)                                [0.160665, 0.478540, 0.558115],
875: (17)                                [0.159194, 0.482237, 0.558073],
876: (17)                                [0.157729, 0.485932, 0.558013],
877: (17)                                [0.156270, 0.489624, 0.557936],
```

```
878: (17)                                [0.154815, 0.493313, 0.557840],
879: (17)                                [0.153364, 0.497000, 0.557724],
880: (17)                                [0.151918, 0.500685, 0.557587],
881: (17)                                [0.150476, 0.504369, 0.557430],
882: (17)                                [0.149039, 0.508051, 0.557250],
883: (17)                                [0.147607, 0.511733, 0.557049],
884: (17)                                [0.146180, 0.515413, 0.556823],
885: (17)                                [0.144759, 0.519093, 0.556572],
886: (17)                                [0.143343, 0.522773, 0.556295],
887: (17)                                [0.141935, 0.526453, 0.555991],
888: (17)                                [0.140536, 0.530132, 0.555659],
889: (17)                                [0.139147, 0.533812, 0.555298],
890: (17)                                [0.137770, 0.537492, 0.554906],
891: (17)                                [0.136408, 0.541173, 0.554483],
892: (17)                                [0.135066, 0.544853, 0.554029],
893: (17)                                [0.133743, 0.548535, 0.553541],
894: (17)                                [0.132444, 0.552216, 0.553018],
895: (17)                                [0.131172, 0.555899, 0.552459],
896: (17)                                [0.129933, 0.559582, 0.551864],
897: (17)                                [0.128729, 0.563265, 0.551229],
898: (17)                                [0.127568, 0.566949, 0.550556],
899: (17)                                [0.126453, 0.570633, 0.549841],
900: (17)                                [0.125394, 0.574318, 0.549086],
901: (17)                                [0.124395, 0.578002, 0.548287],
902: (17)                                [0.123463, 0.581687, 0.547445],
903: (17)                                [0.122606, 0.585371, 0.546557],
904: (17)                                [0.121831, 0.589055, 0.545623],
905: (17)                                [0.121148, 0.592739, 0.544641],
906: (17)                                [0.120565, 0.596422, 0.543611],
907: (17)                                [0.120092, 0.600104, 0.542530],
908: (17)                                [0.119738, 0.603785, 0.541400],
909: (17)                                [0.119512, 0.607464, 0.540218],
910: (17)                                [0.119423, 0.611141, 0.538982],
911: (17)                                [0.119483, 0.614817, 0.537692],
912: (17)                                [0.119699, 0.618490, 0.536347],
913: (17)                                [0.120081, 0.622161, 0.534946],
914: (17)                                [0.120638, 0.625828, 0.533488],
915: (17)                                [0.121380, 0.629492, 0.531973],
916: (17)                                [0.122312, 0.633153, 0.530398],
917: (17)                                [0.123444, 0.636809, 0.528763],
918: (17)                                [0.124780, 0.640461, 0.527068],
919: (17)                                [0.126326, 0.644107, 0.525311],
920: (17)                                [0.128087, 0.647749, 0.523491],
921: (17)                                [0.130067, 0.651384, 0.521608],
922: (17)                                [0.132268, 0.655014, 0.519661],
923: (17)                                [0.134692, 0.658636, 0.517649],
924: (17)                                [0.137339, 0.662252, 0.515571],
925: (17)                                [0.140210, 0.665859, 0.513427],
926: (17)                                [0.143303, 0.669459, 0.511215],
927: (17)                                [0.146616, 0.673050, 0.508936],
928: (17)                                [0.150148, 0.676631, 0.506589],
929: (17)                                [0.153894, 0.680203, 0.504172],
930: (17)                                [0.157851, 0.683765, 0.501686],
931: (17)                                [0.162016, 0.687316, 0.499129],
932: (17)                                [0.166383, 0.690856, 0.496502],
933: (17)                                [0.170948, 0.694384, 0.493803],
934: (17)                                [0.175707, 0.697900, 0.491033],
935: (17)                                [0.180653, 0.701402, 0.488189],
936: (17)                                [0.185783, 0.704891, 0.485273],
937: (17)                                [0.191090, 0.708366, 0.482284],
938: (17)                                [0.196571, 0.711827, 0.479221],
939: (17)                                [0.202219, 0.715272, 0.476084],
940: (17)                                [0.208030, 0.718701, 0.472873],
941: (17)                                [0.214000, 0.722114, 0.469588],
942: (17)                                [0.220124, 0.725509, 0.466226],
943: (17)                                [0.226397, 0.728888, 0.462789],
944: (17)                                [0.232815, 0.732247, 0.459277],
945: (17)                                [0.239374, 0.735588, 0.455688],
946: (17)                                [0.246070, 0.738910, 0.452024],
```

```
947: (17)                                [0.252899, 0.742211, 0.448284],
948: (17)                                [0.259857, 0.745492, 0.444467],
949: (17)                                [0.266941, 0.748751, 0.440573],
950: (17)                                [0.274149, 0.751988, 0.436601],
951: (17)                                [0.281477, 0.755203, 0.432552],
952: (17)                                [0.288921, 0.758394, 0.428426],
953: (17)                                [0.296479, 0.761561, 0.424223],
954: (17)                                [0.304148, 0.764704, 0.419943],
955: (17)                                [0.311925, 0.767822, 0.415586],
956: (17)                                [0.319809, 0.770914, 0.411152],
957: (17)                                [0.327796, 0.773980, 0.406640],
958: (17)                                [0.335885, 0.777018, 0.402049],
959: (17)                                [0.344074, 0.780029, 0.397381],
960: (17)                                [0.352360, 0.783011, 0.392636],
961: (17)                                [0.360741, 0.785964, 0.387814],
962: (17)                                [0.369214, 0.788888, 0.382914],
963: (17)                                [0.377779, 0.791781, 0.377939],
964: (17)                                [0.386433, 0.794644, 0.372886],
965: (17)                                [0.395174, 0.797475, 0.367757],
966: (17)                                [0.404001, 0.800275, 0.362552],
967: (17)                                [0.412913, 0.803041, 0.357269],
968: (17)                                [0.421908, 0.805774, 0.351910],
969: (17)                                [0.430983, 0.808473, 0.346476],
970: (17)                                [0.440137, 0.811138, 0.340967],
971: (17)                                [0.449368, 0.813768, 0.335384],
972: (17)                                [0.458674, 0.816363, 0.329727],
973: (17)                                [0.468053, 0.818921, 0.323998],
974: (17)                                [0.477504, 0.821444, 0.318195],
975: (17)                                [0.487026, 0.823929, 0.312321],
976: (17)                                [0.496615, 0.826376, 0.306377],
977: (17)                                [0.506271, 0.828786, 0.300362],
978: (17)                                [0.515992, 0.831158, 0.294279],
979: (17)                                [0.525776, 0.833491, 0.288127],
980: (17)                                [0.535621, 0.835785, 0.281908],
981: (17)                                [0.545524, 0.838039, 0.275626],
982: (17)                                [0.555484, 0.840254, 0.269281],
983: (17)                                [0.565498, 0.842430, 0.262877],
984: (17)                                [0.575563, 0.844566, 0.256415],
985: (17)                                [0.585678, 0.846661, 0.249897],
986: (17)                                [0.595839, 0.848717, 0.243329],
987: (17)                                [0.606045, 0.850733, 0.236712],
988: (17)                                [0.616293, 0.852709, 0.230052],
989: (17)                                [0.626579, 0.854645, 0.223353],
990: (17)                                [0.636902, 0.856542, 0.216620],
991: (17)                                [0.647257, 0.858400, 0.209861],
992: (17)                                [0.657642, 0.860219, 0.203082],
993: (17)                                [0.668054, 0.861999, 0.196293],
994: (17)                                [0.678489, 0.863742, 0.189503],
995: (17)                                [0.688944, 0.865448, 0.182725],
996: (17)                                [0.699415, 0.867117, 0.175971],
997: (17)                                [0.709898, 0.868751, 0.169257],
998: (17)                                [0.720391, 0.870350, 0.162603],
999: (17)                                [0.730889, 0.871916, 0.156029],
1000: (17)                               [0.741388, 0.873449, 0.149561],
1001: (17)                               [0.751884, 0.874951, 0.143228],
1002: (17)                               [0.762373, 0.876424, 0.137064],
1003: (17)                               [0.772852, 0.877868, 0.131109],
1004: (17)                               [0.783315, 0.879285, 0.125405],
1005: (17)                               [0.793760, 0.880678, 0.120005],
1006: (17)                               [0.804182, 0.882046, 0.114965],
1007: (17)                               [0.814576, 0.883393, 0.110347],
1008: (17)                               [0.824940, 0.884720, 0.106217],
1009: (17)                               [0.835270, 0.886029, 0.102646],
1010: (17)                               [0.845561, 0.887322, 0.099702],
1011: (17)                               [0.855810, 0.888601, 0.097452],
1012: (17)                               [0.866013, 0.889868, 0.095953],
1013: (17)                               [0.876168, 0.891125, 0.095250],
1014: (17)                               [0.886271, 0.892374, 0.095374],
1015: (17)                               [0.896320, 0.893616, 0.096335],
```

```
(17)                            [0.906311, 0.894855, 0.098125],
(17)                            [0.916242, 0.896091, 0.100717],
(17)                            [0.926106, 0.897330, 0.104071],
(17)                            [0.935904, 0.898570, 0.108131],
(17)                            [0.945636, 0.899815, 0.112838],
(17)                            [0.955300, 0.901065, 0.118128],
(17)                            [0.964894, 0.902323, 0.123941],
(17)                            [0.974417, 0.903590, 0.130215],
(17)                            [0.983868, 0.904867, 0.136897],
(17)                            [0.993248, 0.906157, 0.143936]]
(0)            _cividis_data = [[0.000000, 0.135112, 0.304751],
(17)                            [0.000000, 0.138068, 0.311105],
(17)                            [0.000000, 0.141013, 0.317579],
(17)                            [0.000000, 0.143951, 0.323982],
(17)                            [0.000000, 0.146877, 0.330479],
(17)                            [0.000000, 0.149791, 0.337065],
(17)                            [0.000000, 0.152673, 0.343704],
(17)                            [0.000000, 0.155377, 0.350500],
(17)                            [0.000000, 0.157932, 0.357521],
(17)                            [0.000000, 0.160495, 0.364534],
(17)                            [0.000000, 0.163058, 0.371608],
(17)                            [0.000000, 0.165621, 0.378769],
(17)                            [0.000000, 0.168204, 0.385902],
(17)                            [0.000000, 0.170800, 0.393100],
(17)                            [0.000000, 0.173420, 0.400353],
(17)                            [0.000000, 0.176082, 0.407577],
(17)                            [0.000000, 0.178802, 0.414764],
(17)                            [0.000000, 0.181610, 0.421859],
(17)                            [0.000000, 0.184550, 0.428802],
(17)                            [0.000000, 0.186915, 0.435532],
(17)                            [0.000000, 0.188769, 0.439563],
(17)                            [0.000000, 0.190950, 0.441085],
(17)                            [0.000000, 0.193366, 0.441561],
(17)                            [0.003602, 0.195911, 0.441564],
(17)                            [0.017852, 0.198528, 0.441248],
(17)                            [0.032110, 0.201199, 0.440785],
(17)                            [0.046205, 0.203903, 0.440196],
(17)                            [0.058378, 0.206629, 0.439531],
(17)                            [0.068968, 0.209372, 0.438863],
(17)                            [0.078624, 0.212122, 0.438105],
(17)                            [0.087465, 0.214879, 0.437342],
(17)                            [0.095645, 0.217643, 0.436593],
(17)                            [0.103401, 0.220406, 0.435790],
(17)                            [0.110658, 0.223170, 0.435067],
(17)                            [0.117612, 0.225935, 0.434308],
(17)                            [0.124291, 0.228697, 0.433547],
(17)                            [0.130669, 0.231458, 0.432840],
(17)                            [0.136830, 0.234216, 0.432148],
(17)                            [0.142852, 0.236972, 0.431404],
(17)                            [0.148638, 0.239724, 0.430752],
(17)                            [0.154261, 0.242475, 0.430120],
(17)                            [0.159733, 0.245221, 0.429528],
(17)                            [0.165113, 0.247965, 0.428908],
(17)                            [0.170362, 0.250707, 0.428325],
(17)                            [0.175490, 0.253444, 0.427790],
(17)                            [0.180503, 0.256180, 0.427299],
(17)                            [0.185453, 0.258914, 0.426788],
(17)                            [0.190303, 0.261644, 0.426329],
(17)                            [0.195057, 0.264372, 0.425924],
(17)                            [0.199764, 0.267099, 0.425497],
(17)                            [0.204385, 0.269823, 0.425126],
(17)                            [0.208926, 0.272546, 0.424809],
(17)                            [0.213431, 0.275266, 0.424480],
(17)                            [0.217863, 0.277985, 0.424206],
(17)                            [0.222264, 0.280702, 0.423914],
(17)                            [0.226598, 0.283419, 0.423678],
(17)                            [0.230871, 0.286134, 0.423498],
(17)                            [0.235120, 0.288848, 0.423304],
(17)                            [0.239312, 0.291562, 0.423167],
```

```
1085: (17)                              [0.243485, 0.294274, 0.423014],
1086: (17)                              [0.247605, 0.296986, 0.422917],
1087: (17)                              [0.251675, 0.299698, 0.422873],
1088: (17)                              [0.255731, 0.302409, 0.422814],
1089: (17)                              [0.259740, 0.305120, 0.422810],
1090: (17)                              [0.263738, 0.307831, 0.422789],
1091: (17)                              [0.267693, 0.310542, 0.422821],
1092: (17)                              [0.271639, 0.313253, 0.422837],
1093: (17)                              [0.275513, 0.315965, 0.422979],
1094: (17)                              [0.279411, 0.318677, 0.423031],
1095: (17)                              [0.283240, 0.321390, 0.423211],
1096: (17)                              [0.287065, 0.324103, 0.423373],
1097: (17)                              [0.290884, 0.326816, 0.423517],
1098: (17)                              [0.294669, 0.329531, 0.423716],
1099: (17)                              [0.298421, 0.332247, 0.423973],
1100: (17)                              [0.302169, 0.334963, 0.424213],
1101: (17)                              [0.305886, 0.337681, 0.424512],
1102: (17)                              [0.309601, 0.340399, 0.424790],
1103: (17)                              [0.313287, 0.343120, 0.425120],
1104: (17)                              [0.316941, 0.345842, 0.425512],
1105: (17)                              [0.320595, 0.348565, 0.425889],
1106: (17)                              [0.324250, 0.351289, 0.426250],
1107: (17)                              [0.327875, 0.354016, 0.426670],
1108: (17)                              [0.331474, 0.356744, 0.427144],
1109: (17)                              [0.335073, 0.359474, 0.427605],
1110: (17)                              [0.338673, 0.362206, 0.428053],
1111: (17)                              [0.342246, 0.364939, 0.428559],
1112: (17)                              [0.345793, 0.367676, 0.429127],
1113: (17)                              [0.349341, 0.370414, 0.429685],
1114: (17)                              [0.352892, 0.373153, 0.430226],
1115: (17)                              [0.356418, 0.375896, 0.430823],
1116: (17)                              [0.359916, 0.378641, 0.431501],
1117: (17)                              [0.363446, 0.381388, 0.432075],
1118: (17)                              [0.366923, 0.384139, 0.432796],
1119: (17)                              [0.370430, 0.386890, 0.433428],
1120: (17)                              [0.373884, 0.389646, 0.434209],
1121: (17)                              [0.377371, 0.392404, 0.434890],
1122: (17)                              [0.380830, 0.395164, 0.435653],
1123: (17)                              [0.384268, 0.397928, 0.436475],
1124: (17)                              [0.387705, 0.400694, 0.437305],
1125: (17)                              [0.391151, 0.403464, 0.438096],
1126: (17)                              [0.394568, 0.406236, 0.438986],
1127: (17)                              [0.397991, 0.409011, 0.439848],
1128: (17)                              [0.401418, 0.411790, 0.440708],
1129: (17)                              [0.404820, 0.414572, 0.441642],
1130: (17)                              [0.408226, 0.417357, 0.442570],
1131: (17)                              [0.411607, 0.420145, 0.443577],
1132: (17)                              [0.414992, 0.422937, 0.444578],
1133: (17)                              [0.418383, 0.425733, 0.445560],
1134: (17)                              [0.421748, 0.428531, 0.446640],
1135: (17)                              [0.425120, 0.431334, 0.447692],
1136: (17)                              [0.428462, 0.434140, 0.448864],
1137: (17)                              [0.431817, 0.436950, 0.449982],
1138: (17)                              [0.435168, 0.439763, 0.451134],
1139: (17)                              [0.438504, 0.442580, 0.452341],
1140: (17)                              [0.441810, 0.445402, 0.453659],
1141: (17)                              [0.445148, 0.448226, 0.454885],
1142: (17)                              [0.448447, 0.451053, 0.456264],
1143: (17)                              [0.451759, 0.453887, 0.457582],
1144: (17)                              [0.455072, 0.456718, 0.458976],
1145: (17)                              [0.458366, 0.459552, 0.460457],
1146: (17)                              [0.461616, 0.462405, 0.461969],
1147: (17)                              [0.464947, 0.465241, 0.463395],
1148: (17)                              [0.468254, 0.468083, 0.464908],
1149: (17)                              [0.471501, 0.470960, 0.466357],
1150: (17)                              [0.474812, 0.473832, 0.467681],
1151: (17)                              [0.478186, 0.476699, 0.468845],
1152: (17)                              [0.481622, 0.479573, 0.469767],
1153: (17)                              [0.485141, 0.482451, 0.470384],
```

```
1154: (17)                              [0.488697, 0.485318, 0.471008],
1155: (17)                              [0.492278, 0.488198, 0.471453],
1156: (17)                              [0.495913, 0.491076, 0.471751],
1157: (17)                              [0.499552, 0.493960, 0.472032],
1158: (17)                              [0.503185, 0.496851, 0.472305],
1159: (17)                              [0.506866, 0.499743, 0.472432],
1160: (17)                              [0.510540, 0.502643, 0.472550],
1161: (17)                              [0.514226, 0.505546, 0.472640],
1162: (17)                              [0.517920, 0.508454, 0.472707],
1163: (17)                              [0.521643, 0.511367, 0.472639],
1164: (17)                              [0.525348, 0.514285, 0.472660],
1165: (17)                              [0.529086, 0.517207, 0.472543],
1166: (17)                              [0.532829, 0.520135, 0.472401],
1167: (17)                              [0.536553, 0.523067, 0.472352],
1168: (17)                              [0.540307, 0.526005, 0.472163],
1169: (17)                              [0.544069, 0.528948, 0.471947],
1170: (17)                              [0.547840, 0.531895, 0.471704],
1171: (17)                              [0.551612, 0.534849, 0.471439],
1172: (17)                              [0.555393, 0.537807, 0.471147],
1173: (17)                              [0.559181, 0.540771, 0.470829],
1174: (17)                              [0.562972, 0.543741, 0.470488],
1175: (17)                              [0.566802, 0.546715, 0.469988],
1176: (17)                              [0.570607, 0.549695, 0.469593],
1177: (17)                              [0.574417, 0.552682, 0.469172],
1178: (17)                              [0.578236, 0.555673, 0.468724],
1179: (17)                              [0.582087, 0.558670, 0.468118],
1180: (17)                              [0.585916, 0.561674, 0.467618],
1181: (17)                              [0.589753, 0.564682, 0.467090],
1182: (17)                              [0.593622, 0.567697, 0.466401],
1183: (17)                              [0.597469, 0.570718, 0.465821],
1184: (17)                              [0.601354, 0.573743, 0.465074],
1185: (17)                              [0.605211, 0.576777, 0.464441],
1186: (17)                              [0.609105, 0.579816, 0.463638],
1187: (17)                              [0.612977, 0.582861, 0.462950],
1188: (17)                              [0.616852, 0.585913, 0.462237],
1189: (17)                              [0.620765, 0.588970, 0.461351],
1190: (17)                              [0.624654, 0.592034, 0.460583],
1191: (17)                              [0.628576, 0.595104, 0.459641],
1192: (17)                              [0.632506, 0.598180, 0.458668],
1193: (17)                              [0.636412, 0.601264, 0.457818],
1194: (17)                              [0.640352, 0.604354, 0.456791],
1195: (17)                              [0.644270, 0.607450, 0.455886],
1196: (17)                              [0.648222, 0.610553, 0.454801],
1197: (17)                              [0.652178, 0.613664, 0.453689],
1198: (17)                              [0.656114, 0.616780, 0.452702],
1199: (17)                              [0.660082, 0.619904, 0.451534],
1200: (17)                              [0.664055, 0.623034, 0.450338],
1201: (17)                              [0.668008, 0.626171, 0.449270],
1202: (17)                              [0.671991, 0.629316, 0.448018],
1203: (17)                              [0.675981, 0.632468, 0.446736],
1204: (17)                              [0.679979, 0.635626, 0.445424],
1205: (17)                              [0.683950, 0.638793, 0.444251],
1206: (17)                              [0.687957, 0.641966, 0.442886],
1207: (17)                              [0.691971, 0.645145, 0.441491],
1208: (17)                              [0.695985, 0.648334, 0.440072],
1209: (17)                              [0.700008, 0.651529, 0.438624],
1210: (17)                              [0.704037, 0.654731, 0.437147],
1211: (17)                              [0.708067, 0.657942, 0.435647],
1212: (17)                              [0.712105, 0.661160, 0.434117],
1213: (17)                              [0.716177, 0.664384, 0.432386],
1214: (17)                              [0.720222, 0.667618, 0.430805],
1215: (17)                              [0.724274, 0.670859, 0.429194],
1216: (17)                              [0.728334, 0.674107, 0.427554],
1217: (17)                              [0.732422, 0.677364, 0.425717],
1218: (17)                              [0.736488, 0.680629, 0.424028],
1219: (17)                              [0.740589, 0.683900, 0.422131],
1220: (17)                              [0.744664, 0.687181, 0.420393],
1221: (17)                              [0.748772, 0.690470, 0.418448],
1222: (17)                              [0.752886, 0.693766, 0.416472],
```

```
(17)                            [0.756975, 0.697071, 0.414659],
(17)                            [0.761096, 0.700384, 0.412638],
(17)                            [0.765223, 0.703705, 0.410587],
(17)                            [0.769353, 0.707035, 0.408516],
(17)                            [0.773486, 0.710373, 0.406422],
(17)                            [0.777651, 0.713719, 0.404112],
(17)                            [0.781795, 0.717074, 0.401966],
(17)                            [0.785965, 0.720438, 0.399613],
(17)                            [0.790116, 0.723810, 0.397423],
(17)                            [0.794298, 0.727190, 0.395016],
(17)                            [0.798480, 0.730580, 0.392597],
(17)                            [0.802667, 0.733978, 0.390153],
(17)                            [0.806859, 0.737385, 0.387684],
(17)                            [0.811054, 0.740801, 0.385198],
(17)                            [0.815274, 0.744226, 0.382504],
(17)                            [0.819499, 0.747659, 0.379785],
(17)                            [0.823729, 0.751101, 0.377043],
(17)                            [0.827959, 0.754553, 0.374292],
(17)                            [0.832192, 0.758014, 0.371529],
(17)                            [0.836429, 0.761483, 0.368747],
(17)                            [0.840693, 0.764962, 0.365746],
(17)                            [0.844957, 0.768450, 0.362741],
(17)                            [0.849223, 0.771947, 0.359729],
(17)                            [0.853515, 0.775454, 0.356500],
(17)                            [0.857809, 0.778969, 0.353259],
(17)                            [0.862105, 0.782494, 0.350011],
(17)                            [0.866421, 0.786028, 0.346571],
(17)                            [0.870717, 0.789572, 0.343333],
(17)                            [0.875057, 0.793125, 0.339685],
(17)                            [0.879378, 0.796687, 0.336241],
(17)                            [0.883720, 0.800258, 0.332599],
(17)                            [0.888081, 0.803839, 0.328770],
(17)                            [0.892440, 0.807430, 0.324968],
(17)                            [0.896818, 0.811030, 0.320982],
(17)                            [0.901195, 0.814639, 0.317021],
(17)                            [0.905589, 0.818257, 0.312889],
(17)                            [0.910000, 0.821885, 0.308594],
(17)                            [0.914407, 0.825522, 0.304348],
(17)                            [0.918828, 0.829168, 0.299960],
(17)                            [0.923279, 0.832822, 0.295244],
(17)                            [0.927724, 0.836486, 0.290611],
(17)                            [0.932180, 0.840159, 0.285880],
(17)                            [0.936660, 0.843841, 0.280876],
(17)                            [0.941147, 0.847530, 0.275815],
(17)                            [0.945654, 0.851228, 0.270532],
(17)                            [0.950178, 0.854933, 0.265085],
(17)                            [0.954725, 0.858646, 0.259365],
(17)                            [0.959284, 0.862365, 0.253563],
(17)                            [0.963872, 0.866089, 0.247445],
(17)                            [0.968469, 0.869819, 0.241310],
(17)                            [0.973114, 0.873550, 0.234677],
(17)                            [0.977780, 0.877281, 0.227954],
(17)                            [0.982497, 0.881008, 0.220878],
(17)                            [0.987293, 0.884718, 0.213336],
(17)                            [0.992218, 0.888385, 0.205468],
(17)                            [0.994847, 0.892954, 0.203445],
(17)                            [0.995249, 0.898384, 0.207561],
(17)                            [0.995503, 0.903866, 0.212370],
(17)                            [0.995737, 0.909344, 0.217772]]
(0)            _twilight_data = [
(4)                [0.88575015840754434, 0.85000924943067835,  0.8879736506427196],
(4)                [0.88378520195539056, 0.85072940540310626,  0.88723222096949894],
(4)                [0.88172231059285788, 0.85127594077653468,  0.88638056925514819],
(4)                [0.8795410528270573,  0.85165675407495722,  0.8854143767924102],
(4)                [0.87724880858965482, 0.85187028338870274,  0.88434120381311432],
(4)                [0.87485347508575972, 0.85191526123023187,  0.88316926967613829],
(4)                [0.87233134085124076, 0.85180165478080894,  0.88189704355001619],
(4)                [0.86970474853509816, 0.85152403004797894,  0.88053883390003362],
(4)                [0.86696015505333579, 0.8510896085314068,   0.87909766977173343],
```

```
1292: (4)                [0.86408985081463996, 0.85050391167507788,  0.87757925784892632],
1293: (4)                [0.86110245436899846, 0.84976754857001258,  0.87599242923439569],
1294: (4)                [0.85798259245670372, 0.84888934810281835,  0.87434038553446281],
1295: (4)                [0.85472593189256985, 0.84787488124672816,  0.8726282980930582],
1296: (4)                [0.85133714570857189, 0.84672735796116472,  0.87086081657350445],
1297: (4)                [0.84780710702577922, 0.8454546229209523,   0.86904036783694438],
1298: (4)                [0.8441261828674842,  0.84406482711037389,  0.86716973322690072],
1299: (4)                [0.84030420805957784, 0.8425605950855084,   0.865250882410458],
1300: (4)                [0.83634031809191178, 0.84094796518951942,  0.86328528001070159],
1301: (4)                [0.83222705712934408, 0.83923490627754482,  0.86127563500427884],
1302: (4)                [0.82796894316013536, 0.83742600751395202,  0.85922399451306786],
1303: (4)                [0.82357429680252847, 0.83552487764795436,  0.85713191328514948],
1304: (4)                [0.81904654677937527, 0.8335364929949034,   0.85500206287010105],
1305: (4)                [0.81438982121143089, 0.83146558694197847,  0.85283759062147024],
1306: (4)                [0.8095999819094809,  0.82931896673505456,  0.85064441601050367],
1307: (4)                [0.80469164429814577, 0.82709838780560663,  0.84842449296974021],
1308: (4)                [0.79967075421267997, 0.82480781812080928,  0.84618210029578533],
1309: (4)                [0.79454305089231114, 0.82245116226304615,  0.84392184786827984],
1310: (4)                [0.78931445564608915, 0.82003213188702007,  0.8416486380471222],
1311: (4)                [0.78399101042764918, 0.81755426400533426,  0.83936747464036732],
1312: (4)                [0.77857892008227592, 0.81502089378742548,  0.8370834463093898],
1313: (4)                [0.77308416590170936, 0.81243524735466011,  0.83480172950579679],
1314: (4)                [0.76751108504417864, 0.8098007598713145,   0.83252816638059668],
1315: (4)                [0.76186907937980286, 0.80711949387647486,  0.830266486168872],
1316: (4)                [0.75616443584381976, 0.80439408733477935,  0.82802138994719998],
1317: (4)                [0.75040346765406696, 0.80162699008965321,  0.82579737851082424],
1318: (4)                [0.74459247771890169, 0.79882047719583249,  0.82359867586156521],
1319: (4)                [0.73873771700494939, 0.79597665735031009,  0.82142922780433014],
1320: (4)                [0.73284543645523459, 0.79309746468844067,  0.81929263384230377],
1321: (4)                [0.72692177512829703, 0.7901846863592763,   0.81719217466726379],
1322: (4)                [0.72097280665536778, 0.78723995923452639,  0.81513073920879264],
1323: (4)                [0.71500403076252128, 0.78426487091581187,  0.81311116559949914],
1324: (4)                [0.70902078134539304, 0.78126088716070907,  0.81113591855117928],
1325: (4)                [0.7030297722540817,  0.77822904973358131,  0.80920618848056969],
1326: (4)                [0.6970365443886174,  0.77517050008066057,  0.80732335380063447],
1327: (4)                [0.69104641009309098, 0.77208629460678091,  0.80548841690679074],
1328: (4)                [0.68506446154395928, 0.7689774029354699,   0.80370206267176914],
1329: (4)                [0.67909554499882152, 0.76584472131395898,  0.8019646617300199],
1330: (4)                [0.67314422559426212, 0.76268908733890484,  0.80027628545809526],
1331: (4)                [0.66721479803752815, 0.7595112803730375,   0.79863674654537764],
1332: (4)                [0.6613112930078745,  0.75631202708719025,  0.7970456043491897],
1333: (4)                [0.65543692326454717, 0.75309208756768431,  0.79550271129031047],
1334: (4)                [0.64959573004253479, 0.74985201221941766,  0.79400674021499107],
1335: (4)                [0.6437910831099849,  0.7465923800833657,   0.79255653201306053],
1336: (4)                [0.63802586828545982, 0.74331376714033193,  0.79115100459573173],
1337: (4)                [0.6323027138710603,  0.74001672160131404,  0.78978892762640429],
1338: (4)                [0.62662402022604591, 0.73670175403699445,  0.78846901316334561],
1339: (4)                [0.62099193064817548, 0.73336934798923203,  0.78718994624696581],
1340: (4)                [0.61540846411770478, 0.73001995232739691,  0.78595022706750484],
1341: (4)                [0.60987543176093062, 0.72665398759758293,  0.78474835732694714],
1342: (4)                [0.60439434200274855, 0.7232718614323369,   0.78358295593535587],
1343: (4)                [0.5989665814482068,  0.71987394892246725,  0.78245259899346642],
1344: (4)                [0.59359335696837223, 0.7164606049658685,   0.78135588237640097],
1345: (4)                [0.58827579780555495, 0.71303214646458135,  0.78029141405636515],
1346: (4)                [0.58301487036932409, 0.70958887676997473,  0.77925781820476592],
1347: (4)                [0.5778116438998202,  0.70613106157153982,  0.77825345121025524],
1348: (4)                [0.5726668948158774,  0.7026589535425779,   0.77727702680911992],
1349: (4)                [0.56758117853861967, 0.69917279302646274,  0.77632748534275298],
1350: (4)                [0.56255515357219343, 0.69567278381629649,  0.77540359142309845],
1351: (4)                [0.55758940419605174, 0.69215911458254054,  0.7745041337932782],
1352: (4)                [0.55268450589347129, 0.68863194515166382,  0.7736279426902245],
1353: (4)                [0.54784098153018634, 0.68509142218509878,  0.77277386473440868],
1354: (4)                [0.54305932424018233, 0.68153767253065878,  0.77194079697835083],
1355: (4)                [0.53834015575176275, 0.67797081129095405,  0.77112734439057717],
1356: (4)                [0.53368389147728401, 0.67439093705212727,  0.7703325054879735],
1357: (4)                [0.529090861832473,   0.67079812302806219,  0.76955552292313134],
1358: (4)                [0.52456151470593582, 0.66719242996142225,  0.76879541714230948],
1359: (4)                [0.52009627392235558, 0.66357391434030388,  0.76805119403344102],
1360: (4)                [0.5156955988596057,  0.65994260812897998,  0.76732191489596169],
```

```
1361: (4)                [0.51135992541601927, 0.65629853981831865,  0.76660663780645333],
1362: (4)                [0.50708969576451657, 0.65264172403146448,  0.76590445660835849],
1363: (4)                [0.5028853540415561,  0.64897216734095264,  0.76521446718174913],
1364: (4)                [0.49874733661356069, 0.6452898684900934,   0.76453578734180083],
1365: (4)                [0.4946761847863938,  0.64159484119504429,  0.76386719002130909],
1366: (4)                [0.49067224938561221, 0.63788704858847078,  0.76320812763163837],
1367: (4)                [0.4867359599430568,  0.63416646251100506,  0.76255780085924041],
1368: (4)                [0.4828677867260272,  0.6304330455306234,   0.76191537149895305],
1369: (4)                [0.47906816236197386, 0.62668676251860134,  0.76128000375662419],
1370: (4)                [0.47533752394906287, 0.62292757283835809,  0.76065085571817748],
1371: (4)                [0.47167629518877091, 0.61915543242884641,  0.76002709227883047],
1372: (4)                [0.46808490970531597, 0.61537028695790286,  0.75940789891092741],
1373: (4)                [0.46456376716303932, 0.61157208822864151,  0.75879242623025811],
1374: (4)                [0.46111326647023881, 0.607760777169989,    0.75817986436807139],
1375: (4)                [0.45773377230160567, 0.60393630046586455,  0.75756936901859162],
1376: (4)                [0.45442563977552913, 0.60009859503858665,  0.75696013660606487],
1377: (4)                [0.45118918687617743, 0.59624762051353541,  0.75635120643246645],
1378: (4)                [0.44802470933589172, 0.59238331452146575,  0.75574176474107924],
1379: (4)                [0.44493246854215379, 0.5885055998308617,   0.7551311041857901],
1380: (4)                [0.44191271766696399, 0.58461441100175571,  0.75451838884410671],
1381: (4)                [0.43896563958048396, 0.58070969241098491,  0.75390276208285945],
1382: (4)                [0.43609138958356369, 0.57679137998186081,  0.7532834105961016],
1383: (4)                [0.43329008867358393, 0.57285941625606673,  0.75265946532566674],
1384: (4)                [0.43056179073057571, 0.56891374572457176,  0.75203008099312696],
1385: (4)                [0.42790652284925834, 0.5649543060909209,   0.75139443521914839],
1386: (4)                [0.42532423665011354, 0.56098104959950301,  0.75075164989005116],
1387: (4)                [0.42281485675772662, 0.55699392126996583,  0.75010086988227642],
1388: (4)                [0.42037822361396326, 0.55299287158108168,  0.7494412559451894],
1389: (4)                [0.41801414079233629, 0.54897785421888889,  0.74877193167001121],
1390: (4)                [0.4157223260454232,  0.54494882715350401,  0.74809204459000522],
1391: (4)                [0.41350245743314729, 0.54090574771098476,  0.74740073297543086],
1392: (4)                [0.41135414697304568, 0.53684857765005933,  0.74669712855065784],
1393: (4)                [0.4092768899914751,  0.53277730177130322,  0.74598030635707824],
1394: (4)                [0.40727018694219069, 0.52869188011057411,  0.74524942637581271],
1395: (4)                [0.40533343789303178, 0.52459228174983119,  0.74450365836708132],
1396: (4)                [0.40346600333905397, 0.52047847653840029,  0.74374215223567086],
1397: (4)                [0.40166714010896104, 0.51635044969688759,  0.7429640345324835],
1398: (4)                [0.39993606933454834, 0.51220818143218516,  0.74216844571317986],
1399: (4)                [0.3982719152586337,  0.50805166539276136,  0.74135450918099721],
1400: (4)                [0.39667374905665609, 0.50388089053847973,  0.74052138580516735],
1401: (4)                [0.39514058808207631, 0.49969585326377758,  0.73966820211715711],
1402: (4)                [0.39367135736822567, 0.49549655777451179,  0.738794102296364],
1403: (4)                [0.39226494876209317, 0.49128300332899261,  0.73789824784475078],
1404: (4)                [0.39092017571994903, 0.48705520251223039,  0.73697977133881254],
1405: (4)                [0.38963580160340855, 0.48281316715123496,  0.73603782546932739],
1406: (4)                [0.38841053300842432, 0.47855691131792805,  0.73507157641157261],
1407: (4)                [0.38724301459330251, 0.47428645933635388,  0.73408016787854391],
1408: (4)                [0.38613184178892102, 0.4700018340988123,   0.7330627749243106],
1409: (4)                [0.38507556793651387, 0.46570306719930193,  0.73201854033690505],
1410: (4)                [0.38407269378943537, 0.46139018782416635,  0.73094665432902683],
1411: (4)                [0.38312168084402748, 0.45706323581407199,  0.72984626791353258],
1412: (4)                [0.38222094988570376, 0.45272225034283325,  0.72871656144003782],
1413: (4)                [0.38136887930454161, 0.44836727669277859,  0.72755671317141346],
1414: (4)                [0.38056380696565623, 0.44399837208633719,  0.72636587045135315],
1415: (4)                [0.37980403744848751, 0.43961558821222629,  0.72514323778761092],
1416: (4)                [0.37908789283110761, 0.43521897612544935,  0.72388798691323131],
1417: (4)                [0.378413635091359,   0.43080859411413064,  0.72259931993061044],
1418: (4)                [0.37777949753513729, 0.4263845142616835,   0.72127639993530235],
1419: (4)                [0.37718371844251231, 0.42194680223454828,  0.71991841524475775],
1420: (4)                [0.37662448930806297, 0.41749553747893614,  0.71852454736176108],
1421: (4)                [0.37610001286385814, 0.41303079952477062,  0.71709396919920232],
1422: (4)                [0.37560846919442398, 0.40855267638072096,  0.71562585091587549],
1423: (4)                [0.37514802505380473, 0.4040612609993941,   0.7141193695725726],
1424: (4)                [0.37471686019302231, 0.3995566498711684,   0.71257368516500463],
1425: (4)                [0.37431313199312338, 0.39503894828283309,  0.71098796522377461],
1426: (4)                [0.37393499330475782, 0.39050827529375831,  0.70936134293478448],
1427: (4)                [0.3735806215098284,  0.38596474386057539,  0.70769297607310577],
1428: (4)                [0.37324816143326384, 0.38140848555753937,  0.70598200974806036],
1429: (4)                [0.37293578646665032, 0.37683963835219841,  0.70422755780589941],
```

```
1430: (4)                [0.37264166757849604, 0.37225835004836849,  0.7024287314570723],
1431: (4)                [0.37236397858465387, 0.36766477862108266,  0.70058463496520773],
1432: (4)                [0.37210089702443822, 0.36305909736982378,  0.69869434615073722],
1433: (4)                [0.3718506155898596,  0.35844148285875221,  0.69675695810256544],
1434: (4)                [0.37161133234400479, 0.3538121372967869,   0.69477149919380887],
1435: (4)                [0.37138124223736607, 0.34917126878479027,  0.69273703471928827],
1436: (4)                [0.37115856636209105, 0.34451911410230168,  0.69065253586464992],
1437: (4)                [0.37094151551337329, 0.33985591488818123,  0.68851703379505125],
1438: (4)                [0.37072833279422668, 0.33518193808489577,  0.68632948169606767],
1439: (4)                [0.37051738634484427, 0.33049741244307851,  0.68408888788857214],
1440: (4)                [0.37030682071842685, 0.32580269697872455,  0.68179411684486679],
1441: (4)                [0.37009487130772695, 0.3210981375964933,   0.67944405399056851],
1442: (4)                [0.36987980329025361, 0.31638410101153364,  0.67703755438090574],
1443: (4)                [0.36965987626565955, 0.31166098762951971,  0.67457344743419545],
1444: (4)                [0.36943334591276228, 0.30692923551862339,  0.67205052849120617],
1445: (4)                [0.36919847837592484, 0.30218932176507068,  0.66946754331614522],
1446: (4)                [0.36895355306596778, 0.29744175492366276,  0.66682322089824264],
1447: (4)                [0.36869682231895268, 0.29268709856150099,  0.66411625298236909],
1448: (4)                [0.36842655638020444, 0.28792596437778462,  0.66134526910944602],
1449: (4)                [0.36814101479899719, 0.28315901221182987,  0.65850888806972308],
1450: (4)                [0.36783843696531082, 0.27838697181297761,  0.65560566838453704],
1451: (4)                [0.36751707094367697, 0.27361063317090978,  0.65263411711618635],
1452: (4)                [0.36717513650699446, 0.26883085667326956,  0.64959272297892245],
1453: (4)                [0.36681085540107988, 0.26404857724525643,  0.64647991652908243],
1454: (4)                [0.36642243251550632, 0.25926481158628106,  0.64329409140765537],
1455: (4)                [0.36600853966739794, 0.25448043878086224,  0.64003361803368586],
1456: (4)                [0.36556698373538982, 0.24969683475296395,  0.63669675187488584],
1457: (4)                [0.36509579845886808, 0.24491536803550484,  0.63328173520055586],
1458: (4)                [0.36459308890125008, 0.24013747024823828,  0.62978680155026101],
1459: (4)                [0.36405693022088509, 0.23536470386204195,  0.62621013451953023],
1460: (4)                [0.36348537610385145, 0.23059876218396419,  0.62254988622392882],
1461: (4)                [0.36287643560041027, 0.22584149293287031,  0.61880417410823019],
1462: (4)                [0.36222809558295926, 0.22109488427338303,  0.61497112346096128],
1463: (4)                [0.36153829010998356, 0.21636111429594002,  0.61104880679640927],
1464: (4)                [0.36080493826624654, 0.21164251793458128,  0.60703532172064711],
1465: (4)                [0.36002681809096376, 0.20694122817889948,  0.60292845431916875],
1466: (4)                [0.35920088560930186, 0.20226037920758122,  0.5987265295935138],
1467: (4)                [0.35832489966617809, 0.197602942459778,    0.59442768517501066],
1468: (4)                [0.35739663292915563, 0.19297208197842461,  0.59003011251063131],
1469: (4)                [0.35641381143126327, 0.18837119869242164,  0.5855320765920552],
1470: (4)                [0.35537415306906722, 0.18380392577704466,  0.58093191431832802],
1471: (4)                [0.35427534960663759, 0.17927413271618647,  0.57622809660668717],
1472: (4)                [0.35311574421123737, 0.17478570377561287,  0.57141871523555288],
1473: (4)                [0.35189248608873791, 0.17034320478524959,  0.56650284911216653],
1474: (4)                [0.35060304441931012, 0.16595129984720861,  0.56147964703993225],
1475: (4)                [0.34924513554955644, 0.16161477763045118,  0.55634837474163779],
1476: (4)                [0.34781653238777782, 0.15733863511152979,  0.55110853452703257],
1477: (4)                [0.34631507175793091, 0.15312802296627787,  0.5457599924248665],
1478: (4)                [0.34473901574536375, 0.14898820589826409,  0.54030245920406539],
1479: (4)                [0.34308600291572294, 0.14492465359918028,  0.53473704282067103],
1480: (4)                [0.34135411074506483, 0.1409427920655632,   0.52906500940336754],
1481: (4)                [0.33954168752669694, 0.13704801896718169,  0.52328797535085236],
1482: (4)                [0.33764732090671112, 0.13324562282438077,  0.51740807573979475],
1483: (4)                [0.33566978565015315, 0.12954074251271822,  0.51142807215168951],
1484: (4)                [0.33360804901486002, 0.12593818301005921,  0.50535164796654897],
1485: (4)                [0.33146154891145124, 0.12244245263391232,  0.49918274588431072],
1486: (4)                [0.32923005203231409, 0.11905764321981127,  0.49292595612342666],
1487: (4)                [0.3269137124539796,  0.1157873496841953,   0.48658646495697461],
1488: (4)                [0.32451307931207785, 0.11263459791730848,  0.48017007211645196],
1489: (4)                [0.32202882276069322, 0.10960114111258401,  0.47368494725726878],
1490: (4)                [0.31946262395497965, 0.10668879882392659,  0.46713728801395243],
1491: (4)                [0.31681648089023501, 0.10389861387653518,  0.46053414662739794],
1492: (4)                [0.31409278414755532, 0.10123077676403242,  0.45388335612058467],
1493: (4)                [0.31129434479712365, 0.098684771934052201, 0.44719313715161618],
1494: (4)                [0.30842444457210105, 0.096259385340577736, 0.44047194882050544],
1495: (4)                [0.30548675819945936, 0.093952764840823738, 0.43372849999361113],
1496: (4)                [0.30248536364574252, 0.091761187397303601, 0.42697404043749887],
1497: (4)                [0.29942483960214772, 0.089682253716750038, 0.42021619665853854],
1498: (4)                [0.29631000388905288, 0.087713250960463951, 0.41346259134143476],
```

```
1499: (4)                [0.29314593096985248, 0.085850656889620708, 0.40672178082365834],
1500: (4)                [0.28993792445176608, 0.08409078829085731,  0.40000214725256295],
1501: (4)                [0.28669151388283165, 0.082429873848480689, 0.39331182532243375],
1502: (4)                [0.28341239797185225, 0.080864153365499375, 0.38665868550105914],
1503: (4)                [0.28010638576975472, 0.079389994802261526, 0.38005028528138707],
1504: (4)                [0.27677939615815589, 0.078003941033788216, 0.37349382846504675],
1505: (4)                [0.27343739342450812, 0.076702800237496066, 0.36699616136347685],
1506: (4)                [0.27008637749114051, 0.075483675584275545, 0.36056376228111864],
1507: (4)                [0.26673233211995284, 0.074344018028546205, 0.35420276066240958],
1508: (4)                [0.26338121807151404, 0.073281657939897077, 0.34791888996380105],
1509: (4)                [0.26003895187439957, 0.072294781043362205, 0.3417175669546984],
1510: (4)                [0.25671191651083902, 0.071380106242082242, 0.33560648984600089],
1511: (4)                [0.25340685873736807, 0.070533582926851829, 0.3295945757321303],
1512: (4)                [0.25012845306199383, 0.069758206429106989, 0.32368100685760637],
1513: (4)                [0.24688226237958999, 0.069053639449204451, 0.31786993834254956],
1514: (4)                [0.24367372557466271, 0.068419855150922693, 0.31216524050888372],
1515: (4)                [0.24050813332295939, 0.067857103814855602, 0.30657054493678321],
1516: (4)                [0.23739062429054825, 0.067365888050555517, 0.30108922184065873],
1517: (4)                [0.23433055727563878, 0.066935599661639394, 0.29574009929867601],
1518: (4)                [0.23132955273021344, 0.066576186939090592, 0.29051361067988485],
1519: (4)                [0.2283917709422868,  0.06628997924139618,  0.28541074411068496],
1520: (4)                [0.22552164337737857, 0.066078173119395595, 0.28043398847505197],
1521: (4)                [0.22272706739121817, 0.065933790675651943, 0.27559714652053702],
1522: (4)                [0.22001251100779617, 0.065857918918907604, 0.27090279994325861],
1523: (4)                [0.21737845072382705, 0.065859661233562045, 0.26634209349669508],
1524: (4)                [0.21482843531473683, 0.065940385613778491, 0.26191675992376573],
1525: (4)                [0.21237411048541005, 0.066085024661758446, 0.25765165093569542],
1526: (4)                [0.21001214221188125, 0.066308573918947178, 0.2535289048041211],
1527: (4)                [0.2077442377448806,  0.06661453200418091,  0.24954644291943817],
1528: (4)                [0.20558051999470117, 0.066990462397868739, 0.24572497420147632],
1529: (4)                [0.20352007949514977, 0.067444179612424215, 0.24205576625191821],
1530: (4)                [0.20156133764129841, 0.067983271026200248, 0.23852974228695395],
1531: (4)                [0.19971571438603364, 0.068592710553704722, 0.23517094067076993],
1532: (4)                [0.19794834061899208, 0.069314066071660657, 0.23194647381302336],
1533: (4)                [0.1960826032659409,  0.070321227242423623, 0.22874673279569585],
1534: (4)                [0.19410351363791453, 0.071608304856891569, 0.22558727307410353],
1535: (4)                [0.19199449184606268, 0.073182830649273306, 0.22243385243433622],
1536: (4)                [0.18975853639094634, 0.075019861862143766, 0.2193005075652994],
1537: (4)                [0.18739228342697645, 0.077102096899588329, 0.21618875376309582],
1538: (4)                [0.18488035509396164, 0.079425730279723883, 0.21307651648984993],
1539: (4)                [0.18774482037046955, 0.077251588468039312, 0.21387448578597812],
1540: (4)                [0.19049578401722037, 0.075311278416787641, 0.2146562337112265],
1541: (4)                [0.1931548636579131,  0.073606819040117955, 0.21542362939081539],
1542: (4)                [0.19571853588267552, 0.072157781039602742, 0.21617499187076789],
1543: (4)                [0.19819343656336558, 0.070974625252738788, 0.21690975060032436],
1544: (4)                [0.20058760685133747, 0.070064576149984209, 0.21762721310371608],
1545: (4)                [0.20290365333558247, 0.069435248580458964, 0.21833167885096033],
1546: (4)                [0.20531725273301316, 0.068919592266397572, 0.21911516689288835],
1547: (4)                [0.20785704662965598, 0.068484398797025281, 0.22000133917653536],
1548: (4)                [0.21052882914958676, 0.06812195249816172,  0.22098759107715404],
1549: (4)                [0.2133313859647627,  0.067830148426026665, 0.22207043213024291],
1550: (4)                [0.21625279838647882, 0.067616330270516389, 0.22324568672294431],
1551: (4)                [0.21930503925136402, 0.067465786362940039, 0.22451023616807558],
1552: (4)                [0.22247308588973624, 0.067388214053092838, 0.22585960379408354],
1553: (4)                [0.2257539681670791,  0.067382132300147474, 0.22728984778098055],
1554: (4)                [0.22915620278592841, 0.067434730871152565, 0.22879681433956656],
1555: (4)                [0.23266299920501882, 0.067557104388479783, 0.23037617493752832],
1556: (4)                [0.23627495835774248, 0.06774359820987802,  0.23202360805926608],
1557: (4)                [0.23999586188690308, 0.067985029964779953, 0.23373434258507808],
1558: (4)                [0.24381149720247919, 0.068289851529011875, 0.23550427698321885],
1559: (4)                [0.24772092990501099, 0.068653337909486523, 0.2373288009471749],
1560: (4)                [0.25172899728289466, 0.069064630826035506, 0.23920260612763083],
1561: (4)                [0.25582135547481771, 0.06953231029187984,  0.24112190491594204],
1562: (4)                [0.25999463887892144, 0.070053855603861875, 0.24308218808684579],
1563: (4)                [0.26425512207060942, 0.070616595622995437, 0.24507758869355967],
1564: (4)                [0.26859095948172862, 0.071226716277922458, 0.24710443563450618],
1565: (4)                [0.27299701518897301, 0.071883355446163511, 0.24915847093232929],
1566: (4)                [0.27747150809142801, 0.072582969899254779, 0.25123493995942769],
1567: (4)                [0.28201746297366942, 0.073315693214040967, 0.25332800295084507],
```

```
1568: (4)                [0.28662309235899847, 0.074088460826808866, 0.25543478673717029],
1569: (4)                [0.29128515387578635, 0.074899049847466703, 0.25755101595750435],
1570: (4)                [0.2960004726065818,  0.075745336000958424, 0.25967245030364566],
1571: (4)                [0.30077276812918691, 0.076617824336164764, 0.26179294097819672],
1572: (4)                [0.30559226007249934, 0.077521963107537312, 0.26391006692119662],
1573: (4)                [0.31045520848595526, 0.078456871676182177, 0.2660200572779356],
1574: (4)                [0.31535870009205808, 0.079420997315243186, 0.26811904076941961],
1575: (4)                [0.32029986557994061, 0.080412994737554838, 0.27020322893039511],
1576: (4)                [0.32527888860401261, 0.081428390076546092, 0.27226772884656186],
1577: (4)                [0.33029174471181438, 0.08246763389003825,  0.27430929404579435],
1578: (4)                [0.33533353224455448, 0.083532434119003962, 0.27632534356790039],
1579: (4)                [0.34040164359597463, 0.084622236191702671, 0.27831254595259397],
1580: (4)                [0.34549355713871799, 0.085736654965126335, 0.28026769921081435],
1581: (4)                [0.35060678246032478, 0.08687555176033529,  0.28218770540182386],
1582: (4)                [0.35573889947341125, 0.088038974350243354, 0.2840695897279818],
1583: (4)                [0.36088752387578377, 0.089227194362745205, 0.28591050458531014],
1584: (4)                [0.36605031412464006, 0.090440685427697898, 0.2877077458811747],
1585: (4)                [0.37122508431309342, 0.091679997480262732, 0.28945865397633169],
1586: (4)                [0.3764103053221462,  0.092945198093777909, 0.29116024157313919],
1587: (4)                [0.38160247377467543, 0.094238731263712183, 0.29281107506269488],
1588: (4)                [0.38679939079544168, 0.09556181960083443,  0.29440901248173756],
1589: (4)                [0.39199887556812907, 0.09691583650296684,  0.29595212005509081],
1590: (4)                [0.39719876876325577, 0.098302320968278623, 0.29743856476285779],
1591: (4)                [0.40239692379737496, 0.099722930314950553, 0.29886674369733968],
1592: (4)                [0.40759120392688708, 0.10117945586419633,  0.30023519507728602],
1593: (4)                [0.41277985630360303, 0.1026734006932461,   0.30154226437468967],
1594: (4)                [0.41796105205173684, 0.10420644885760968,  0.30278652039631843],
1595: (4)                [0.42313214269556043, 0.10578120994917611,  0.3039675809469457],
1596: (4)                [0.42829101315789753, 0.1073997763055258,   0.30508479060294547],
1597: (4)                [0.4334355841041439,  0.1090642347484701,   0.30613767928289148],
1598: (4)                [0.43856378187931538, 0.11077667828375456,  0.30712600062348083],
1599: (4)                [0.44367358645071275, 0.11253912421257944,  0.30804973095465449],
1600: (4)                [0.44876299173174822, 0.11435355574622549,  0.30890905921943196],
1601: (4)                [0.45383005086999889, 0.11622183788331528,  0.30970441249844921],
1602: (4)                [0.45887288947308297, 0.11814571137706886,  0.31043636979038808],
1603: (4)                [0.46389102840284874, 0.12012561256850712,  0.31110343446582983],
1604: (4)                [0.46888111384598413, 0.12216445576414045,  0.31170911458932665],
1605: (4)                [0.473841437035254,   0.12426354237989065,  0.31225470169927194],
1606: (4)                [0.47877034239726296, 0.12642401401409453,  0.31274172735821959],
1607: (4)                [0.48366628618847957, 0.12864679022013889,  0.31317188565991266],
1608: (4)                [0.48852847371852987, 0.13093210934893723,  0.31354553695453014],
1609: (4)                [0.49335504375145617, 0.13328091630401023,  0.31386561956734976],
1610: (4)                [0.49814435462074153, 0.13569380302451714,  0.314135190862664],
1611: (4)                [0.50289524974970612, 0.13817086581280427,  0.31435662153833671],
1612: (4)                [0.50760681181053691, 0.14071192654913128,  0.31453200120082569],
1613: (4)                [0.51227835105321762, 0.14331656120063752,  0.3146630922831542],
1614: (4)                [0.51690848800544464, 0.14598463068714407,  0.31475407592280041],
1615: (4)                [0.52149652863229956, 0.14871544765633712,  0.31480767954534428],
1616: (4)                [0.52604189625477482, 0.15150818660835483,  0.31482653406646727],
1617: (4)                [0.53054420489856446, 0.15436183633886777,  0.31481299789187128],
1618: (4)                [0.5350027976174474,  0.15727540775107324,  0.31477085207396532],
1619: (4)                [0.53941736649199057, 0.16024769309971934,  0.31470295028655965],
1620: (4)                [0.54378771313608565, 0.16327738551419116,  0.31461204226295625],
1621: (4)                [0.54811370033467621, 0.1663630904279047,   0.31450102990914708],
1622: (4)                [0.55239521572711914, 0.16950338809328983,  0.31437291554615371],
1623: (4)                [0.55663229034969341, 0.17269677158182117,  0.31423043195101424],
1624: (4)                [0.56082499039117173, 0.17594170887918095,  0.31407639883970623],
1625: (4)                [0.56497343529017696, 0.17923664950367169,  0.3139136046337036],
1626: (4)                [0.56907784784011428, 0.18258004462335425,  0.31374440956796529],
1627: (4)                [0.57313845754107873, 0.18597036007065024,  0.31357126868520002],
1628: (4)                [0.57715550812992045, 0.18940601489760422,  0.31339704333572083],
1629: (4)                [0.58112932761586555, 0.19288548904692518,  0.31322399394183942],
1630: (4)                [0.58506024396466882, 0.19640737049066315,  0.31305401163732732],
1631: (4)                [0.58894861935544707, 0.19997020971775276,  0.31288922211590126],
1632: (4)                [0.59279480536520257, 0.20357251410079796,  0.31273234839304942],
1633: (4)                [0.59659918109122367, 0.207212956082026,    0.31258523031121233],
1634: (4)                [0.60036213010411577, 0.21089030138947745,  0.31244934410414688],
1635: (4)                [0.60408401696732739, 0.21460331490206347,  0.31232652641170694],
1636: (4)                [0.60776523994818654, 0.21835070166659282,  0.31221903291870201],
```

```
1637: (4)                [0.6114062072731884,  0.22213124697023234,  0.31212881396435238],
1638: (4)                [0.61500723236391375, 0.22594402043981826,  0.31205680685765741],
1639: (4)                [0.61856865258877192, 0.22978799249179921,  0.31200463838728931],
1640: (4)                [0.62209079821082613, 0.2336621873300741,   0.31197383273627388],
1641: (4)                [0.62557416500434959, 0.23756535071152696,  0.31196698314912269],
1642: (4)                [0.62901892016985872, 0.24149689191922535,  0.31198447195645718],
1643: (4)                [0.63242534854210275, 0.24545598775548677,  0.31202765974624452],
1644: (4)                [0.6357937104834237,  0.24944185818822678,  0.31209793953300591],
1645: (4)                [0.6391243387840212,  0.25345365461983138,  0.31219689612063978],
1646: (4)                [0.642417577481186,   0.257490519876798,    0.31232631707560987],
1647: (4)                [0.64567349382645434, 0.26155203161615281,  0.31248673753935263],
1648: (4)                [0.64889230169458245, 0.26563755336209077,  0.31267941819570189],
1649: (4)                [0.65207417290277303, 0.26974650525236699,  0.31290560605819168],
1650: (4)                [0.65521932609327127, 0.27387826652410152,  0.3131666792687211],
1651: (4)                [0.6583280801134499,  0.27803210957665631,  0.3134643447952643],
1652: (4)                [0.66140037532601781, 0.28220778870555907,  0.31379912926498488],
1653: (4)                [0.66443632469878844, 0.28640483614256179,  0.31417223403606975],
1654: (4)                [0.66743603766369131, 0.29062280081258873,  0.31458483752056837],
1655: (4)                [0.67039959547676198, 0.29486126309253047,  0.31503813956872212],
1656: (4)                [0.67332725564817331, 0.29911962764489264,  0.31553372323982209],
1657: (4)                [0.67621897924409746, 0.30339762792450425,  0.3160724937230589],
1658: (4)                [0.67907474028157344, 0.30769497879760166,  0.31665545668946665],
1659: (4)                [0.68189457150944521, 0.31201133280550686,  0.31728380489244951],
1660: (4)                [0.68467850942494535, 0.31634634821222207,  0.31795870784057567],
1661: (4)                [0.68742656435169625, 0.32069970535138104,  0.31868137622277692],
1662: (4)                [0.6901389321505248,  0.32507091815606004,  0.31945332332898302],
1663: (4)                [0.69281544846764931, 0.32945984647042675,  0.3202754315314667],
1664: (4)                [0.69545608346891119, 0.33386622163232865,  0.32114884306985791],
1665: (4)                [0.6980608153581771,  0.33828976326048621,  0.32207478855218091],
1666: (4)                [0.70062962477242097, 0.34273019305341756,  0.32305449047765694],
1667: (4)                [0.70316249458814151, 0.34718723719597999,  0.32408913679491225],
1668: (4)                [0.70565951122610093, 0.35166052978120937,  0.32518014084085567],
1669: (4)                [0.70812059568420482, 0.35614985523380299,  0.32632861885644465],
1670: (4)                [0.7105456546582587,  0.36065500290840113,  0.32753574162788762],
1671: (4)                [0.71293466839773467, 0.36517570519856757,  0.3288027427038317],
1672: (4)                [0.71528760614847287, 0.36971170225223449,  0.3301308728723546],
1673: (4)                [0.71760444908133847, 0.37426272710686193,  0.33152138620958932],
1674: (4)                [0.71988521490549851, 0.37882848839337313,  0.33297555200245399],
1675: (4)                [0.7221299918421461,  0.38340864508963057,  0.33449469983585844],
1676: (4)                [0.72433865647781592, 0.38800301593162145,  0.33607995965691828],
1677: (4)                [0.72651122900227549, 0.3926113126792577,   0.3377325942005665],
1678: (4)                [0.72864773856716547, 0.39723324476747235,  0.33945384341064017],
1679: (4)                [0.73074820754845171, 0.401868526884681,    0.3412449533046818],
1680: (4)                [0.73281270506268747, 0.4065168468778026,   0.34310715173410822],
1681: (4)                [0.73484133598564938, 0.41117787004519513,  0.34504169470809071],
1682: (4)                [0.73683422173585866, 0.41585125850290111,  0.34704978520758401],
1683: (4)                [0.73879140024599266, 0.42053672992315327,  0.34913260148542435],
1684: (4)                [0.74071301619506091, 0.4252339389526239,   0.35129130890802607],
1685: (4)                [0.7425992159973317,  0.42994254036133867,  0.35352709245374592],
1686: (4)                [0.74445018676570673, 0.43466217184617112,  0.35584108091122535],
1687: (4)                [0.74626615789163442, 0.43939245044973502,  0.35823439142300639],
1688: (4)                [0.74804739275559562, 0.44413297780351974,  0.36070813602540136],
1689: (4)                [0.74979420547170472, 0.44888333481548809,  0.36326337558360278],
1690: (4)                [0.75150685045891663, 0.45364314496866825,  0.36590112443835765],
1691: (4)                [0.75318566369046569, 0.45841199172949604,  0.36862236642234769],
1692: (4)                [0.75483105066959544, 0.46318942799460555,  0.3714280448394211],
1693: (4)                [0.75644341577140706, 0.46797501437948458,  0.37431909037543515],
1694: (4)                [0.75802325538455839, 0.4727682731566229,   0.37729635531096678],
1695: (4)                [0.75957111105340058, 0.47756871222057079,  0.380360657784311],
1696: (4)                [0.7610876378057071,  0.48237579130289127,  0.38351275723852291],
1697: (4)                [0.76257333554052609, 0.48718906673415824,  0.38675335037837993],
1698: (4)                [0.76402885609288662, 0.49200802533379656,  0.39008308392311997],
1699: (4)                [0.76545492593330511, 0.49683212909727231,  0.39350254000115381],
1700: (4)                [0.76685228950643891, 0.5016608471009063,   0.39701221751773474],
1701: (4)                [0.76822176599735303, 0.50649362371287909,  0.40061257089416885],
1702: (4)                [0.7695642334401418,  0.5113298901696085,   0.40430398069682483],
1703: (4)                [0.77088091962302474, 0.51616892643469103,  0.40808667584648967],
1704: (4)                [0.77217257229605551, 0.5210102658711383,   0.41196089987122869],
1705: (4)                [0.77344021829889886, 0.52585332093451564,  0.41592679539764366],
```

```
1706: (4)                [0.77468494746063199, 0.53069749384776732,  0.41998440356963762],
1707: (4)                [0.77590790730685699, 0.53554217882461186,  0.42413367909988375],
1708: (4)                [0.7771103295521099,  0.54038674910561235,  0.42837450371258479],
1709: (4)                [0.77829345807633121, 0.54523059488426595,  0.432706647838971],
1710: (4)                [0.77945862731506643, 0.55007308413977274,  0.43712979856444761],
1711: (4)                [0.78060774749483774, 0.55491335744890613,  0.44164332426364639],
1712: (4)                [0.78174180478981836, 0.55975098052594863,  0.44624687186865436],
1713: (4)                [0.78286225264440912, 0.56458533111166875,  0.45093985823706345],
1714: (4)                [0.78397060836414478, 0.56941578326710418,  0.45572154742892063],
1715: (4)                [0.78506845019606841, 0.5742417003617839,   0.46059116206904965],
1716: (4)                [0.78615737132332963, 0.5790624629815756,   0.46554778281918402],
1717: (4)                [0.78723904108188347, 0.58387743744557208,  0.47059039582133383],
1718: (4)                [0.78831514045623963, 0.58868600173562435,  0.47571791879076081],
1719: (4)                [0.78938737766251943, 0.5934875421745599,   0.48092913815357724],
1720: (4)                [0.79045776847727878, 0.59828134277062461,  0.48622257801969754],
1721: (4)                [0.79152832843475607, 0.60306670593147205,  0.49159667021646397],
1722: (4)                [0.79260034304237448, 0.60784322087037024,  0.49705020621532009],
1723: (4)                [0.79367559698664958, 0.61261029334072192,  0.50258161291269432],
1724: (4)                [0.79475585972654039, 0.61736734400220705,  0.50818921213102985],
1725: (4)                [0.79584292379583765, 0.62211378808451145,  0.51387124091909786],
1726: (4)                [0.79693854719951607, 0.62684905679296699,  0.5196258425240281],
1727: (4)                [0.79804447815136637, 0.63157258225089552,  0.52545108144834785],
1728: (4)                [0.7991624518501963,  0.63628379372029187,  0.53134495942561433],
1729: (4)                [0.80029415389753977, 0.64098213306749863,  0.53730535185141037],
1730: (4)                [0.80144124292560048, 0.64566703459218766,  0.5433300863249918],
1731: (4)                [0.80260531146112946, 0.65033793748103852,  0.54941691584603647],
1732: (4)                [0.80378792531077625, 0.65499426549472628,  0.55556350867083815],
1733: (4)                [0.80499054790810298, 0.65963545027564163,  0.56176745110546977],
1734: (4)                [0.80621460526927058, 0.66426089585282289,  0.56802629178649788],
1735: (4)                [0.8074614045096935,  0.6688700095398864,   0.57433746373459582],
1736: (4)                [0.80873219170089694, 0.67346216702194517,  0.58069834805576737],
1737: (4)                [0.81002809466520687, 0.67803672673971815,  0.58710626908082753],
1738: (4)                [0.81135014011763329, 0.68259301546243389,  0.59355848909050757],
1739: (4)                [0.81269922039881493, 0.68713033714618876,  0.60005214820435104],
1740: (4)                [0.81407611046993344, 0.69164794791482131,  0.6065843782630862],
1741: (4)                [0.81548146627279483, 0.69614505508308089,  0.61315221209322646],
1742: (4)                [0.81691575775055891, 0.70062083014783982,  0.61975260637257923],
1743: (4)                [0.81837931164498223, 0.70507438189635097,  0.62638245478933297],
1744: (4)                [0.81987230650455289, 0.70950474978787481,  0.63303857040067113],
1745: (4)                [0.8213947205565636,  0.7139109141951604,   0.63971766697672761],
1746: (4)                [0.82294635110428427, 0.71829177331290062,  0.6464164243818421],
1747: (4)                [0.8245268129450285,  0.72264614312088882,  0.65313137915422603],
1748: (4)                [0.82613549710580259, 0.72697275518238258,  0.65985900156216504],
1749: (4)                [0.8277716072353446,  0.73127023324078089,  0.66659570204682972],
1750: (4)                [0.82943407816481474, 0.7355371221572935,   0.67333772009301907],
1751: (4)                [0.83112163529096306, 0.73977184647638616,  0.68008125203631464],
1752: (4)                [0.83283277185777982, 0.74397271817459876,  0.68682235874648545],
1753: (4)                [0.8345656905566583,  0.7481379479992134,   0.69355697649863846],
1754: (4)                [0.83631898844737929, 0.75226548952875261,  0.70027999028864962],
1755: (4)                [0.83809123476131964, 0.75635314860808633,  0.70698561390212977],
1756: (4)                [0.83987839884120874, 0.76039907199779677,  0.71367147811129228],
1757: (4)                [0.84167750766845151, 0.76440101200982946,  0.72033299387284622],
1758: (4)                [0.84348529222933699, 0.76835660399870176,  0.72696536998972039],
1759: (4)                [0.84529810731955113, 0.77226338601044719,  0.73356368240541492],
1760: (4)                [0.84711195507965098, 0.77611880236047159,  0.74012275762807056],
1761: (4)                [0.84892245563117641, 0.77992021407650147,  0.74663719293664366],
1762: (4)                [0.85072697023178789, 0.78366457342383888,  0.7530974636118285],
1763: (4)                [0.85251907207708444, 0.78734936133548439,  0.7594994148789691],
1764: (4)                [0.85429219611470464, 0.79097196777091994,  0.76583801477914104],
1765: (4)                [0.85604022314725403, 0.79452963601550608,  0.77210610037674143],
1766: (4)                [0.85775662943504905, 0.79801963142713928,  0.77829571667247499],
1767: (4)                [0.8594346370300241,  0.8014392309950078,   0.78439788751383921],
1768: (4)                [0.86107117027565516, 0.80478517909812231,  0.79039529663736285],
1769: (4)                [0.86265601051127572, 0.80805523804261525,  0.796282666437655],
1770: (4)                [0.86418343723941027, 0.81124644224653542,  0.80204612696863953],
1771: (4)                [0.86564934325605325, 0.81435544067514909,  0.80766972324164554],
1772: (4)                [0.86705314907048503, 0.81737804041911244,  0.81313419626911398],
1773: (4)                [0.86839954695818633, 0.82030875512181523,  0.81841638963128993],
1774: (4)                [0.86969131502613806, 0.82314158859569164,  0.82350476683173168],
```

```
(4)                [0.87093846717297507, 0.82586857889438514,  0.82838497261149613],
(4)                [0.87215331978454325, 0.82848052823709672,  0.8330486712880828],
(4)                [0.87335171360916275, 0.83096715251272624,  0.83748851001197089],
(4)                [0.87453793320260187, 0.83331972948645461,  0.84171925358069011],
(4)                [0.87571458709961403, 0.8355302318472394,   0.84575537519027078],
(4)                [0.87687848451614692, 0.83759238071186537,  0.84961373549150254],
(4)                [0.87802298436649007, 0.83950165618540074,  0.85330645352458923],
(4)                [0.87913244240792765, 0.84125554884475906,  0.85685572291039636],
(4)                [0.88019293315695812, 0.84285224824778615,  0.86027399927156634],
(4)                [0.88119169871341951, 0.84429066717717349,  0.86356595168669881],
(4)                [0.88211542489401606, 0.84557007254559347,  0.86673765046233331],
(4)                [0.88295168595448525, 0.84668970275699273,  0.86979617048190971],
(4)                [0.88369127145898041, 0.84764891761519268,  0.87274147101441557],
(4)                [0.88432713054113543, 0.84844741572055415,  0.87556785228242973],
(4)                [0.88485138159908572, 0.84908426422893801,  0.87828235285372469],
(4)                [0.88525897972630474, 0.84955892810989209,  0.88088414794024839],
(4)                [0.88554714811952384, 0.84987174283631584,  0.88336206121170946],
(4)                [0.88571155122845646, 0.85002186115856315,  0.88572538990087124]]
(0)            _twilight_shifted_data = (_twilight_data[len(_twilight_data)//2:] +
(26)                                     _twilight_data[:len(_twilight_data)//2])
(0)            _twilight_shifted_data.reverse()
(0)            _turbo_data = [[0.18995, 0.07176, 0.23217],
(15)                          [0.19483, 0.08339, 0.26149],
(15)                          [0.19956, 0.09498, 0.29024],
(15)                          [0.20415, 0.10652, 0.31844],
(15)                          [0.20860, 0.11802, 0.34607],
(15)                          [0.21291, 0.12947, 0.37314],
(15)                          [0.21708, 0.14087, 0.39964],
(15)                          [0.22111, 0.15223, 0.42558],
(15)                          [0.22500, 0.16354, 0.45096],
(15)                          [0.22875, 0.17481, 0.47578],
(15)                          [0.23236, 0.18603, 0.50004],
(15)                          [0.23582, 0.19720, 0.52373],
(15)                          [0.23915, 0.20833, 0.54686],
(15)                          [0.24234, 0.21941, 0.56942],
(15)                          [0.24539, 0.23044, 0.59142],
(15)                          [0.24830, 0.24143, 0.61286],
(15)                          [0.25107, 0.25237, 0.63374],
(15)                          [0.25369, 0.26327, 0.65406],
(15)                          [0.25618, 0.27412, 0.67381],
(15)                          [0.25853, 0.28492, 0.69300],
(15)                          [0.26074, 0.29568, 0.71162],
(15)                          [0.26280, 0.30639, 0.72968],
(15)                          [0.26473, 0.31706, 0.74718],
(15)                          [0.26652, 0.32768, 0.76412],
(15)                          [0.26816, 0.33825, 0.78050],
(15)                          [0.26967, 0.34878, 0.79631],
(15)                          [0.27103, 0.35926, 0.81156],
(15)                          [0.27226, 0.36970, 0.82624],
(15)                          [0.27334, 0.38008, 0.84037],
(15)                          [0.27429, 0.39043, 0.85393],
(15)                          [0.27509, 0.40072, 0.86692],
(15)                          [0.27576, 0.41097, 0.87936],
(15)                          [0.27628, 0.42118, 0.89123],
(15)                          [0.27667, 0.43134, 0.90254],
(15)                          [0.27691, 0.44145, 0.91328],
(15)                          [0.27701, 0.45152, 0.92347],
(15)                          [0.27698, 0.46153, 0.93309],
(15)                          [0.27680, 0.47151, 0.94214],
(15)                          [0.27648, 0.48144, 0.95064],
(15)                          [0.27603, 0.49132, 0.95857],
(15)                          [0.27543, 0.50115, 0.96594],
(15)                          [0.27469, 0.51094, 0.97275],
(15)                          [0.27381, 0.52069, 0.97899],
(15)                          [0.27273, 0.53040, 0.98461],
(15)                          [0.27106, 0.54015, 0.98930],
(15)                          [0.26878, 0.54995, 0.99303],
(15)                          [0.26592, 0.55979, 0.99583],
(15)                          [0.26252, 0.56967, 0.99773],
```

```
1844: (15)                          [0.25862, 0.57958, 0.99876],
1845: (15)                          [0.25425, 0.58950, 0.99896],
1846: (15)                          [0.24946, 0.59943, 0.99835],
1847: (15)                          [0.24427, 0.60937, 0.99697],
1848: (15)                          [0.23874, 0.61931, 0.99485],
1849: (15)                          [0.23288, 0.62923, 0.99202],
1850: (15)                          [0.22676, 0.63913, 0.98851],
1851: (15)                          [0.22039, 0.64901, 0.98436],
1852: (15)                          [0.21382, 0.65886, 0.97959],
1853: (15)                          [0.20708, 0.66866, 0.97423],
1854: (15)                          [0.20021, 0.67842, 0.96833],
1855: (15)                          [0.19326, 0.68812, 0.96190],
1856: (15)                          [0.18625, 0.69775, 0.95498],
1857: (15)                          [0.17923, 0.70732, 0.94761],
1858: (15)                          [0.17223, 0.71680, 0.93981],
1859: (15)                          [0.16529, 0.72620, 0.93161],
1860: (15)                          [0.15844, 0.73551, 0.92305],
1861: (15)                          [0.15173, 0.74472, 0.91416],
1862: (15)                          [0.14519, 0.75381, 0.90496],
1863: (15)                          [0.13886, 0.76279, 0.89550],
1864: (15)                          [0.13278, 0.77165, 0.88580],
1865: (15)                          [0.12698, 0.78037, 0.87590],
1866: (15)                          [0.12151, 0.78896, 0.86581],
1867: (15)                          [0.11639, 0.79740, 0.85559],
1868: (15)                          [0.11167, 0.80569, 0.84525],
1869: (15)                          [0.10738, 0.81381, 0.83484],
1870: (15)                          [0.10357, 0.82177, 0.82437],
1871: (15)                          [0.10026, 0.82955, 0.81389],
1872: (15)                          [0.09750, 0.83714, 0.80342],
1873: (15)                          [0.09532, 0.84455, 0.79299],
1874: (15)                          [0.09377, 0.85175, 0.78264],
1875: (15)                          [0.09287, 0.85875, 0.77240],
1876: (15)                          [0.09267, 0.86554, 0.76230],
1877: (15)                          [0.09320, 0.87211, 0.75237],
1878: (15)                          [0.09451, 0.87844, 0.74265],
1879: (15)                          [0.09662, 0.88454, 0.73316],
1880: (15)                          [0.09958, 0.89040, 0.72393],
1881: (15)                          [0.10342, 0.89600, 0.71500],
1882: (15)                          [0.10815, 0.90142, 0.70599],
1883: (15)                          [0.11374, 0.90673, 0.69651],
1884: (15)                          [0.12014, 0.91193, 0.68660],
1885: (15)                          [0.12733, 0.91701, 0.67627],
1886: (15)                          [0.13526, 0.92197, 0.66556],
1887: (15)                          [0.14391, 0.92680, 0.65448],
1888: (15)                          [0.15323, 0.93151, 0.64308],
1889: (15)                          [0.16319, 0.93609, 0.63137],
1890: (15)                          [0.17377, 0.94053, 0.61938],
1891: (15)                          [0.18491, 0.94484, 0.60713],
1892: (15)                          [0.19659, 0.94901, 0.59466],
1893: (15)                          [0.20877, 0.95304, 0.58199],
1894: (15)                          [0.22142, 0.95692, 0.56914],
1895: (15)                          [0.23449, 0.96065, 0.55614],
1896: (15)                          [0.24797, 0.96423, 0.54303],
1897: (15)                          [0.26180, 0.96765, 0.52981],
1898: (15)                          [0.27597, 0.97092, 0.51653],
1899: (15)                          [0.29042, 0.97403, 0.50321],
1900: (15)                          [0.30513, 0.97697, 0.48987],
1901: (15)                          [0.32006, 0.97974, 0.47654],
1902: (15)                          [0.33517, 0.98234, 0.46325],
1903: (15)                          [0.35043, 0.98477, 0.45002],
1904: (15)                          [0.36581, 0.98702, 0.43688],
1905: (15)                          [0.38127, 0.98909, 0.42386],
1906: (15)                          [0.39678, 0.99098, 0.41098],
1907: (15)                          [0.41229, 0.99268, 0.39826],
1908: (15)                          [0.42778, 0.99419, 0.38575],
1909: (15)                          [0.44321, 0.99551, 0.37345],
1910: (15)                          [0.45854, 0.99663, 0.36140],
1911: (15)                          [0.47375, 0.99755, 0.34963],
1912: (15)                          [0.48879, 0.99828, 0.33816],
```

```
1913: (15)                          [0.50362, 0.99879, 0.32701],
1914: (15)                          [0.51822, 0.99910, 0.31622],
1915: (15)                          [0.53255, 0.99919, 0.30581],
1916: (15)                          [0.54658, 0.99907, 0.29581],
1917: (15)                          [0.56026, 0.99873, 0.28623],
1918: (15)                          [0.57357, 0.99817, 0.27712],
1919: (15)                          [0.58646, 0.99739, 0.26849],
1920: (15)                          [0.59891, 0.99638, 0.26038],
1921: (15)                          [0.61088, 0.99514, 0.25280],
1922: (15)                          [0.62233, 0.99366, 0.24579],
1923: (15)                          [0.63323, 0.99195, 0.23937],
1924: (15)                          [0.64362, 0.98999, 0.23356],
1925: (15)                          [0.65394, 0.98775, 0.22835],
1926: (15)                          [0.66428, 0.98524, 0.22370],
1927: (15)                          [0.67462, 0.98246, 0.21960],
1928: (15)                          [0.68494, 0.97941, 0.21602],
1929: (15)                          [0.69525, 0.97610, 0.21294],
1930: (15)                          [0.70553, 0.97255, 0.21032],
1931: (15)                          [0.71577, 0.96875, 0.20815],
1932: (15)                          [0.72596, 0.96470, 0.20640],
1933: (15)                          [0.73610, 0.96043, 0.20504],
1934: (15)                          [0.74617, 0.95593, 0.20406],
1935: (15)                          [0.75617, 0.95121, 0.20343],
1936: (15)                          [0.76608, 0.94627, 0.20311],
1937: (15)                          [0.77591, 0.94113, 0.20310],
1938: (15)                          [0.78563, 0.93579, 0.20336],
1939: (15)                          [0.79524, 0.93025, 0.20386],
1940: (15)                          [0.80473, 0.92452, 0.20459],
1941: (15)                          [0.81410, 0.91861, 0.20552],
1942: (15)                          [0.82333, 0.91253, 0.20663],
1943: (15)                          [0.83241, 0.90627, 0.20788],
1944: (15)                          [0.84133, 0.89986, 0.20926],
1945: (15)                          [0.85010, 0.89328, 0.21074],
1946: (15)                          [0.85868, 0.88655, 0.21230],
1947: (15)                          [0.86709, 0.87968, 0.21391],
1948: (15)                          [0.87530, 0.87267, 0.21555],
1949: (15)                          [0.88331, 0.86553, 0.21719],
1950: (15)                          [0.89112, 0.85826, 0.21880],
1951: (15)                          [0.89870, 0.85087, 0.22038],
1952: (15)                          [0.90605, 0.84337, 0.22188],
1953: (15)                          [0.91317, 0.83576, 0.22328],
1954: (15)                          [0.92004, 0.82806, 0.22456],
1955: (15)                          [0.92666, 0.82025, 0.22570],
1956: (15)                          [0.93301, 0.81236, 0.22667],
1957: (15)                          [0.93909, 0.80439, 0.22744],
1958: (15)                          [0.94489, 0.79634, 0.22800],
1959: (15)                          [0.95039, 0.78823, 0.22831],
1960: (15)                          [0.95560, 0.78005, 0.22836],
1961: (15)                          [0.96049, 0.77181, 0.22811],
1962: (15)                          [0.96507, 0.76352, 0.22754],
1963: (15)                          [0.96931, 0.75519, 0.22663],
1964: (15)                          [0.97323, 0.74682, 0.22536],
1965: (15)                          [0.97679, 0.73842, 0.22369],
1966: (15)                          [0.98000, 0.73000, 0.22161],
1967: (15)                          [0.98289, 0.72140, 0.21918],
1968: (15)                          [0.98549, 0.71250, 0.21650],
1969: (15)                          [0.98781, 0.70330, 0.21358],
1970: (15)                          [0.98986, 0.69382, 0.21043],
1971: (15)                          [0.99163, 0.68408, 0.20706],
1972: (15)                          [0.99314, 0.67408, 0.20348],
1973: (15)                          [0.99438, 0.66386, 0.19971],
1974: (15)                          [0.99535, 0.65341, 0.19577],
1975: (15)                          [0.99607, 0.64277, 0.19165],
1976: (15)                          [0.99654, 0.63193, 0.18738],
1977: (15)                          [0.99675, 0.62093, 0.18297],
1978: (15)                          [0.99672, 0.60977, 0.17842],
1979: (15)                          [0.99644, 0.59846, 0.17376],
1980: (15)                          [0.99593, 0.58703, 0.16899],
1981: (15)                          [0.99517, 0.57549, 0.16412],
```

```
1982: (15)                          [0.99419, 0.56386, 0.15918],
1983: (15)                          [0.99297, 0.55214, 0.15417],
1984: (15)                          [0.99153, 0.54036, 0.14910],
1985: (15)                          [0.98987, 0.52854, 0.14398],
1986: (15)                          [0.98799, 0.51667, 0.13883],
1987: (15)                          [0.98590, 0.50479, 0.13367],
1988: (15)                          [0.98360, 0.49291, 0.12849],
1989: (15)                          [0.98108, 0.48104, 0.12332],
1990: (15)                          [0.97837, 0.46920, 0.11817],
1991: (15)                          [0.97545, 0.45740, 0.11305],
1992: (15)                          [0.97234, 0.44565, 0.10797],
1993: (15)                          [0.96904, 0.43399, 0.10294],
1994: (15)                          [0.96555, 0.42241, 0.09798],
1995: (15)                          [0.96187, 0.41093, 0.09310],
1996: (15)                          [0.95801, 0.39958, 0.08831],
1997: (15)                          [0.95398, 0.38836, 0.08362],
1998: (15)                          [0.94977, 0.37729, 0.07905],
1999: (15)                          [0.94538, 0.36638, 0.07461],
2000: (15)                          [0.94084, 0.35566, 0.07031],
2001: (15)                          [0.93612, 0.34513, 0.06616],
2002: (15)                          [0.93125, 0.33482, 0.06218],
2003: (15)                          [0.92623, 0.32473, 0.05837],
2004: (15)                          [0.92105, 0.31489, 0.05475],
2005: (15)                          [0.91572, 0.30530, 0.05134],
2006: (15)                          [0.91024, 0.29599, 0.04814],
2007: (15)                          [0.90463, 0.28696, 0.04516],
2008: (15)                          [0.89888, 0.27824, 0.04243],
2009: (15)                          [0.89298, 0.26981, 0.03993],
2010: (15)                          [0.88691, 0.26152, 0.03753],
2011: (15)                          [0.88066, 0.25334, 0.03521],
2012: (15)                          [0.87422, 0.24526, 0.03297],
2013: (15)                          [0.86760, 0.23730, 0.03082],
2014: (15)                          [0.86079, 0.22945, 0.02875],
2015: (15)                          [0.85380, 0.22170, 0.02677],
2016: (15)                          [0.84662, 0.21407, 0.02487],
2017: (15)                          [0.83926, 0.20654, 0.02305],
2018: (15)                          [0.83172, 0.19912, 0.02131],
2019: (15)                          [0.82399, 0.19182, 0.01966],
2020: (15)                          [0.81608, 0.18462, 0.01809],
2021: (15)                          [0.80799, 0.17753, 0.01660],
2022: (15)                          [0.79971, 0.17055, 0.01520],
2023: (15)                          [0.79125, 0.16368, 0.01387],
2024: (15)                          [0.78260, 0.15693, 0.01264],
2025: (15)                          [0.77377, 0.15028, 0.01148],
2026: (15)                          [0.76476, 0.14374, 0.01041],
2027: (15)                          [0.75556, 0.13731, 0.00942],
2028: (15)                          [0.74617, 0.13098, 0.00851],
2029: (15)                          [0.73661, 0.12477, 0.00769],
2030: (15)                          [0.72686, 0.11867, 0.00695],
2031: (15)                          [0.71692, 0.11268, 0.00629],
2032: (15)                          [0.70680, 0.10680, 0.00571],
2033: (15)                          [0.69650, 0.10102, 0.00522],
2034: (15)                          [0.68602, 0.09536, 0.00481],
2035: (15)                          [0.67535, 0.08980, 0.00449],
2036: (15)                          [0.66449, 0.08436, 0.00424],
2037: (15)                          [0.65345, 0.07902, 0.00408],
2038: (15)                          [0.64223, 0.07380, 0.00401],
2039: (15)                          [0.63082, 0.06868, 0.00401],
2040: (15)                          [0.61923, 0.06367, 0.00410],
2041: (15)                          [0.60746, 0.05878, 0.00427],
2042: (15)                          [0.59550, 0.05399, 0.00453],
2043: (15)                          [0.58336, 0.04931, 0.00486],
2044: (15)                          [0.57103, 0.04474, 0.00529],
2045: (15)                          [0.55852, 0.04028, 0.00579],
2046: (15)                          [0.54583, 0.03593, 0.00638],
2047: (15)                          [0.53295, 0.03169, 0.00705],
2048: (15)                          [0.51989, 0.02756, 0.00780],
2049: (15)                          [0.50664, 0.02354, 0.00863],
2050: (15)                          [0.49321, 0.01963, 0.00955],
```

```
2051: (15)                          [0.47960, 0.01583, 0.01055]]
2052: (0)           cmaps = {
2053: (4)               name: ListedColormap(data, name=name) for name, data in [
2054: (8)                   ('magma', _magma_data),
2055: (8)                   ('inferno', _inferno_data),
2056: (8)                   ('plasma', _plasma_data),
2057: (8)                   ('viridis', _viridis_data),
2058: (8)                   ('cividis', _cividis_data),
2059: (8)                   ('twilight', _twilight_data),
2060: (8)                   ('twilight_shifted', _twilight_shifted_data),
2061: (8)                   ('turbo', _turbo_data),
2062: (4)               ]}

----------------------------------------

File 7 - _color_data.py:

1: (0)              BASE_COLORS = {
2: (4)                  'b': (0, 0, 1),        # blue
3: (4)                  'g': (0, 0.5, 0),      # green
4: (4)                  'r': (1, 0, 0),        # red
5: (4)                  'c': (0, 0.75, 0.75),  # cyan
6: (4)                  'm': (0.75, 0, 0.75),  # magenta
7: (4)                  'y': (0.75, 0.75, 0),  # yellow
8: (4)                  'k': (0, 0, 0),        # black
9: (4)                  'w': (1, 1, 1),        # white
10: (0)             }
11: (0)             TABLEAU_COLORS = {
12: (4)                 'tab:blue': '#1f77b4',
13: (4)                 'tab:orange': '#ff7f0e',
14: (4)                 'tab:green': '#2ca02c',
15: (4)                 'tab:red': '#d62728',
16: (4)                 'tab:purple': '#9467bd',
17: (4)                 'tab:brown': '#8c564b',
18: (4)                 'tab:pink': '#e377c2',
19: (4)                 'tab:gray': '#7f7f7f',
20: (4)                 'tab:olive': '#bcbd22',
21: (4)                 'tab:cyan': '#17becf',
22: (0)             }
23: (0)             XKCD_COLORS = {
24: (4)                 'cloudy blue': '#acc2d9',
25: (4)                 'dark pastel green': '#56ae57',
26: (4)                 'dust': '#b2996e',
27: (4)                 'electric lime': '#a8ff04',
28: (4)                 'fresh green': '#69d84f',
29: (4)                 'light eggplant': '#894585',
30: (4)                 'nasty green': '#70b23f',
31: (4)                 'really light blue': '#d4ffff',
32: (4)                 'tea': '#65ab7c',
33: (4)                 'warm purple': '#952e8f',
34: (4)                 'yellowish tan': '#fcfc81',
35: (4)                 'cement': '#a5a391',
36: (4)                 'dark grass green': '#388004',
37: (4)                 'dusty teal': '#4c9085',
38: (4)                 'grey teal': '#5e9b8a',
39: (4)                 'macaroni and cheese': '#efb435',
40: (4)                 'pinkish tan': '#d99b82',
41: (4)                 'spruce': '#0a5f38',
42: (4)                 'strong blue': '#0c06f7',
43: (4)                 'toxic green': '#61de2a',
44: (4)                 'windows blue': '#3778bf',
45: (4)                 'blue blue': '#2242c7',
46: (4)                 'blue with a hint of purple': '#533cc6',
47: (4)                 'booger': '#9bb53c',
48: (4)                 'bright sea green': '#05ffa6',
49: (4)                 'dark green blue': '#1f6357',
50: (4)                 'deep turquoise': '#017374',
51: (4)                 'green teal': '#0cb577',
52: (4)                 'strong pink': '#ff0789',
```

```
53: (4)                 'bland': '#afa88b',
54: (4)                 'deep aqua': '#08787f',
55: (4)                 'lavender pink': '#dd85d7',
56: (4)                 'light moss green': '#a6c875',
57: (4)                 'light seafoam green': '#a7ffb5',
58: (4)                 'olive yellow': '#c2b709',
59: (4)                 'pig pink': '#e78ea5',
60: (4)                 'deep lilac': '#966ebd',
61: (4)                 'desert': '#ccad60',
62: (4)                 'dusty lavender': '#ac86a8',
63: (4)                 'purpley grey': '#947e94',
64: (4)                 'purply': '#983fb2',
65: (4)                 'candy pink': '#ff63e9',
66: (4)                 'light pastel green': '#b2fba5',
67: (4)                 'boring green': '#63b365',
68: (4)                 'kiwi green': '#8ee53f',
69: (4)                 'light grey green': '#b7e1a1',
70: (4)                 'orange pink': '#ff6f52',
71: (4)                 'tea green': '#bdf8a3',
72: (4)                 'very light brown': '#d3b683',
73: (4)                 'egg shell': '#fffcc4',
74: (4)                 'eggplant purple': '#430541',
75: (4)                 'powder pink': '#ffb2d0',
76: (4)                 'reddish grey': '#997570',
77: (4)                 'baby shit brown': '#ad900d',
78: (4)                 'liliac': '#c48efd',
79: (4)                 'stormy blue': '#507b9c',
80: (4)                 'ugly brown': '#7d7103',
81: (4)                 'custard': '#fffd78',
82: (4)                 'darkish pink': '#da467d',
83: (4)                 'deep brown': '#410200',
84: (4)                 'greenish beige': '#c9d179',
85: (4)                 'manilla': '#fffa86',
86: (4)                 'off blue': '#5684ae',
87: (4)                 'battleship grey': '#6b7c85',
88: (4)                 'browny green': '#6f6c0a',
89: (4)                 'bruise': '#7e4071',
90: (4)                 'kelley green': '#009337',
91: (4)                 'sickly yellow': '#d0e429',
92: (4)                 'sunny yellow': '#fff917',
93: (4)                 'azul': '#1d5dec',
94: (4)                 'darkgreen': '#054907',
95: (4)                 'green/yellow': '#b5ce08',
96: (4)                 'lichen': '#8fb67b',
97: (4)                 'light light green': '#c8ffb0',
98: (4)                 'pale gold': '#fdde6c',
99: (4)                 'sun yellow': '#ffdf22',
100: (4)                'tan green': '#a9be70',
101: (4)                'burple': '#6832e3',
102: (4)                'butterscotch': '#fdb147',
103: (4)                'toupe': '#c7ac7d',
104: (4)                'dark cream': '#fff39a',
105: (4)                'indian red': '#850e04',
106: (4)                'light lavendar': '#efc0fe',
107: (4)                'poison green': '#40fd14',
108: (4)                'baby puke green': '#b6c406',
109: (4)                'bright yellow green': '#9dff00',
110: (4)                'charcoal grey': '#3c4142',
111: (4)                'squash': '#f2ab15',
112: (4)                'cinnamon': '#ac4f06',
113: (4)                'light pea green': '#c4fe82',
114: (4)                'radioactive green': '#2cfa1f',
115: (4)                'raw sienna': '#9a6200',
116: (4)                'baby purple': '#ca9bf7',
117: (4)                'cocoa': '#875f42',
118: (4)                'light royal blue': '#3a2efe',
119: (4)                'orangeish': '#fd8d49',
120: (4)                'rust brown': '#8b3103',
121: (4)                'sand brown': '#cba560',
```

```
(4)                 'swamp': '#698339',
(4)                 'tealish green': '#0cdc73',
(4)                 'burnt siena': '#b75203',
(4)                 'camo': '#7f8f4e',
(4)                 'dusk blue': '#26538d',
(4)                 'fern': '#63a950',
(4)                 'old rose': '#c87f89',
(4)                 'pale light green': '#b1fc99',
(4)                 'peachy pink': '#ff9a8a',
(4)                 'rosy pink': '#f6688e',
(4)                 'light bluish green': '#76fda8',
(4)                 'light bright green': '#53fe5c',
(4)                 'light neon green': '#4efd54',
(4)                 'light seafoam': '#a0febf',
(4)                 'tiffany blue': '#7bf2da',
(4)                 'washed out green': '#bcf5a6',
(4)                 'browny orange': '#ca6b02',
(4)                 'nice blue': '#107ab0',
(4)                 'sapphire': '#2138ab',
(4)                 'greyish teal': '#719f91',
(4)                 'orangey yellow': '#fdb915',
(4)                 'parchment': '#fefcaf',
(4)                 'straw': '#fcf679',
(4)                 'very dark brown': '#1d0200',
(4)                 'terracota': '#cb6843',
(4)                 'ugly blue': '#31668a',
(4)                 'clear blue': '#247afd',
(4)                 'creme': '#ffffb6',
(4)                 'foam green': '#90fda9',
(4)                 'grey/green': '#86a17d',
(4)                 'light gold': '#fddc5c',
(4)                 'seafoam blue': '#78d1b6',
(4)                 'topaz': '#13bbaf',
(4)                 'violet pink': '#fb5ffc',
(4)                 'wintergreen': '#20f986',
(4)                 'yellow tan': '#ffe36e',
(4)                 'dark fuchsia': '#9d0759',
(4)                 'indigo blue': '#3a18b1',
(4)                 'light yellowish green': '#c2ff89',
(4)                 'pale magenta': '#d767ad',
(4)                 'rich purple': '#720058',
(4)                 'sunflower yellow': '#ffda03',
(4)                 'green/blue': '#01c08d',
(4)                 'leather': '#ac7434',
(4)                 'racing green': '#014600',
(4)                 'vivid purple': '#9900fa',
(4)                 'dark royal blue': '#02066f',
(4)                 'hazel': '#8e7618',
(4)                 'muted pink': '#d1768f',
(4)                 'booger green': '#96b403',
(4)                 'canary': '#fdff63',
(4)                 'cool grey': '#95a3a6',
(4)                 'dark taupe': '#7f684e',
(4)                 'darkish purple': '#751973',
(4)                 'true green': '#089404',
(4)                 'coral pink': '#ff6163',
(4)                 'dark sage': '#598556',
(4)                 'dark slate blue': '#214761',
(4)                 'flat blue': '#3c73a8',
(4)                 'mushroom': '#ba9e88',
(4)                 'rich blue': '#021bf9',
(4)                 'dirty purple': '#734a65',
(4)                 'greenblue': '#23c48b',
(4)                 'icky green': '#8fae22',
(4)                 'light khaki': '#e6f2a2',
(4)                 'warm blue': '#4b57db',
(4)                 'dark hot pink': '#d90166',
(4)                 'deep sea blue': '#015482',
(4)                 'carmine': '#9d0216',
```

```
(4)                 'dark yellow green': '#728f02',
(4)                 'pale peach': '#ffe5ad',
(4)                 'plum purple': '#4e0550',
(4)                 'golden rod': '#f9bc08',
(4)                 'neon red': '#ff073a',
(4)                 'old pink': '#c77986',
(4)                 'very pale blue': '#d6fffe',
(4)                 'blood orange': '#fe4b03',
(4)                 'grapefruit': '#fd5956',
(4)                 'sand yellow': '#fce166',
(4)                 'clay brown': '#b2713d',
(4)                 'dark blue grey': '#1f3b4d',
(4)                 'flat green': '#699d4c',
(4)                 'light green blue': '#56fca2',
(4)                 'warm pink': '#fb5581',
(4)                 'dodger blue': '#3e82fc',
(4)                 'gross green': '#a0bf16',
(4)                 'ice': '#d6fffa',
(4)                 'metallic blue': '#4f738e',
(4)                 'pale salmon': '#ffb19a',
(4)                 'sap green': '#5c8b15',
(4)                 'algae': '#54ac68',
(4)                 'bluey grey': '#89a0b0',
(4)                 'greeny grey': '#7ea07a',
(4)                 'highlighter green': '#1bfc06',
(4)                 'light light blue': '#cafffb',
(4)                 'light mint': '#b6ffbb',
(4)                 'raw umber': '#a75e09',
(4)                 'vivid blue': '#152eff',
(4)                 'deep lavender': '#8d5eb7',
(4)                 'dull teal': '#5f9e8f',
(4)                 'light greenish blue': '#63f7b4',
(4)                 'mud green': '#606602',
(4)                 'pinky': '#fc86aa',
(4)                 'red wine': '#8c0034',
(4)                 'shit green': '#758000',
(4)                 'tan brown': '#ab7e4c',
(4)                 'darkblue': '#030764',
(4)                 'rosa': '#fe86a4',
(4)                 'lipstick': '#d5174e',
(4)                 'pale mauve': '#fed0fc',
(4)                 'claret': '#680018',
(4)                 'dandelion': '#fedf08',
(4)                 'orangered': '#fe420f',
(4)                 'poop green': '#6f7c00',
(4)                 'ruby': '#ca0147',
(4)                 'dark': '#1b2431',
(4)                 'greenish turquoise': '#00fbb0',
(4)                 'pastel red': '#db5856',
(4)                 'piss yellow': '#ddd618',
(4)                 'bright cyan': '#41fdfe',
(4)                 'dark coral': '#cf524e',
(4)                 'algae green': '#21c36f',
(4)                 'darkish red': '#a90308',
(4)                 'reddy brown': '#6e1005',
(4)                 'blush pink': '#fe828c',
(4)                 'camouflage green': '#4b6113',
(4)                 'lawn green': '#4da409',
(4)                 'putty': '#beae8a',
(4)                 'vibrant blue': '#0339f8',
(4)                 'dark sand': '#a88f59',
(4)                 'purple/blue': '#5d21d0',
(4)                 'saffron': '#feb209',
(4)                 'twilight': '#4e518b',
(4)                 'warm brown': '#964e02',
(4)                 'bluegrey': '#85a3b2',
(4)                 'bubble gum pink': '#ff69af',
(4)                 'duck egg blue': '#c3fbf4',
(4)                 'greenish cyan': '#2afeb7',
```

```
(4)                 'petrol': '#005f6a',
(4)                 'royal': '#0c1793',
(4)                 'butter': '#ffff81',
(4)                 'dusty orange': '#f0833a',
(4)                 'off yellow': '#f1f33f',
(4)                 'pale olive green': '#b1d27b',
(4)                 'orangish': '#fc824a',
(4)                 'leaf': '#71aa34',
(4)                 'light blue grey': '#b7c9e2',
(4)                 'dried blood': '#4b0101',
(4)                 'lightish purple': '#a552e6',
(4)                 'rusty red': '#af2f0d',
(4)                 'lavender blue': '#8b88f8',
(4)                 'light grass green': '#9af764',
(4)                 'light mint green': '#a6fbb2',
(4)                 'sunflower': '#ffc512',
(4)                 'velvet': '#750851',
(4)                 'brick orange': '#c14a09',
(4)                 'lightish red': '#fe2f4a',
(4)                 'pure blue': '#0203e2',
(4)                 'twilight blue': '#0a437a',
(4)                 'violet red': '#a50055',
(4)                 'yellowy brown': '#ae8b0c',
(4)                 'carnation': '#fd798f',
(4)                 'muddy yellow': '#bfac05',
(4)                 'dark seafoam green': '#3eaf76',
(4)                 'deep rose': '#c74767',
(4)                 'dusty red': '#b9484e',
(4)                 'grey/blue': '#647d8e',
(4)                 'lemon lime': '#bffe28',
(4)                 'purple/pink': '#d725de',
(4)                 'brown yellow': '#b29705',
(4)                 'purple brown': '#673a3f',
(4)                 'wisteria': '#a87dc2',
(4)                 'banana yellow': '#fafe4b',
(4)                 'lipstick red': '#c0022f',
(4)                 'water blue': '#0e87cc',
(4)                 'brown grey': '#8d8468',
(4)                 'vibrant purple': '#ad03de',
(4)                 'baby green': '#8cff9e',
(4)                 'barf green': '#94ac02',
(4)                 'eggshell blue': '#c4fff7',
(4)                 'sandy yellow': '#fdee73',
(4)                 'cool green': '#33b864',
(4)                 'pale': '#fff9d0',
(4)                 'blue/grey': '#758da3',
(4)                 'hot magenta': '#f504c9',
(4)                 'greyblue': '#77a1b5',
(4)                 'purpley': '#8756e4',
(4)                 'baby shit green': '#889717',
(4)                 'brownish pink': '#c27e79',
(4)                 'dark aquamarine': '#017371',
(4)                 'diarrhea': '#9f8303',
(4)                 'light mustard': '#f7d560',
(4)                 'pale sky blue': '#bdf6fe',
(4)                 'turtle green': '#75b84f',
(4)                 'bright olive': '#9cbb04',
(4)                 'dark grey blue': '#29465b',
(4)                 'greeny brown': '#696006',
(4)                 'lemon green': '#adf802',
(4)                 'light periwinkle': '#c1c6fc',
(4)                 'seaweed green': '#35ad6b',
(4)                 'sunshine yellow': '#fffd37',
(4)                 'ugly purple': '#a442a0',
(4)                 'medium pink': '#f36196',
(4)                 'puke brown': '#947706',
(4)                 'very light pink': '#fff4f2',
(4)                 'viridian': '#1e9167',
(4)                 'bile': '#b5c306',
```

```
(4)                 'faded yellow': '#feff7f',
(4)                 'very pale green': '#cffdbc',
(4)                 'vibrant green': '#0add08',
(4)                 'bright lime': '#87fd05',
(4)                 'spearmint': '#1ef876',
(4)                 'light aquamarine': '#7bfdc7',
(4)                 'light sage': '#bcecac',
(4)                 'yellowgreen': '#bbf90f',
(4)                 'baby poo': '#ab9004',
(4)                 'dark seafoam': '#1fb57a',
(4)                 'deep teal': '#00555a',
(4)                 'heather': '#a484ac',
(4)                 'rust orange': '#c45508',
(4)                 'dirty blue': '#3f829d',
(4)                 'fern green': '#548d44',
(4)                 'bright lilac': '#c95efb',
(4)                 'weird green': '#3ae57f',
(4)                 'peacock blue': '#016795',
(4)                 'avocado green': '#87a922',
(4)                 'faded orange': '#f0944d',
(4)                 'grape purple': '#5d1451',
(4)                 'hot green': '#25ff29',
(4)                 'lime yellow': '#d0fe1d',
(4)                 'mango': '#ffa62b',
(4)                 'shamrock': '#01b44c',
(4)                 'bubblegum': '#ff6cb5',
(4)                 'purplish brown': '#6b4247',
(4)                 'vomit yellow': '#c7c10c',
(4)                 'pale cyan': '#b7fffa',
(4)                 'key lime': '#aeff6e',
(4)                 'tomato red': '#ec2d01',
(4)                 'lightgreen': '#76ff7b',
(4)                 'merlot': '#730039',
(4)                 'night blue': '#040348',
(4)                 'purpleish pink': '#df4ec8',
(4)                 'apple': '#6ecb3c',
(4)                 'baby poop green': '#8f9805',
(4)                 'green apple': '#5edc1f',
(4)                 'heliotrope': '#d94ff5',
(4)                 'yellow/green': '#c8fd3d',
(4)                 'almost black': '#070d0d',
(4)                 'cool blue': '#4984b8',
(4)                 'leafy green': '#51b73b',
(4)                 'mustard brown': '#ac7e04',
(4)                 'dusk': '#4e5481',
(4)                 'dull brown': '#876e4b',
(4)                 'frog green': '#58bc08',
(4)                 'vivid green': '#2fef10',
(4)                 'bright light green': '#2dfe54',
(4)                 'fluro green': '#0aff02',
(4)                 'kiwi': '#9cef43',
(4)                 'seaweed': '#18d17b',
(4)                 'navy green': '#35530a',
(4)                 'ultramarine blue': '#1805db',
(4)                 'iris': '#6258c4',
(4)                 'pastel orange': '#ff964f',
(4)                 'yellowish orange': '#ffab0f',
(4)                 'perrywinkle': '#8f8ce7',
(4)                 'tealish': '#24bca8',
(4)                 'dark plum': '#3f012c',
(4)                 'pear': '#cbf85f',
(4)                 'pinkish orange': '#ff724c',
(4)                 'midnight purple': '#280137',
(4)                 'light urple': '#b36ff6',
(4)                 'dark mint': '#48c072',
(4)                 'greenish tan': '#bccb7a',
(4)                 'light burgundy': '#a8415b',
(4)                 'turquoise blue': '#06b1c4',
(4)                 'ugly pink': '#cd7584',
```

```
(4)                 'sandy': '#f1da7a',
(4)                 'electric pink': '#ff0490',
(4)                 'muted purple': '#805b87',
(4)                 'mid green': '#50a747',
(4)                 'greyish': '#a8a495',
(4)                 'neon yellow': '#cfff04',
(4)                 'banana': '#ffff7e',
(4)                 'carnation pink': '#ff7fa7',
(4)                 'tomato': '#ef4026',
(4)                 'sea': '#3c9992',
(4)                 'muddy brown': '#886806',
(4)                 'turquoise green': '#04f489',
(4)                 'buff': '#fef69e',
(4)                 'fawn': '#cfaf7b',
(4)                 'muted blue': '#3b719f',
(4)                 'pale rose': '#fdc1c5',
(4)                 'dark mint green': '#20c073',
(4)                 'amethyst': '#9b5fc0',
(4)                 'blue/green': '#0f9b8e',
(4)                 'chestnut': '#742802',
(4)                 'sick green': '#9db92c',
(4)                 'pea': '#a4bf20',
(4)                 'rusty orange': '#cd5909',
(4)                 'stone': '#ada587',
(4)                 'rose red': '#be013c',
(4)                 'pale aqua': '#b8ffeb',
(4)                 'deep orange': '#dc4d01',
(4)                 'earth': '#a2653e',
(4)                 'mossy green': '#638b27',
(4)                 'grassy green': '#419c03',
(4)                 'pale lime green': '#b1ff65',
(4)                 'light grey blue': '#9dbcd4',
(4)                 'pale grey': '#fdfdfe',
(4)                 'asparagus': '#77ab56',
(4)                 'blueberry': '#464196',
(4)                 'purple red': '#990147',
(4)                 'pale lime': '#befd73',
(4)                 'greenish teal': '#32bf84',
(4)                 'caramel': '#af6f09',
(4)                 'deep magenta': '#a0025c',
(4)                 'light peach': '#ffd8b1',
(4)                 'milk chocolate': '#7f4e1e',
(4)                 'ocher': '#bf9b0c',
(4)                 'off green': '#6ba353',
(4)                 'purply pink': '#f075e6',
(4)                 'lightblue': '#7bc8f6',
(4)                 'dusky blue': '#475f94',
(4)                 'golden': '#f5bf03',
(4)                 'light beige': '#fffeb6',
(4)                 'butter yellow': '#fffd74',
(4)                 'dusky purple': '#895b7b',
(4)                 'french blue': '#436bad',
(4)                 'ugly yellow': '#d0c101',
(4)                 'greeny yellow': '#c6f808',
(4)                 'orangish red': '#f43605',
(4)                 'shamrock green': '#02c14d',
(4)                 'orangish brown': '#b25f03',
(4)                 'tree green': '#2a7e19',
(4)                 'deep violet': '#490648',
(4)                 'gunmetal': '#536267',
(4)                 'blue/purple': '#5a06ef',
(4)                 'cherry': '#cf0234',
(4)                 'sandy brown': '#c4a661',
(4)                 'warm grey': '#978a84',
(4)                 'dark indigo': '#1f0954',
(4)                 'midnight': '#03012d',
(4)                 'bluey green': '#2bb179',
(4)                 'grey pink': '#c3909b',
(4)                 'soft purple': '#a66fb5',
```

```
(4)                 'blood': '#770001',
(4)                 'brown red': '#922b05',
(4)                 'medium grey': '#7d7f7c',
(4)                 'berry': '#990f4b',
(4)                 'poo': '#8f7303',
(4)                 'purpley pink': '#c83cb9',
(4)                 'light salmon': '#fea993',
(4)                 'snot': '#acbb0d',
(4)                 'easter purple': '#c071fe',
(4)                 'light yellow green': '#ccfd7f',
(4)                 'dark navy blue': '#00022e',
(4)                 'drab': '#828344',
(4)                 'light rose': '#ffc5cb',
(4)                 'rouge': '#ab1239',
(4)                 'purplish red': '#b0054b',
(4)                 'slime green': '#99cc04',
(4)                 'baby poop': '#937c00',
(4)                 'irish green': '#019529',
(4)                 'pink/purple': '#ef1de7',
(4)                 'dark navy': '#000435',
(4)                 'greeny blue': '#42b395',
(4)                 'light plum': '#9d5783',
(4)                 'pinkish grey': '#c8aca9',
(4)                 'dirty orange': '#c87606',
(4)                 'rust red': '#aa2704',
(4)                 'pale lilac': '#e4cbff',
(4)                 'orangey red': '#fa4224',
(4)                 'primary blue': '#0804f9',
(4)                 'kermit green': '#5cb200',
(4)                 'brownish purple': '#76424e',
(4)                 'murky green': '#6c7a0e',
(4)                 'wheat': '#fbdd7e',
(4)                 'very dark purple': '#2a0134',
(4)                 'bottle green': '#044a05',
(4)                 'watermelon': '#fd4659',
(4)                 'deep sky blue': '#0d75f8',
(4)                 'fire engine red': '#fe0002',
(4)                 'yellow ochre': '#cb9d06',
(4)                 'pumpkin orange': '#fb7d07',
(4)                 'pale olive': '#b9cc81',
(4)                 'light lilac': '#edc8ff',
(4)                 'lightish green': '#61e160',
(4)                 'carolina blue': '#8ab8fe',
(4)                 'mulberry': '#920a4e',
(4)                 'shocking pink': '#fe02a2',
(4)                 'auburn': '#9a3001',
(4)                 'bright lime green': '#65fe08',
(4)                 'celadon': '#befdb7',
(4)                 'pinkish brown': '#b17261',
(4)                 'poo brown': '#885f01',
(4)                 'bright sky blue': '#02ccfe',
(4)                 'celery': '#c1fd95',
(4)                 'dirt brown': '#836539',
(4)                 'strawberry': '#fb2943',
(4)                 'dark lime': '#84b701',
(4)                 'copper': '#b66325',
(4)                 'medium brown': '#7f5112',
(4)                 'muted green': '#5fa052',
(4)                 "robin's egg": '#6dedfd',
(4)                 'bright aqua': '#0bf9ea',
(4)                 'bright lavender': '#c760ff',
(4)                 'ivory': '#ffffcb',
(4)                 'very light purple': '#f6cefc',
(4)                 'light navy': '#155084',
(4)                 'pink red': '#f5054f',
(4)                 'olive brown': '#645403',
(4)                 'poop brown': '#7a5901',
(4)                 'mustard green': '#a8b504',
(4)                 'ocean green': '#3d9973',
```

```
(4)                 'very dark blue': '#000133',
(4)                 'dusty green': '#76a973',
(4)                 'light navy blue': '#2e5a88',
(4)                 'minty green': '#0bf77d',
(4)                 'adobe': '#bd6c48',
(4)                 'barney': '#ac1db8',
(4)                 'jade green': '#2baf6a',
(4)                 'bright light blue': '#26f7fd',
(4)                 'light lime': '#aefd6c',
(4)                 'dark khaki': '#9b8f55',
(4)                 'orange yellow': '#ffad01',
(4)                 'ocre': '#c69c04',
(4)                 'maize': '#f4d054',
(4)                 'faded pink': '#de9dac',
(4)                 'british racing green': '#05480d',
(4)                 'sandstone': '#c9ae74',
(4)                 'mud brown': '#60460f',
(4)                 'light sea green': '#98f6b0',
(4)                 'robin egg blue': '#8af1fe',
(4)                 'aqua marine': '#2ee8bb',
(4)                 'dark sea green': '#11875d',
(4)                 'soft pink': '#fdb0c0',
(4)                 'orangey brown': '#b16002',
(4)                 'cherry red': '#f7022a',
(4)                 'burnt yellow': '#d5ab09',
(4)                 'brownish grey': '#86775f',
(4)                 'camel': '#c69f59',
(4)                 'purplish grey': '#7a687f',
(4)                 'marine': '#042e60',
(4)                 'greyish pink': '#c88d94',
(4)                 'pale turquoise': '#a5fbd5',
(4)                 'pastel yellow': '#fffe71',
(4)                 'bluey purple': '#6241c7',
(4)                 'canary yellow': '#fffe40',
(4)                 'faded red': '#d3494e',
(4)                 'sepia': '#985e2b',
(4)                 'coffee': '#a6814c',
(4)                 'bright magenta': '#ff08e8',
(4)                 'mocha': '#9d7651',
(4)                 'ecru': '#feffca',
(4)                 'purpleish': '#98568d',
(4)                 'cranberry': '#9e003a',
(4)                 'darkish green': '#287c37',
(4)                 'brown orange': '#b96902',
(4)                 'dusky rose': '#ba6873',
(4)                 'melon': '#ff7855',
(4)                 'sickly green': '#94b21c',
(4)                 'silver': '#c5c9c7',
(4)                 'purply blue': '#661aee',
(4)                 'purpleish blue': '#6140ef',
(4)                 'hospital green': '#9be5aa',
(4)                 'shit brown': '#7b5804',
(4)                 'mid blue': '#276ab3',
(4)                 'amber': '#feb308',
(4)                 'easter green': '#8cfd7e',
(4)                 'soft blue': '#6488ea',
(4)                 'cerulean blue': '#056eee',
(4)                 'golden brown': '#b27a01',
(4)                 'bright turquoise': '#0ffef9',
(4)                 'red pink': '#fa2a55',
(4)                 'red purple': '#820747',
(4)                 'greyish brown': '#7a6a4f',
(4)                 'vermillion': '#f4320c',
(4)                 'russet': '#a13905',
(4)                 'steel grey': '#6f828a',
(4)                 'lighter purple': '#a55af4',
(4)                 'bright violet': '#ad0afd',
(4)                 'prussian blue': '#004577',
(4)                 'slate green': '#658d6d',
```

```
(4)                 'dirty pink': '#ca7b80',
(4)                 'dark blue green': '#005249',
(4)                 'pine': '#2b5d34',
(4)                 'yellowy green': '#bff128',
(4)                 'dark gold': '#b59410',
(4)                 'bluish': '#2976bb',
(4)                 'darkish blue': '#014182',
(4)                 'dull red': '#bb3f3f',
(4)                 'pinky red': '#fc2647',
(4)                 'bronze': '#a87900',
(4)                 'pale teal': '#82cbb2',
(4)                 'military green': '#667c3e',
(4)                 'barbie pink': '#fe46a5',
(4)                 'bubblegum pink': '#fe83cc',
(4)                 'pea soup green': '#94a617',
(4)                 'dark mustard': '#a88905',
(4)                 'shit': '#7f5f00',
(4)                 'medium purple': '#9e43a2',
(4)                 'very dark green': '#062e03',
(4)                 'dirt': '#8a6e45',
(4)                 'dusky pink': '#cc7a8b',
(4)                 'red violet': '#9e0168',
(4)                 'lemon yellow': '#fdff38',
(4)                 'pistachio': '#c0fa8b',
(4)                 'dull yellow': '#eedc5b',
(4)                 'dark lime green': '#7ebd01',
(4)                 'denim blue': '#3b5b92',
(4)                 'teal blue': '#01889f',
(4)                 'lightish blue': '#3d7afd',
(4)                 'purpley blue': '#5f34e7',
(4)                 'light indigo': '#6d5acf',
(4)                 'swamp green': '#748500',
(4)                 'brown green': '#706c11',
(4)                 'dark maroon': '#3c0008',
(4)                 'hot purple': '#cb00f5',
(4)                 'dark forest green': '#002d04',
(4)                 'faded blue': '#658cbb',
(4)                 'drab green': '#749551',
(4)                 'light lime green': '#b9ff66',
(4)                 'snot green': '#9dc100',
(4)                 'yellowish': '#faee66',
(4)                 'light blue green': '#7efbb3',
(4)                 'bordeaux': '#7b002c',
(4)                 'light mauve': '#c292a1',
(4)                 'ocean': '#017b92',
(4)                 'marigold': '#fcc006',
(4)                 'muddy green': '#657432',
(4)                 'dull orange': '#d8863b',
(4)                 'steel': '#738595',
(4)                 'electric purple': '#aa23ff',
(4)                 'fluorescent green': '#08ff08',
(4)                 'yellowish brown': '#9b7a01',
(4)                 'blush': '#f29e8e',
(4)                 'soft green': '#6fc276',
(4)                 'bright orange': '#ff5b00',
(4)                 'lemon': '#fdff52',
(4)                 'purple grey': '#866f85',
(4)                 'acid green': '#8ffe09',
(4)                 'pale lavender': '#eecffe',
(4)                 'violet blue': '#510ac9',
(4)                 'light forest green': '#4f9153',
(4)                 'burnt red': '#9f2305',
(4)                 'khaki green': '#728639',
(4)                 'cerise': '#de0c62',
(4)                 'faded purple': '#916e99',
(4)                 'apricot': '#ffb16d',
(4)                 'dark olive green': '#3c4d03',
(4)                 'grey brown': '#7f7053',
(4)                 'green grey': '#77926f',
```

```
(4)                 'true blue': '#010fcc',
(4)                 'pale violet': '#ceaefa',
(4)                 'periwinkle blue': '#8f99fb',
(4)                 'light sky blue': '#c6fcff',
(4)                 'blurple': '#5539cc',
(4)                 'green brown': '#544e03',
(4)                 'bluegreen': '#017a79',
(4)                 'bright teal': '#01f9c6',
(4)                 'brownish yellow': '#c9b003',
(4)                 'pea soup': '#929901',
(4)                 'forest': '#0b5509',
(4)                 'barney purple': '#a00498',
(4)                 'ultramarine': '#2000b1',
(4)                 'purplish': '#94568c',
(4)                 'puke yellow': '#c2be0e',
(4)                 'bluish grey': '#748b97',
(4)                 'dark periwinkle': '#665fd1',
(4)                 'dark lilac': '#9c6da5',
(4)                 'reddish': '#c44240',
(4)                 'light maroon': '#a24857',
(4)                 'dusty purple': '#825f87',
(4)                 'terra cotta': '#c9643b',
(4)                 'avocado': '#90b134',
(4)                 'marine blue': '#01386a',
(4)                 'teal green': '#25a36f',
(4)                 'slate grey': '#59656d',
(4)                 'lighter green': '#75fd63',
(4)                 'electric green': '#21fc0d',
(4)                 'dusty blue': '#5a86ad',
(4)                 'golden yellow': '#fec615',
(4)                 'bright yellow': '#fffd01',
(4)                 'light lavender': '#dfc5fe',
(4)                 'umber': '#b26400',
(4)                 'poop': '#7f5e00',
(4)                 'dark peach': '#de7e5d',
(4)                 'jungle green': '#048243',
(4)                 'eggshell': '#ffffd4',
(4)                 'denim': '#3b638c',
(4)                 'yellow brown': '#b79400',
(4)                 'dull purple': '#84597e',
(4)                 'chocolate brown': '#411900',
(4)                 'wine red': '#7b0323',
(4)                 'neon blue': '#04d9ff',
(4)                 'dirty green': '#667e2c',
(4)                 'light tan': '#fbeeac',
(4)                 'ice blue': '#d7fffe',
(4)                 'cadet blue': '#4e7496',
(4)                 'dark mauve': '#874c62',
(4)                 'very light blue': '#d5ffff',
(4)                 'grey purple': '#826d8c',
(4)                 'pastel pink': '#ffbacd',
(4)                 'very light green': '#d1ffbd',
(4)                 'dark sky blue': '#448ee4',
(4)                 'evergreen': '#05472a',
(4)                 'dull pink': '#d5869d',
(4)                 'aubergine': '#3d0734',
(4)                 'mahogany': '#4a0100',
(4)                 'reddish orange': '#f8481c',
(4)                 'deep green': '#02590f',
(4)                 'vomit green': '#89a203',
(4)                 'purple pink': '#e03fd8',
(4)                 'dusty pink': '#d58a94',
(4)                 'faded green': '#7bb274',
(4)                 'camo green': '#526525',
(4)                 'pinky purple': '#c94cbe',
(4)                 'pink purple': '#db4bda',
(4)                 'brownish red': '#9e3623',
(4)                 'dark rose': '#b5485d',
(4)                 'mud': '#735c12',
```

```
(4)                 'brownish': '#9c6d57',
(4)                 'emerald green': '#028f1e',
(4)                 'pale brown': '#b1916e',
(4)                 'dull blue': '#49759c',
(4)                 'burnt umber': '#a0450e',
(4)                 'medium green': '#39ad48',
(4)                 'clay': '#b66a50',
(4)                 'light aqua': '#8cffdb',
(4)                 'light olive green': '#a4be5c',
(4)                 'brownish orange': '#cb7723',
(4)                 'dark aqua': '#05696b',
(4)                 'purplish pink': '#ce5dae',
(4)                 'dark salmon': '#c85a53',
(4)                 'greenish grey': '#96ae8d',
(4)                 'jade': '#1fa774',
(4)                 'ugly green': '#7a9703',
(4)                 'dark beige': '#ac9362',
(4)                 'emerald': '#01a049',
(4)                 'pale red': '#d9544d',
(4)                 'light magenta': '#fa5ff7',
(4)                 'sky': '#82cafc',
(4)                 'light cyan': '#acfffc',
(4)                 'yellow orange': '#fcb001',
(4)                 'reddish purple': '#910951',
(4)                 'reddish pink': '#fe2c54',
(4)                 'orchid': '#c875c4',
(4)                 'dirty yellow': '#cdc50a',
(4)                 'orange red': '#fd411e',
(4)                 'deep red': '#9a0200',
(4)                 'orange brown': '#be6400',
(4)                 'cobalt blue': '#030aa7',
(4)                 'neon pink': '#fe019a',
(4)                 'rose pink': '#f7879a',
(4)                 'greyish purple': '#887191',
(4)                 'raspberry': '#b00149',
(4)                 'aqua green': '#12e193',
(4)                 'salmon pink': '#fe7b7c',
(4)                 'tangerine': '#ff9408',
(4)                 'brownish green': '#6a6e09',
(4)                 'red brown': '#8b2e16',
(4)                 'greenish brown': '#696112',
(4)                 'pumpkin': '#e17701',
(4)                 'pine green': '#0a481e',
(4)                 'charcoal': '#343837',
(4)                 'baby pink': '#ffb7ce',
(4)                 'cornflower': '#6a79f7',
(4)                 'blue violet': '#5d06e9',
(4)                 'chocolate': '#3d1c02',
(4)                 'greyish green': '#82a67d',
(4)                 'scarlet': '#be0119',
(4)                 'green yellow': '#c9ff27',
(4)                 'dark olive': '#373e02',
(4)                 'sienna': '#a9561e',
(4)                 'pastel purple': '#caa0ff',
(4)                 'terracotta': '#ca6641',
(4)                 'aqua blue': '#02d8e9',
(4)                 'sage green': '#88b378',
(4)                 'blood red': '#980002',
(4)                 'deep pink': '#cb0162',
(4)                 'grass': '#5cac2d',
(4)                 'moss': '#769958',
(4)                 'pastel blue': '#a2bffe',
(4)                 'bluish green': '#10a674',
(4)                 'green blue': '#06b48b',
(4)                 'dark tan': '#af884a',
(4)                 'greenish blue': '#0b8b87',
(4)                 'pale orange': '#ffa756',
(4)                 'vomit': '#a2a415',
(4)                 'forrest green': '#154406',
```

```
(4)                 'dark lavender': '#856798',
(4)                 'dark violet': '#34013f',
(4)                 'purple blue': '#632de9',
(4)                 'dark cyan': '#0a888a',
(4)                 'olive drab': '#6f7632',
(4)                 'pinkish': '#d46a7e',
(4)                 'cobalt': '#1e488f',
(4)                 'neon purple': '#bc13fe',
(4)                 'light turquoise': '#7ef4cc',
(4)                 'apple green': '#76cd26',
(4)                 'dull green': '#74a662',
(4)                 'wine': '#80013f',
(4)                 'powder blue': '#b1d1fc',
(4)                 'off white': '#ffffe4',
(4)                 'electric blue': '#0652ff',
(4)                 'dark turquoise': '#045c5a',
(4)                 'blue purple': '#5729ce',
(4)                 'azure': '#069af3',
(4)                 'bright red': '#ff000d',
(4)                 'pinkish red': '#f10c45',
(4)                 'cornflower blue': '#5170d7',
(4)                 'light olive': '#acbf69',
(4)                 'grape': '#6c3461',
(4)                 'greyish blue': '#5e819d',
(4)                 'purplish blue': '#601ef9',
(4)                 'yellowish green': '#b0dd16',
(4)                 'greenish yellow': '#cdfd02',
(4)                 'medium blue': '#2c6fbb',
(4)                 'dusty rose': '#c0737a',
(4)                 'light violet': '#d6b4fc',
(4)                 'midnight blue': '#020035',
(4)                 'bluish purple': '#703be7',
(4)                 'red orange': '#fd3c06',
(4)                 'dark magenta': '#960056',
(4)                 'greenish': '#40a368',
(4)                 'ocean blue': '#03719c',
(4)                 'coral': '#fc5a50',
(4)                 'cream': '#ffffc2',
(4)                 'reddish brown': '#7f2b0a',
(4)                 'burnt sienna': '#b04e0f',
(4)                 'brick': '#a03623',
(4)                 'sage': '#87ae73',
(4)                 'grey green': '#789b73',
(4)                 'white': '#ffffff',
(4)                 "robin's egg blue": '#98eff9',
(4)                 'moss green': '#658b38',
(4)                 'steel blue': '#5a7d9a',
(4)                 'eggplant': '#380835',
(4)                 'light yellow': '#fffe7a',
(4)                 'leaf green': '#5ca904',
(4)                 'light grey': '#d8dcd6',
(4)                 'puke': '#a5a502',
(4)                 'pinkish purple': '#d648d7',
(4)                 'sea blue': '#047495',
(4)                 'pale purple': '#b790d4',
(4)                 'slate blue': '#5b7c99',
(4)                 'blue grey': '#607c8e',
(4)                 'hunter green': '#0b4008',
(4)                 'fuchsia': '#ed0dd9',
(4)                 'crimson': '#8c000f',
(4)                 'pale yellow': '#ffff84',
(4)                 'ochre': '#bf9005',
(4)                 'mustard yellow': '#d2bd0a',
(4)                 'light red': '#ff474c',
(4)                 'cerulean': '#0485d1',
(4)                 'pale pink': '#ffcfdc',
(4)                 'deep blue': '#040273',
(4)                 'rust': '#a83c09',
(4)                 'light teal': '#90e4c1',
```

```
(4)                 'slate': '#516572',
(4)                 'goldenrod': '#fac205',
(4)                 'dark yellow': '#d5b60a',
(4)                 'dark grey': '#363737',
(4)                 'army green': '#4b5d16',
(4)                 'grey blue': '#6b8ba4',
(4)                 'seafoam': '#80f9ad',
(4)                 'puce': '#a57e52',
(4)                 'spring green': '#a9f971',
(4)                 'dark orange': '#c65102',
(4)                 'sand': '#e2ca76',
(4)                 'pastel green': '#b0ff9d',
(4)                 'mint': '#9ffeb0',
(4)                 'light orange': '#fdaa48',
(4)                 'bright pink': '#fe01b1',
(4)                 'chartreuse': '#c1f80a',
(4)                 'deep purple': '#36013f',
(4)                 'dark brown': '#341c02',
(4)                 'taupe': '#b9a281',
(4)                 'pea green': '#8eab12',
(4)                 'puke green': '#9aae07',
(4)                 'kelly green': '#02ab2e',
(4)                 'seafoam green': '#7af9ab',
(4)                 'blue green': '#137e6d',
(4)                 'khaki': '#aaa662',
(4)                 'burgundy': '#610023',
(4)                 'dark teal': '#014d4e',
(4)                 'brick red': '#8f1402',
(4)                 'royal purple': '#4b006e',
(4)                 'plum': '#580f41',
(4)                 'mint green': '#8fff9f',
(4)                 'gold': '#dbb40c',
(4)                 'baby blue': '#a2cffe',
(4)                 'yellow green': '#c0fb2d',
(4)                 'bright purple': '#be03fd',
(4)                 'dark red': '#840000',
(4)                 'pale blue': '#d0fefe',
(4)                 'grass green': '#3f9b0b',
(4)                 'navy': '#01153e',
(4)                 'aquamarine': '#04d8b2',
(4)                 'burnt orange': '#c04e01',
(4)                 'neon green': '#0cff0c',
(4)                 'bright blue': '#0165fc',
(4)                 'rose': '#cf6275',
(4)                 'light pink': '#ffd1df',
(4)                 'mustard': '#ceb301',
(4)                 'indigo': '#380282',
(4)                 'lime': '#aaff32',
(4)                 'sea green': '#53fca1',
(4)                 'periwinkle': '#8e82fe',
(4)                 'dark pink': '#cb416b',
(4)                 'olive green': '#677a04',
(4)                 'peach': '#ffb07c',
(4)                 'pale green': '#c7fdb5',
(4)                 'light brown': '#ad8150',
(4)                 'hot pink': '#ff028d',
(4)                 'black': '#000000',
(4)                 'lilac': '#cea2fd',
(4)                 'navy blue': '#001146',
(4)                 'royal blue': '#0504aa',
(4)                 'beige': '#e6daa6',
(4)                 'salmon': '#ff796c',
(4)                 'olive': '#6e750e',
(4)                 'maroon': '#650021',
(4)                 'bright green': '#01ff07',
(4)                 'dark purple': '#35063e',
(4)                 'mauve': '#ae7181',
(4)                 'forest green': '#06470c',
(4)                 'aqua': '#13eac9',
```

```
(4)                 'cyan': '#00ffff',
(4)                 'tan': '#d1b26f',
(4)                 'dark blue': '#00035b',
(4)                 'lavender': '#c79fef',
(4)                 'turquoise': '#06c2ac',
(4)                 'dark green': '#033500',
(4)                 'violet': '#9a0eea',
(4)                 'light purple': '#bf77f6',
(4)                 'lime green': '#89fe05',
(4)                 'grey': '#929591',
(4)                 'sky blue': '#75bbfd',
(4)                 'yellow': '#ffff14',
(4)                 'magenta': '#c20078',
(4)                 'light green': '#96f97b',
(4)                 'orange': '#f97306',
(4)                 'teal': '#029386',
(4)                 'light blue': '#95d0fc',
(4)                 'red': '#e50000',
(4)                 'brown': '#653700',
(4)                 'pink': '#ff81c0',
(4)                 'blue': '#0343df',
(4)                 'green': '#15b01a',
(4)                 'purple': '#7e1e9c'}
(0)             XKCD_COLORS = {'xkcd:' + name: value for name, value in XKCD_COLORS.items()}
(0)             CSS4_COLORS = {
(4)                 'aliceblue':            '#F0F8FF',
(4)                 'antiquewhite':         '#FAEBD7',
(4)                 'aqua':                 '#00FFFF',
(4)                 'aquamarine':           '#7FFFD4',
(4)                 'azure':                '#F0FFFF',
(4)                 'beige':                '#F5F5DC',
(4)                 'bisque':               '#FFE4C4',
(4)                 'black':                '#000000',
(4)                 'blanchedalmond':       '#FFEBCD',
(4)                 'blue':                 '#0000FF',
(4)                 'blueviolet':           '#8A2BE2',
(4)                 'brown':                '#A52A2A',
(4)                 'burlywood':            '#DEB887',
(4)                 'cadetblue':            '#5F9EA0',
(4)                 'chartreuse':           '#7FFF00',
(4)                 'chocolate':            '#D2691E',
(4)                 'coral':                '#FF7F50',
(4)                 'cornflowerblue':       '#6495ED',
(4)                 'cornsilk':             '#FFF8DC',
(4)                 'crimson':              '#DC143C',
(4)                 'cyan':                 '#00FFFF',
(4)                 'darkblue':             '#00008B',
(4)                 'darkcyan':             '#008B8B',
(4)                 'darkgoldenrod':        '#B8860B',
(4)                 'darkgray':             '#A9A9A9',
(4)                 'darkgreen':            '#006400',
(4)                 'darkgrey':             '#A9A9A9',
(4)                 'darkkhaki':            '#BDB76B',
(4)                 'darkmagenta':          '#8B008B',
(4)                 'darkolivegreen':       '#556B2F',
(4)                 'darkorange':           '#FF8C00',
(4)                 'darkorchid':           '#9932CC',
(4)                 'darkred':              '#8B0000',
(4)                 'darksalmon':           '#E9967A',
(4)                 'darkseagreen':         '#8FBC8F',
(4)                 'darkslateblue':        '#483D8B',
(4)                 'darkslategray':        '#2F4F4F',
(4)                 'darkslategrey':        '#2F4F4F',
(4)                 'darkturquoise':        '#00CED1',
(4)                 'darkviolet':           '#9400D3',
(4)                 'deeppink':             '#FF1493',
(4)                 'deepskyblue':          '#00BFFF',
(4)                 'dimgray':              '#696969',
(4)                 'dimgrey':              '#696969',
```

```
(4)                 'dodgerblue':           '#1E90FF',
(4)                 'firebrick':            '#B22222',
(4)                 'floralwhite':          '#FFFAF0',
(4)                 'forestgreen':          '#228B22',
(4)                 'fuchsia':              '#FF00FF',
(4)                 'gainsboro':            '#DCDCDC',
(4)                 'ghostwhite':           '#F8F8FF',
(4)                 'gold':                 '#FFD700',
(4)                 'goldenrod':            '#DAA520',
(4)                 'gray':                 '#808080',
(4)                 'green':                '#008000',
(4)                 'greenyellow':          '#ADFF2F',
(4)                 'grey':                 '#808080',
(4)                 'honeydew':             '#F0FFF0',
(4)                 'hotpink':              '#FF69B4',
(4)                 'indianred':            '#CD5C5C',
(4)                 'indigo':               '#4B0082',
(4)                 'ivory':                '#FFFFF0',
(4)                 'khaki':                '#F0E68C',
(4)                 'lavender':             '#E6E6FA',
(4)                 'lavenderblush':        '#FFF0F5',
(4)                 'lawngreen':            '#7CFC00',
(4)                 'lemonchiffon':         '#FFFACD',
(4)                 'lightblue':            '#ADD8E6',
(4)                 'lightcoral':           '#F08080',
(4)                 'lightcyan':            '#E0FFFF',
(4)                 'lightgoldenrodyellow': '#FAFAD2',
(4)                 'lightgray':            '#D3D3D3',
(4)                 'lightgreen':           '#90EE90',
(4)                 'lightgrey':            '#D3D3D3',
(4)                 'lightpink':            '#FFB6C1',
(4)                 'lightsalmon':          '#FFA07A',
(4)                 'lightseagreen':        '#20B2AA',
(4)                 'lightskyblue':         '#87CEFA',
(4)                 'lightslategray':       '#778899',
(4)                 'lightslategrey':       '#778899',
(4)                 'lightsteelblue':       '#B0C4DE',
(4)                 'lightyellow':          '#FFFFE0',
(4)                 'lime':                 '#00FF00',
(4)                 'limegreen':            '#32CD32',
(4)                 'linen':                '#FAF0E6',
(4)                 'magenta':              '#FF00FF',
(4)                 'maroon':               '#800000',
(4)                 'mediumaquamarine':     '#66CDAA',
(4)                 'mediumblue':           '#0000CD',
(4)                 'mediumorchid':         '#BA55D3',
(4)                 'mediumpurple':         '#9370DB',
(4)                 'mediumseagreen':       '#3CB371',
(4)                 'mediumslateblue':      '#7B68EE',
(4)                 'mediumspringgreen':    '#00FA9A',
(4)                 'mediumturquoise':      '#48D1CC',
(4)                 'mediumvioletred':      '#C71585',
(4)                 'midnightblue':         '#191970',
(4)                 'mintcream':            '#F5FFFA',
(4)                 'mistyrose':            '#FFE4E1',
(4)                 'moccasin':             '#FFE4B5',
(4)                 'navajowhite':          '#FFDEAD',
(4)                 'navy':                 '#000080',
(4)                 'oldlace':              '#FDF5E6',
(4)                 'olive':                '#808000',
(4)                 'olivedrab':            '#6B8E23',
(4)                 'orange':               '#FFA500',
(4)                 'orangered':            '#FF4500',
(4)                 'orchid':               '#DA70D6',
(4)                 'palegoldenrod':        '#EEE8AA',
(4)                 'palegreen':            '#98FB98',
(4)                 'paleturquoise':        '#AFEEEE',
(4)                 'palevioletred':        '#DB7093',
(4)                 'papayawhip':           '#FFEFD5',
```

```
1088: (4)               'peachpuff':            '#FFDAB9',
1089: (4)               'peru':                 '#CD853F',
1090: (4)               'pink':                 '#FFC0CB',
1091: (4)               'plum':                 '#DDA0DD',
1092: (4)               'powderblue':           '#B0E0E6',
1093: (4)               'purple':               '#800080',
1094: (4)               'rebeccapurple':        '#663399',
1095: (4)               'red':                  '#FF0000',
1096: (4)               'rosybrown':            '#BC8F8F',
1097: (4)               'royalblue':            '#4169E1',
1098: (4)               'saddlebrown':          '#8B4513',
1099: (4)               'salmon':               '#FA8072',
1100: (4)               'sandybrown':           '#F4A460',
1101: (4)               'seagreen':             '#2E8B57',
1102: (4)               'seashell':             '#FFF5EE',
1103: (4)               'sienna':               '#A0522D',
1104: (4)               'silver':               '#C0C0C0',
1105: (4)               'skyblue':              '#87CEEB',
1106: (4)               'slateblue':            '#6A5ACD',
1107: (4)               'slategray':            '#708090',
1108: (4)               'slategrey':            '#708090',
1109: (4)               'snow':                 '#FFFAFA',
1110: (4)               'springgreen':          '#00FF7F',
1111: (4)               'steelblue':            '#4682B4',
1112: (4)               'tan':                  '#D2B48C',
1113: (4)               'teal':                 '#008080',
1114: (4)               'thistle':              '#D8BFD8',
1115: (4)               'tomato':               '#FF6347',
1116: (4)               'turquoise':            '#40E0D0',
1117: (4)               'violet':               '#EE82EE',
1118: (4)               'wheat':                '#F5DEB3',
1119: (4)               'white':                '#FFFFFF',
1120: (4)               'whitesmoke':           '#F5F5F5',
1121: (4)               'yellow':               '#FFFF00',
1122: (4)               'yellowgreen':          '#9ACD32'}

----------------------------------------

File 8 - _constrained_layout.py:

1: (0)              """
2: (0)              Adjust subplot layouts so that there are no overlapping Axes or Axes
3: (0)              decorations.  All Axes decorations are dealt with (labels, ticks, titles,
4: (0)              ticklabels) and some dependent artists are also dealt with (colorbar,
5: (0)              suptitle).
6: (0)              Layout is done via `~matplotlib.gridspec`, with one constraint per gridspec,
7: (0)              so it is possible to have overlapping Axes if the gridspecs overlap (i.e.
8: (0)              using `~matplotlib.gridspec.GridSpecFromSubplotSpec`).  Axes placed using
9: (0)              ``figure.subplots()`` or ``figure.add_subplots()`` will participate in the
10: (0)             layout.  Axes manually placed via ``figure.add_axes()`` will not.
11: (0)             See Tutorial: :ref:`constrainedlayout_guide`
12: (0)             General idea:
13: (0)             -------------
14: (0)             First, a figure has a gridspec that divides the figure into nrows and ncols,
15: (0)             with heights and widths set by ``height_ratios`` and ``width_ratios``,
16: (0)             often just set to 1 for an equal grid.
17: (0)             Subplotspecs that are derived from this gridspec can contain either a
18: (0)             ``SubPanel``, a ``GridSpecFromSubplotSpec``, or an ``Axes``.  The ``SubPanel``
19: (0)             and ``GridSpecFromSubplotSpec`` are dealt with recursively and each contain an
20: (0)             analogous layout.
21: (0)             Each ``GridSpec`` has a ``_layoutgrid`` attached to it.  The ``_layoutgrid``
22: (0)             has the same logical layout as the ``GridSpec``.   Each row of the grid spec
23: (0)             has a top and bottom "margin" and each column has a left and right "margin".
24: (0)             The "inner" height of each row is constrained to be the same (or as modified
25: (0)             by ``height_ratio``), and the "inner" width of each column is
26: (0)             constrained to be the same (as modified by ``width_ratio``), where "inner"
27: (0)             is the width or height of each column/row minus the size of the margins.
28: (0)             Then the size of the margins for each row and column are determined as the
29: (0)             max width of the decorators on each Axes that has decorators in that margin.
```

```
 30: (0)             For instance, a normal Axes would have a left margin that includes the
 31: (0)             left ticklabels, and the ylabel if it exists.  The right margin may include a
 32: (0)             colorbar, the bottom margin the xaxis decorations, and the top margin the
 33: (0)             title.
 34: (0)             With these constraints, the solver then finds appropriate bounds for the
 35: (0)             columns and rows.  It's possible that the margins take up the whole figure,
 36: (0)             in which case the algorithm is not applied and a warning is raised.
 37: (0)             See the tutorial :ref:`constrainedlayout_guide`
 38: (0)             for more discussion of the algorithm with examples.
 39: (0)             """
 40: (0)             import logging
 41: (0)             import numpy as np
 42: (0)             from matplotlib import _api, artist as martist
 43: (0)             import matplotlib.transforms as mtransforms
 44: (0)             import matplotlib._layoutgrid as mlayoutgrid
 45: (0)             _log = logging.getLogger(__name__)
 46: (0)             def do_constrained_layout(fig, h_pad, w_pad,
 47: (26)                                       hspace=None, wspace=None, rect=(0, 0, 1, 1),
 48: (26)                                       compress=False):
 49: (4)                 """
 50: (4)                 Do the constrained_layout.  Called at draw time in
 51: (5)                  ``figure.constrained_layout()``
 52: (4)                 Parameters
 53: (4)                 ----------
 54: (4)                 fig : `~matplotlib.figure.Figure`
 55: (8)                     `.Figure` instance to do the layout in.
 56: (4)                 h_pad, w_pad : float
 57: (6)                   Padding around the Axes elements in figure-normalized units.
 58: (4)                 hspace, wspace : float
 59: (7)                    Fraction of the figure to dedicate to space between the
 60: (7)                    Axes.  These are evenly spread between the gaps between the Axes.
 61: (7)                    A value of 0.2 for a three-column layout would have a space
 62: (7)                    of 0.1 of the figure width between each column.
 63: (7)                    If h/wspace < h/w_pad, then the pads are used instead.
 64: (4)                 rect : tuple of 4 floats
 65: (8)                     Rectangle in figure coordinates to perform constrained layout in
 66: (8)                     [left, bottom, width, height], each from 0-1.
 67: (4)                 compress : bool
 68: (8)                     Whether to shift Axes so that white space in between them is
 69: (8)                     removed. This is useful for simple grids of fixed-aspect Axes (e.g.
 70: (8)                     a grid of images).
 71: (4)                 Returns
 72: (4)                 -------
 73: (4)                 layoutgrid : private debugging structure
 74: (4)                 """
 75: (4)                 renderer = fig._get_renderer()
 76: (4)                 layoutgrids = make_layoutgrids(fig, None, rect=rect)
 77: (4)                 if not layoutgrids['hasgrids']:
 78: (8)                     _api.warn_external('There are no gridspecs with layoutgrids. '
 79: (27)                                        'Possibly did not call parent GridSpec with the'
 80: (27)                                        ' "figure" keyword')
 81: (8)                     return
 82: (4)                 for _ in range(2):
 83: (8)                     make_layout_margins(layoutgrids, fig, renderer, h_pad=h_pad,
 84: (28)                                         w_pad=w_pad, hspace=hspace, wspace=wspace)
 85: (8)                     make_margin_suptitles(layoutgrids, fig, renderer, h_pad=h_pad,
 86: (30)                                           w_pad=w_pad)
 87: (8)                     match_submerged_margins(layoutgrids, fig)
 88: (8)                     layoutgrids[fig].update_variables()
 89: (8)                     warn_collapsed = ('constrained_layout not applied because '
 90: (26)                                      'axes sizes collapsed to zero.  Try making '
 91: (26)                                      'figure larger or Axes decorations smaller.')
 92: (8)                     if check_no_collapsed_axes(layoutgrids, fig):
 93: (12)                        reposition_axes(layoutgrids, fig, renderer, h_pad=h_pad,
 94: (28)                                         w_pad=w_pad, hspace=hspace, wspace=wspace)
 95: (12)                        if compress:
 96: (16)                            layoutgrids = compress_fixed_aspect(layoutgrids, fig)
 97: (16)                            layoutgrids[fig].update_variables()
 98: (16)                            if check_no_collapsed_axes(layoutgrids, fig):
```

```
 99: (20)                                reposition_axes(layoutgrids, fig, renderer, h_pad=h_pad,
100: (36)                                                w_pad=w_pad, hspace=hspace, wspace=wspace)
101: (16)                            else:
102: (20)                                _api.warn_external(warn_collapsed)
103: (8)                    else:
104: (12)                       _api.warn_external(warn_collapsed)
105: (8)                    reset_margins(layoutgrids, fig)
106: (4)                return layoutgrids
107: (0)            def make_layoutgrids(fig, layoutgrids, rect=(0, 0, 1, 1)):
108: (4)                """
109: (4)                Make the layoutgrid tree.
110: (4)                (Sub)Figures get a layoutgrid so we can have figure margins.
111: (4)                Gridspecs that are attached to Axes get a layoutgrid so Axes
112: (4)                can have margins.
113: (4)                """
114: (4)                if layoutgrids is None:
115: (8)                    layoutgrids = dict()
116: (8)                    layoutgrids['hasgrids'] = False
117: (4)                if not hasattr(fig, '_parent'):
118: (8)                    layoutgrids[fig] = mlayoutgrid.LayoutGrid(parent=rect, name='figlb')
119: (4)                else:
120: (8)                    gs = fig._subplotspec.get_gridspec()
121: (8)                    layoutgrids = make_layoutgrids_gs(layoutgrids, gs)
122: (8)                    parentlb = layoutgrids[gs]
123: (8)                    layoutgrids[fig] = mlayoutgrid.LayoutGrid(
124: (12)                       parent=parentlb,
125: (12)                       name='panellb',
126: (12)                       parent_inner=True,
127: (12)                       nrows=1, ncols=1,
128: (12)                       parent_pos=(fig._subplotspec.rowspan,
129: (24)                                   fig._subplotspec.colspan))
130: (4)                for sfig in fig.subfigs:
131: (8)                    layoutgrids = make_layoutgrids(sfig, layoutgrids)
132: (4)                for ax in fig._localaxes:
133: (8)                    gs = ax.get_gridspec()
134: (8)                    if gs is not None:
135: (12)                       layoutgrids = make_layoutgrids_gs(layoutgrids, gs)
136: (4)                return layoutgrids
137: (0)            def make_layoutgrids_gs(layoutgrids, gs):
138: (4)                """
139: (4)                Make the layoutgrid for a gridspec (and anything nested in the gridspec)
140: (4)                """
141: (4)                if gs in layoutgrids or gs.figure is None:
142: (8)                    return layoutgrids
143: (4)                layoutgrids['hasgrids'] = True
144: (4)                if not hasattr(gs, '_subplot_spec'):
145: (8)                    parent = layoutgrids[gs.figure]
146: (8)                    layoutgrids[gs] = mlayoutgrid.LayoutGrid(
147: (16)                           parent=parent,
148: (16)                           parent_inner=True,
149: (16)                           name='gridspec',
150: (16)                           ncols=gs._ncols, nrows=gs._nrows,
151: (16)                           width_ratios=gs.get_width_ratios(),
152: (16)                           height_ratios=gs.get_height_ratios())
153: (4)                else:
154: (8)                    subplot_spec = gs._subplot_spec
155: (8)                    parentgs = subplot_spec.get_gridspec()
156: (8)                    if parentgs not in layoutgrids:
157: (12)                       layoutgrids = make_layoutgrids_gs(layoutgrids, parentgs)
158: (8)                    subspeclb = layoutgrids[parentgs]
159: (8)                    rep = (gs, 'top')
160: (8)                    if rep not in layoutgrids:
161: (12)                       layoutgrids[rep] = mlayoutgrid.LayoutGrid(
162: (16)                           parent=subspeclb,
163: (16)                           name='top',
164: (16)                           nrows=1, ncols=1,
165: (16)                           parent_pos=(subplot_spec.rowspan, subplot_spec.colspan))
166: (8)                    layoutgrids[gs] = mlayoutgrid.LayoutGrid(
167: (16)                           parent=layoutgrids[rep],
```

```
(16)                            name='gridspec',
(16)                            nrows=gs._nrows, ncols=gs._ncols,
(16)                            width_ratios=gs.get_width_ratios(),
(16)                            height_ratios=gs.get_height_ratios())
(4)                 return layoutgrids
(0)             def check_no_collapsed_axes(layoutgrids, fig):
(4)                 """
(4)                 Check that no Axes have collapsed to zero size.
(4)                 """
(4)                 for sfig in fig.subfigs:
(8)                     ok = check_no_collapsed_axes(layoutgrids, sfig)
(8)                     if not ok:
(12)                        return False
(4)                 for ax in fig.axes:
(8)                     gs = ax.get_gridspec()
(8)                     if gs in layoutgrids:  # also implies gs is not None.
(12)                        lg = layoutgrids[gs]
(12)                        for i in range(gs.nrows):
(16)                            for j in range(gs.ncols):
(20)                                bb = lg.get_inner_bbox(i, j)
(20)                                if bb.width <= 0 or bb.height <= 0:
(24)                                    return False
(4)                 return True
(0)             def compress_fixed_aspect(layoutgrids, fig):
(4)                 gs = None
(4)                 for ax in fig.axes:
(8)                     if ax.get_subplotspec() is None:
(12)                        continue
(8)                     ax.apply_aspect()
(8)                     sub = ax.get_subplotspec()
(8)                     _gs = sub.get_gridspec()
(8)                     if gs is None:
(12)                        gs = _gs
(12)                        extraw = np.zeros(gs.ncols)
(12)                        extrah = np.zeros(gs.nrows)
(8)                     elif _gs != gs:
(12)                        raise ValueError('Cannot do compressed layout if Axes are not'
(29)                                         'all from the same gridspec')
(8)                     orig = ax.get_position(original=True)
(8)                     actual = ax.get_position(original=False)
(8)                     dw = orig.width - actual.width
(8)                     if dw > 0:
(12)                        extraw[sub.colspan] = np.maximum(extraw[sub.colspan], dw)
(8)                     dh = orig.height - actual.height
(8)                     if dh > 0:
(12)                        extrah[sub.rowspan] = np.maximum(extrah[sub.rowspan], dh)
(4)                 if gs is None:
(8)                     raise ValueError('Cannot do compressed layout if no Axes '
(25)                                     'are part of a gridspec.')
(4)                 w = np.sum(extraw) / 2
(4)                 layoutgrids[fig].edit_margin_min('left', w)
(4)                 layoutgrids[fig].edit_margin_min('right', w)
(4)                 h = np.sum(extrah) / 2
(4)                 layoutgrids[fig].edit_margin_min('top', h)
(4)                 layoutgrids[fig].edit_margin_min('bottom', h)
(4)                 return layoutgrids
(0)             def get_margin_from_padding(obj, *, w_pad=0, h_pad=0,
(28)                                        hspace=0, wspace=0):
(4)                 ss = obj._subplotspec
(4)                 gs = ss.get_gridspec()
(4)                 if hasattr(gs, 'hspace'):
(8)                     _hspace = (gs.hspace if gs.hspace is not None else hspace)
(8)                     _wspace = (gs.wspace if gs.wspace is not None else wspace)
(4)                 else:
(8)                     _hspace = (gs._hspace if gs._hspace is not None else hspace)
(8)                     _wspace = (gs._wspace if gs._wspace is not None else wspace)
(4)                 _wspace = _wspace / 2
(4)                 _hspace = _hspace / 2
(4)                 nrows, ncols = gs.get_geometry()
```

```
(4)                 margin = {'leftcb': w_pad, 'rightcb': w_pad,
(14)                          'bottomcb': h_pad, 'topcb': h_pad,
(14)                          'left': 0, 'right': 0,
(14)                          'top': 0, 'bottom': 0}
(4)                 if _wspace / ncols > w_pad:
(8)                     if ss.colspan.start > 0:
(12)                        margin['leftcb'] = _wspace / ncols
(8)                     if ss.colspan.stop < ncols:
(12)                        margin['rightcb'] = _wspace / ncols
(4)                 if _hspace / nrows > h_pad:
(8)                     if ss.rowspan.stop < nrows:
(12)                        margin['bottomcb'] = _hspace / nrows
(8)                     if ss.rowspan.start > 0:
(12)                        margin['topcb'] = _hspace / nrows
(4)                 return margin
(0)             def make_layout_margins(layoutgrids, fig, renderer, *, w_pad=0, h_pad=0,
(24)                                    hspace=0, wspace=0):
(4)                 """
(4)                 For each Axes, make a margin between the *pos* layoutbox and the
(4)                 *axes* layoutbox be a minimum size that can accommodate the
(4)                 decorations on the axis.
(4)                 Then make room for colorbars.
(4)                 Parameters
(4)                 ----------
(4)                 layoutgrids : dict
(4)                 fig : `~matplotlib.figure.Figure`
(8)                     `.Figure` instance to do the layout in.
(4)                 renderer : `~matplotlib.backend_bases.RendererBase` subclass.
(8)                     The renderer to use.
(4)                 w_pad, h_pad : float, default: 0
(8)                     Width and height padding (in fraction of figure).
(4)                 hspace, wspace : float, default: 0
(8)                     Width and height padding as fraction of figure size divided by
(8)                     number of columns or rows.
(4)                 """
(4)                 for sfig in fig.subfigs:  # recursively make child panel margins
(8)                     ss = sfig._subplotspec
(8)                     gs = ss.get_gridspec()
(8)                     make_layout_margins(layoutgrids, sfig, renderer,
(28)                                        w_pad=w_pad, h_pad=h_pad,
(28)                                        hspace=hspace, wspace=wspace)
(8)                     margins = get_margin_from_padding(sfig, w_pad=0, h_pad=0,
(42)                                                      hspace=hspace, wspace=wspace)
(8)                     layoutgrids[gs].edit_outer_margin_mins(margins, ss)
(4)                 for ax in fig._localaxes:
(8)                     if not ax.get_subplotspec() or not ax.get_in_layout():
(12)                        continue
(8)                     ss = ax.get_subplotspec()
(8)                     gs = ss.get_gridspec()
(8)                     if gs not in layoutgrids:
(12)                        return
(8)                     margin = get_margin_from_padding(ax, w_pad=w_pad, h_pad=h_pad,
(41)                                                     hspace=hspace, wspace=wspace)
(8)                     pos, bbox = get_pos_and_bbox(ax, renderer)
(8)                     margin['left'] += pos.x0 - bbox.x0
(8)                     margin['right'] += bbox.x1 - pos.x1
(8)                     margin['bottom'] += pos.y0 - bbox.y0
(8)                     margin['top'] += bbox.y1 - pos.y1
(8)                     for cbax in ax._colorbars:
(12)                        pad = colorbar_get_pad(layoutgrids, cbax)
(12)                        cbp_rspan, cbp_cspan = get_cb_parent_spans(cbax)
(12)                        loc = cbax._colorbar_info['location']
(12)                        cbpos, cbbbox = get_pos_and_bbox(cbax, renderer)
(12)                        if loc == 'right':
(16)                            if cbp_cspan.stop == ss.colspan.stop:
(20)                                margin['rightcb'] += cbbbox.width + pad
(12)                        elif loc == 'left':
(16)                            if cbp_cspan.start == ss.colspan.start:
(20)                                margin['leftcb'] += cbbbox.width + pad
```

```
306: (12)                        elif loc == 'top':
307: (16)                            if cbp_rspan.start == ss.rowspan.start:
308: (20)                                margin['topcb'] += cbbbox.height + pad
309: (12)                        else:
310: (16)                            if cbp_rspan.stop == ss.rowspan.stop:
311: (20)                                margin['bottomcb'] += cbbbox.height + pad
312: (12)                        if loc in ['top', 'bottom']:
313: (16)                            if (cbp_cspan.start == ss.colspan.start and
314: (24)                                    cbbbox.x0 < bbox.x0):
315: (20)                                margin['left'] += bbox.x0 - cbbbox.x0
316: (16)                            if (cbp_cspan.stop == ss.colspan.stop and
317: (24)                                    cbbbox.x1 > bbox.x1):
318: (20)                                margin['right'] += cbbbox.x1 - bbox.x1
319: (12)                        if loc in ['left', 'right']:
320: (16)                            if (cbp_rspan.stop == ss.rowspan.stop and
321: (24)                                    cbbbox.y0 < bbox.y0):
322: (20)                                margin['bottom'] += bbox.y0 - cbbbox.y0
323: (16)                            if (cbp_rspan.start == ss.rowspan.start and
324: (24)                                    cbbbox.y1 > bbox.y1):
325: (20)                                margin['top'] += cbbbox.y1 - bbox.y1
326: (8)                    layoutgrids[gs].edit_outer_margin_mins(margin, ss)
327: (4)                for leg in fig.legends:
328: (8)                    inv_trans_fig = None
329: (8)                    if leg._outside_loc and leg._bbox_to_anchor is None:
330: (12)                       if inv_trans_fig is None:
331: (16)                           inv_trans_fig = fig.transFigure.inverted().transform_bbox
332: (12)                       bbox = inv_trans_fig(leg.get_tightbbox(renderer))
333: (12)                       w = bbox.width + 2 * w_pad
334: (12)                       h = bbox.height + 2 * h_pad
335: (12)                       legendloc = leg._outside_loc
336: (12)                       if legendloc == 'lower':
337: (16)                           layoutgrids[fig].edit_margin_min('bottom', h)
338: (12)                       elif legendloc == 'upper':
339: (16)                           layoutgrids[fig].edit_margin_min('top', h)
340: (12)                       if legendloc == 'right':
341: (16)                           layoutgrids[fig].edit_margin_min('right', w)
342: (12)                       elif legendloc == 'left':
343: (16)                           layoutgrids[fig].edit_margin_min('left', w)
344: (0)            def make_margin_suptitles(layoutgrids, fig, renderer, *, w_pad=0, h_pad=0):
345: (4)                inv_trans_fig = fig.transFigure.inverted().transform_bbox
346: (4)                padbox = mtransforms.Bbox([[0, 0], [w_pad, h_pad]])
347: (4)                padbox = (fig.transFigure -
348: (14)                         fig.transSubfigure).transform_bbox(padbox)
349: (4)                h_pad_local = padbox.height
350: (4)                w_pad_local = padbox.width
351: (4)                for sfig in fig.subfigs:
352: (8)                    make_margin_suptitles(layoutgrids, sfig, renderer,
353: (30)                                         w_pad=w_pad, h_pad=h_pad)
354: (4)                if fig._suptitle is not None and fig._suptitle.get_in_layout():
355: (8)                    p = fig._suptitle.get_position()
356: (8)                    if getattr(fig._suptitle, '_autopos', False):
357: (12)                       fig._suptitle.set_position((p[0], 1 - h_pad_local))
358: (12)                       bbox = inv_trans_fig(fig._suptitle.get_tightbbox(renderer))
359: (12)                       layoutgrids[fig].edit_margin_min('top', bbox.height + 2 * h_pad)
360: (4)                if fig._supxlabel is not None and fig._supxlabel.get_in_layout():
361: (8)                    p = fig._supxlabel.get_position()
362: (8)                    if getattr(fig._supxlabel, '_autopos', False):
363: (12)                       fig._supxlabel.set_position((p[0], h_pad_local))
364: (12)                       bbox = inv_trans_fig(fig._supxlabel.get_tightbbox(renderer))
365: (12)                       layoutgrids[fig].edit_margin_min('bottom',
366: (45)                                                        bbox.height + 2 * h_pad)
367: (4)                if fig._supylabel is not None and fig._supylabel.get_in_layout():
368: (8)                    p = fig._supylabel.get_position()
369: (8)                    if getattr(fig._supylabel, '_autopos', False):
370: (12)                       fig._supylabel.set_position((w_pad_local, p[1]))
371: (12)                       bbox = inv_trans_fig(fig._supylabel.get_tightbbox(renderer))
372: (12)                       layoutgrids[fig].edit_margin_min('left', bbox.width + 2 * w_pad)
373: (0)            def match_submerged_margins(layoutgrids, fig):
374: (4)                """
```

```
(4)                 Make the margins that are submerged inside an Axes the same size.
(4)                 This allows Axes that span two columns (or rows) that are offset
(4)                 from one another to have the same size.
(4)                 This gives the proper layout for something like::
(8)                     fig = plt.figure(constrained_layout=True)
(8)                     axs = fig.subplot_mosaic("AAAB\nCCDD")
(4)                 Without this routine, the Axes D will be wider than C, because the
(4)                 margin width between the two columns in C has no width by default,
(4)                 whereas the margins between the two columns of D are set by the
(4)                 width of the margin between A and B. However, obviously the user would
(4)                 like C and D to be the same size, so we need to add constraints to these
(4)                 "submerged" margins.
(4)                 This routine makes all the interior margins the same, and the spacing
(4)                 between the three columns in A and the two column in C are all set to the
(4)                 margins between the two columns of D.
(4)                 See test_constrained_layout::test_constrained_layout12 for an example.
(4)                 """
(4)                 for sfig in fig.subfigs:
(8)                     match_submerged_margins(layoutgrids, sfig)
(4)                 axs = [a for a in fig.get_axes()
(11)                       if a.get_subplotspec() is not None and a.get_in_layout()]
(4)                 for ax1 in axs:
(8)                     ss1 = ax1.get_subplotspec()
(8)                     if ss1.get_gridspec() not in layoutgrids:
(12)                        axs.remove(ax1)
(12)                        continue
(8)                     lg1 = layoutgrids[ss1.get_gridspec()]
(8)                     if len(ss1.colspan) > 1:
(12)                        maxsubl = np.max(
(16)                            lg1.margin_vals['left'][ss1.colspan[1:]] +
(16)                            lg1.margin_vals['leftcb'][ss1.colspan[1:]]
(12)                        )
(12)                        maxsubr = np.max(
(16)                            lg1.margin_vals['right'][ss1.colspan[:-1]] +
(16)                            lg1.margin_vals['rightcb'][ss1.colspan[:-1]]
(12)                        )
(12)                        for ax2 in axs:
(16)                            ss2 = ax2.get_subplotspec()
(16)                            lg2 = layoutgrids[ss2.get_gridspec()]
(16)                            if lg2 is not None and len(ss2.colspan) > 1:
(20)                                maxsubl2 = np.max(
(24)                                    lg2.margin_vals['left'][ss2.colspan[1:]] +
(24)                                    lg2.margin_vals['leftcb'][ss2.colspan[1:]])
(20)                                if maxsubl2 > maxsubl:
(24)                                    maxsubl = maxsubl2
(20)                                maxsubr2 = np.max(
(24)                                    lg2.margin_vals['right'][ss2.colspan[:-1]] +
(24)                                    lg2.margin_vals['rightcb'][ss2.colspan[:-1]])
(20)                                if maxsubr2 > maxsubr:
(24)                                    maxsubr = maxsubr2
(12)                        for i in ss1.colspan[1:]:
(16)                            lg1.edit_margin_min('left', maxsubl, cell=i)
(12)                        for i in ss1.colspan[:-1]:
(16)                            lg1.edit_margin_min('right', maxsubr, cell=i)
(8)                     if len(ss1.rowspan) > 1:
(12)                        maxsubt = np.max(
(16)                            lg1.margin_vals['top'][ss1.rowspan[1:]] +
(16)                            lg1.margin_vals['topcb'][ss1.rowspan[1:]]
(12)                        )
(12)                        maxsubb = np.max(
(16)                            lg1.margin_vals['bottom'][ss1.rowspan[:-1]] +
(16)                            lg1.margin_vals['bottomcb'][ss1.rowspan[:-1]]
(12)                        )
(12)                        for ax2 in axs:
(16)                            ss2 = ax2.get_subplotspec()
(16)                            lg2 = layoutgrids[ss2.get_gridspec()]
(16)                            if lg2 is not None:
(20)                                if len(ss2.rowspan) > 1:
(24)                                    maxsubt = np.max([np.max(
```

```
(28)                                    lg2.margin_vals['top'][ss2.rowspan[1:]] +
(28)                                    lg2.margin_vals['topcb'][ss2.rowspan[1:]]
(24)                                ), maxsubt])
(24)                                maxsubb = np.max([np.max(
(28)                                    lg2.margin_vals['bottom'][ss2.rowspan[:-1]] +
(28)                                    lg2.margin_vals['bottomcb'][ss2.rowspan[:-1]]
(24)                                ), maxsubb])
(12)                        for i in ss1.rowspan[1:]:
(16)                            lg1.edit_margin_min('top', maxsubt, cell=i)
(12)                        for i in ss1.rowspan[:-1]:
(16)                            lg1.edit_margin_min('bottom', maxsubb, cell=i)
(0)             def get_cb_parent_spans(cbax):
(4)                 """
(4)                 Figure out which subplotspecs this colorbar belongs to.
(4)                 Parameters
(4)                 ----------
(4)                 cbax : `~matplotlib.axes.Axes`
(8)                     Axes for the colorbar.
(4)                 """
(4)                 rowstart = np.inf
(4)                 rowstop = -np.inf
(4)                 colstart = np.inf
(4)                 colstop = -np.inf
(4)                 for parent in cbax._colorbar_info['parents']:
(8)                     ss = parent.get_subplotspec()
(8)                     rowstart = min(ss.rowspan.start, rowstart)
(8)                     rowstop = max(ss.rowspan.stop, rowstop)
(8)                     colstart = min(ss.colspan.start, colstart)
(8)                     colstop = max(ss.colspan.stop, colstop)
(4)                 rowspan = range(rowstart, rowstop)
(4)                 colspan = range(colstart, colstop)
(4)                 return rowspan, colspan
(0)             def get_pos_and_bbox(ax, renderer):
(4)                 """
(4)                 Get the position and the bbox for the Axes.
(4)                 Parameters
(4)                 ----------
(4)                 ax : `~matplotlib.axes.Axes`
(4)                 renderer : `~matplotlib.backend_bases.RendererBase` subclass.
(4)                 Returns
(4)                 -------
(4)                 pos : `~matplotlib.transforms.Bbox`
(8)                     Position in figure coordinates.
(4)                 bbox : `~matplotlib.transforms.Bbox`
(8)                     Tight bounding box in figure coordinates.
(4)                 """
(4)                 fig = ax.figure
(4)                 pos = ax.get_position(original=True)
(4)                 pos = pos.transformed(fig.transSubfigure - fig.transFigure)
(4)                 tightbbox = martist._get_tightbbox_for_layout_only(ax, renderer)
(4)                 if tightbbox is None:
(8)                     bbox = pos
(4)                 else:
(8)                     bbox = tightbbox.transformed(fig.transFigure.inverted())
(4)                 return pos, bbox
(0)             def reposition_axes(layoutgrids, fig, renderer, *,
(20)                                w_pad=0, h_pad=0, hspace=0, wspace=0):
(4)                 """
(4)                 Reposition all the Axes based on the new inner bounding box.
(4)                 """
(4)                 trans_fig_to_subfig = fig.transFigure - fig.transSubfigure
(4)                 for sfig in fig.subfigs:
(8)                     bbox = layoutgrids[sfig].get_outer_bbox()
(8)                     sfig._redo_transform_rel_fig(
(12)                        bbox=bbox.transformed(trans_fig_to_subfig))
(8)                     reposition_axes(layoutgrids, sfig, renderer,
(24)                                    w_pad=w_pad, h_pad=h_pad,
(24)                                    wspace=wspace, hspace=hspace)
(4)                 for ax in fig._localaxes:
```

```
(8)                     if ax.get_subplotspec() is None or not ax.get_in_layout():
(12)                        continue
(8)                     ss = ax.get_subplotspec()
(8)                     gs = ss.get_gridspec()
(8)                     if gs not in layoutgrids:
(12)                        return
(8)                     bbox = layoutgrids[gs].get_inner_bbox(rows=ss.rowspan,
(46)                                                          cols=ss.colspan)
(8)                     newbbox = trans_fig_to_subfig.transform_bbox(bbox)
(8)                     ax._set_position(newbbox)
(8)                     offset = {'left': 0, 'right': 0, 'bottom': 0, 'top': 0}
(8)                     for nn, cbax in enumerate(ax._colorbars[::-1]):
(12)                        if ax == cbax._colorbar_info['parents'][0]:
(16)                            reposition_colorbar(layoutgrids, cbax, renderer,
(36)                                                offset=offset)
(0)             def reposition_colorbar(layoutgrids, cbax, renderer, *, offset=None):
(4)                 """
(4)                 Place the colorbar in its new place.
(4)                 Parameters
(4)                 ----------
(4)                 layoutgrids : dict
(4)                 cbax : `~matplotlib.axes.Axes`
(8)                     Axes for the colorbar.
(4)                 renderer : `~matplotlib.backend_bases.RendererBase` subclass.
(8)                     The renderer to use.
(4)                 offset : array-like
(8)                     Offset the colorbar needs to be pushed to in order to
(8)                     account for multiple colorbars.
(4)                 """
(4)                 parents = cbax._colorbar_info['parents']
(4)                 gs = parents[0].get_gridspec()
(4)                 fig = cbax.figure
(4)                 trans_fig_to_subfig = fig.transFigure - fig.transSubfigure
(4)                 cb_rspans, cb_cspans = get_cb_parent_spans(cbax)
(4)                 bboxparent = layoutgrids[gs].get_bbox_for_cb(rows=cb_rspans,
(49)                                                             cols=cb_cspans)
(4)                 pb = layoutgrids[gs].get_inner_bbox(rows=cb_rspans, cols=cb_cspans)
(4)                 location = cbax._colorbar_info['location']
(4)                 anchor = cbax._colorbar_info['anchor']
(4)                 fraction = cbax._colorbar_info['fraction']
(4)                 aspect = cbax._colorbar_info['aspect']
(4)                 shrink = cbax._colorbar_info['shrink']
(4)                 cbpos, cbbbox = get_pos_and_bbox(cbax, renderer)
(4)                 cbpad = colorbar_get_pad(layoutgrids, cbax)
(4)                 if location in ('left', 'right'):
(8)                     pbcb = pb.shrunk(fraction, shrink).anchored(anchor, pb)
(8)                     if location == 'right':
(12)                        lmargin = cbpos.x0 - cbbbox.x0
(12)                        dx = bboxparent.x1 - pbcb.x0 + offset['right']
(12)                        dx += cbpad + lmargin
(12)                        offset['right'] += cbbbox.width + cbpad
(12)                        pbcb = pbcb.translated(dx, 0)
(8)                     else:
(12)                        lmargin = cbpos.x0 - cbbbox.x0
(12)                        dx = bboxparent.x0 - pbcb.x0  # edge of parent
(12)                        dx += -cbbbox.width - cbpad + lmargin - offset['left']
(12)                        offset['left'] += cbbbox.width + cbpad
(12)                        pbcb = pbcb.translated(dx, 0)
(4)                 else:  # horizontal axes:
(8)                     pbcb = pb.shrunk(shrink, fraction).anchored(anchor, pb)
(8)                     if location == 'top':
(12)                        bmargin = cbpos.y0 - cbbbox.y0
(12)                        dy = bboxparent.y1 - pbcb.y0 + offset['top']
(12)                        dy += cbpad + bmargin
(12)                        offset['top'] += cbbbox.height + cbpad
(12)                        pbcb = pbcb.translated(0, dy)
(8)                     else:
(12)                        bmargin = cbpos.y0 - cbbbox.y0
(12)                        dy = bboxparent.y0 - pbcb.y0
```

```
582: (12)                        dy += -cbbbox.height - cbpad + bmargin - offset['bottom']
583: (12)                        offset['bottom'] += cbbbox.height + cbpad
584: (12)                        pbcb = pbcb.translated(0, dy)
585: (4)                pbcb = trans_fig_to_subfig.transform_bbox(pbcb)
586: (4)                cbax.set_transform(fig.transSubfigure)
587: (4)                cbax._set_position(pbcb)
588: (4)                cbax.set_anchor(anchor)
589: (4)                if location in ['bottom', 'top']:
590: (8)                    aspect = 1 / aspect
591: (4)                cbax.set_box_aspect(aspect)
592: (4)                cbax.set_aspect('auto')
593: (4)                return offset
594: (0)            def reset_margins(layoutgrids, fig):
595: (4)                """
596: (4)                Reset the margins in the layoutboxes of *fig*.
597: (4)                Margins are usually set as a minimum, so if the figure gets smaller
598: (4)                the minimum needs to be zero in order for it to grow again.
599: (4)                """
600: (4)                for sfig in fig.subfigs:
601: (8)                    reset_margins(layoutgrids, sfig)
602: (4)                for ax in fig.axes:
603: (8)                    if ax.get_in_layout():
604: (12)                       gs = ax.get_gridspec()
605: (12)                       if gs in layoutgrids:  # also implies gs is not None.
606: (16)                           layoutgrids[gs].reset_margins()
607: (4)                layoutgrids[fig].reset_margins()
608: (0)            def colorbar_get_pad(layoutgrids, cax):
609: (4)                parents = cax._colorbar_info['parents']
610: (4)                gs = parents[0].get_gridspec()
611: (4)                cb_rspans, cb_cspans = get_cb_parent_spans(cax)
612: (4)                bboxouter = layoutgrids[gs].get_inner_bbox(rows=cb_rspans, cols=cb_cspans)
613: (4)                if cax._colorbar_info['location'] in ['right', 'left']:
614: (8)                    size = bboxouter.width
615: (4)                else:
616: (8)                    size = bboxouter.height
617: (4)                return cax._colorbar_info['pad'] * size
```

File 9 - _docstring.py:

```
1: (0)              import inspect
2: (0)              from . import _api
3: (0)              def kwarg_doc(text):
4: (4)                  """
5: (4)                  Decorator for defining the kwdoc documentation of artist properties.
6: (4)                  This decorator can be applied to artist property setter methods.
7: (4)                  The given text is stored in a private attribute ``_kwarg_doc`` on
8: (4)                  the method.  It is used to overwrite auto-generated documentation
9: (4)                  in the *kwdoc list* for artists. The kwdoc list is used to document
10: (4)                 ``**kwargs`` when they are properties of an artist. See e.g. the
11: (4)                 ``**kwargs`` section in `.Axes.text`.
12: (4)                 The text should contain the supported types, as well as the default
13: (4)                 value if applicable, e.g.:
14: (8)                     @_docstring.kwarg_doc("bool, default: :rc:`text.usetex`")
15: (8)                     def set_usetex(self, usetex):
16: (4)                 See Also
17: (4)                 --------
18: (4)                 matplotlib.artist.kwdoc
19: (4)                 """
20: (4)                 def decorator(func):
21: (8)                     func._kwarg_doc = text
22: (8)                     return func
23: (4)                 return decorator
24: (0)             class Substitution:
25: (4)                 """
26: (4)                 A decorator that performs %-substitution on an object's docstring.
27: (4)                 This decorator should be robust even if ``obj.__doc__`` is None (for
28: (4)                 example, if -OO was passed to the interpreter).
```

```
29: (4)                 Usage: construct a docstring.Substitution with a sequence or dictionary
30: (4)                 suitable for performing substitution; then decorate a suitable function
31: (4)                 with the constructed object, e.g.::
32: (8)                     sub_author_name = Substitution(author='Jason')
33: (8)                     @sub_author_name
34: (8)                     def some_function(x):
35: (12)                        "%(author)s wrote this function"
36: (4)                 One can also use positional arguments::
37: (8)                     sub_first_last_names = Substitution('Edgar Allen', 'Poe')
38: (8)                     @sub_first_last_names
39: (8)                     def some_function(x):
40: (12)                        "%s %s wrote the Raven"
41: (4)                 """
42: (4)                 def __init__(self, *args, **kwargs):
43: (8)                     if args and kwargs:
44: (12)                        raise TypeError("Only positional or keyword args are allowed")
45: (8)                     self.params = args or kwargs
46: (4)                 def __call__(self, func):
47: (8)                     if func.__doc__:
48: (12)                        func.__doc__ = inspect.cleandoc(func.__doc__) % self.params
49: (8)                     return func
50: (4)                 def update(self, *args, **kwargs):
51: (8)                     """
52: (8)                     Update ``self.params`` (which must be a dict) with the supplied args.
53: (8)                     """
54: (8)                     self.params.update(*args, **kwargs)
55: (0)             class _ArtistKwdocLoader(dict):
56: (4)                 def __missing__(self, key):
57: (8)                     if not key.endswith(":kwdoc"):
58: (12)                        raise KeyError(key)
59: (8)                     name = key[:-len(":kwdoc")]
60: (8)                     from matplotlib.artist import Artist, kwdoc
61: (8)                     try:
62: (12)                        cls, = [cls for cls in _api.recursive_subclasses(Artist)
63: (20)                                if cls.__name__ == name]
64: (8)                     except ValueError as e:
65: (12)                        raise KeyError(key) from e
66: (8)                     return self.setdefault(key, kwdoc(cls))
67: (0)             class _ArtistPropertiesSubstitution(Substitution):
68: (4)                 """
69: (4)                 A `.Substitution` with two additional features:
70: (4)                 - Substitutions of the form ``%(classname:kwdoc)s`` (ending with the
71: (6)                   literal ":kwdoc" suffix) trigger lookup of an Artist subclass with the
72: (6)                   given *classname*, and are substituted with the `.kwdoc` of that class.
73: (4)                 - Decorating a class triggers substitution both on the class docstring and
74: (6)                   on the class' ``__init__`` docstring (which is a commonly required
75: (6)                   pattern for Artist subclasses).
76: (4)                 """
77: (4)                 def __init__(self):
78: (8)                     self.params = _ArtistKwdocLoader()
79: (4)                 def __call__(self, obj):
80: (8)                     super().__call__(obj)
81: (8)                     if isinstance(obj, type) and obj.__init__ != object.__init__:
82: (12)                        self(obj.__init__)
83: (8)                     return obj
84: (0)             def copy(source):
85: (4)                 """Copy a docstring from another source function (if present)."""
86: (4)                 def do_copy(target):
87: (8)                     if source.__doc__:
88: (12)                        target.__doc__ = source.__doc__
89: (8)                     return target
90: (4)                 return do_copy
91: (0)             dedent_interpd = interpd = _ArtistPropertiesSubstitution()
```

File 10 - _enums.py:

```
1: (0)              """
```

```
2: (0)              Enums representing sets of strings that Matplotlib uses as input parameters.
3: (0)              Matplotlib often uses simple data types like strings or tuples to define a
4: (0)              concept; e.g. the line capstyle can be specified as one of 'butt', 'round',
5: (0)              or 'projecting'. The classes in this module are used internally and serve to
6: (0)              document these concepts formally.
7: (0)              As an end-user you will not use these classes directly, but only the values
8: (0)              they define.
9: (0)              """
10: (0)             from enum import Enum, auto
11: (0)             from matplotlib import _docstring
12: (0)             class _AutoStringNameEnum(Enum):
13: (4)                 """Automate the ``name = 'name'`` part of making a (str, Enum)."""
14: (4)                 def _generate_next_value_(name, start, count, last_values):
15: (8)                     return name
16: (4)                 def __hash__(self):
17: (8)                     return str(self).__hash__()
18: (0)             class JoinStyle(str, _AutoStringNameEnum):
19: (4)                 """
20: (4)                 Define how the connection between two line segments is drawn.
21: (4)                 For a visual impression of each *JoinStyle*, `view these docs online
22: (4)                 <JoinStyle>`, or run `JoinStyle.demo`.
23: (4)                 Lines in Matplotlib are typically defined by a 1D `~.path.Path` and a
24: (4)                 finite ``linewidth``, where the underlying 1D `~.path.Path` represents the
25: (4)                 center of the stroked line.
26: (4)                 By default, `~.backend_bases.GraphicsContextBase` defines the boundaries
of
27: (4)                 a stroked line to simply be every point within some radius,
28: (4)                 ``linewidth/2``, away from any point of the center line. However, this
29: (4)                 results in corners appearing "rounded", which may not be the desired
30: (4)                 behavior if you are drawing, for example, a polygon or pointed star.
31: (4)                 **Supported values:**
32: (4)                 .. rst-class:: value-list
33: (8)                     'miter'
34: (12)                        the "arrow-tip" style. Each boundary of the filled-in area will
35: (12)                        extend in a straight line parallel to the tangent vector of the
36: (12)                        centerline at the point it meets the corner, until they meet in a
37: (12)                        sharp point.
38: (8)                     'round'
39: (12)                        stokes every point within a radius of ``linewidth/2`` of the
center
40: (12)                        lines.
41: (8)                     'bevel'
42: (12)                        the "squared-off" style. It can be thought of as a rounded corner
43: (12)                        where the "circular" part of the corner has been cut off.
44: (4)                 .. note::
45: (8)                     Very long miter tips are cut off (to form a *bevel*) after a
46: (8)                     backend-dependent limit called the "miter limit", which specifies the
47: (8)                     maximum allowed ratio of miter length to line width. For example, the
48: (8)                     PDF backend uses the default value of 10 specified by the PDF
standard,
49: (8)                     while the SVG backend does not even specify the miter limit, resulting
50: (8)                     in a default value of 4 per the SVG specification. Matplotlib does not
51: (8)                     currently allow the user to adjust this parameter.
52: (8)                     A more detailed description of the effect of a miter limit can be
found
53: (8)                     in the `Mozilla Developer Docs
54: (8)                     <https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/stroke-
miterlimit>`_
55: (4)                 .. plot::
56: (8)                     :alt: Demo of possible JoinStyle's
57: (8)                     from matplotlib._enums import JoinStyle
58: (8)                     JoinStyle.demo()
59: (4)                 """
60: (4)                 miter = auto()
61: (4)                 round = auto()
62: (4)                 bevel = auto()
63: (4)                 @staticmethod
64: (4)                 def demo():
65: (8)                     """Demonstrate how each JoinStyle looks for various join angles."""
```

```
66: (8)                     import numpy as np
67: (8)                     import matplotlib.pyplot as plt
68: (8)                     def plot_angle(ax, x, y, angle, style):
69: (12)                        phi = np.radians(angle)
70: (12)                        xx = [x + .5, x, x + .5*np.cos(phi)]
71: (12)                        yy = [y, y, y + .5*np.sin(phi)]
72: (12)                        ax.plot(xx, yy, lw=12, color='tab:blue', solid_joinstyle=style)
73: (12)                        ax.plot(xx, yy, lw=1, color='black')
74: (12)                        ax.plot(xx[1], yy[1], 'o', color='tab:red', markersize=3)
75: (8)                     fig, ax = plt.subplots(figsize=(5, 4), constrained_layout=True)
76: (8)                     ax.set_title('Join style')
77: (8)                     for x, style in enumerate(['miter', 'round', 'bevel']):
78: (12)                        ax.text(x, 5, style)
79: (12)                        for y, angle in enumerate([20, 45, 60, 90, 120]):
80: (16)                            plot_angle(ax, x, y, angle, style)
81: (16)                            if x == 0:
82: (20)                                ax.text(-1.3, y, f'{angle} degrees')
83: (8)                     ax.set_xlim(-1.5, 2.75)
84: (8)                     ax.set_ylim(-.5, 5.5)
85: (8)                     ax.set_axis_off()
86: (8)                     fig.show()
87: (0)             JoinStyle.input_description = "{" \
88: (8)                     + ", ".join([f"'{js.name}'" for js in JoinStyle]) \
89: (8)                     + "}"
90: (0)             class CapStyle(str, _AutoStringNameEnum):
91: (4)                 r"""
92: (4)                 Define how the two endpoints (caps) of an unclosed line are drawn.
93: (4)                 How to draw the start and end points of lines that represent a closed
curve
94: (4)                 (i.e. that end in a `~.path.Path.CLOSEPOLY`) is controlled by the line's
95: (4)                 `JoinStyle`. For all other lines, how the start and end points are drawn
is
96: (4)                 controlled by the *CapStyle*.
97: (4)                 For a visual impression of each *CapStyle*, `view these docs online
98: (4)                 <CapStyle>` or run `CapStyle.demo`.
99: (4)                 By default, `~.backend_bases.GraphicsContextBase` draws a stroked line as
100: (4)                squared off at its endpoints.
101: (4)                **Supported values:**
102: (4)                .. rst-class:: value-list
103: (8)                    'butt'
104: (12)                       the line is squared off at its endpoint.
105: (8)                    'projecting'
106: (12)                       the line is squared off as in *butt*, but the filled in area
107: (12)                       extends beyond the endpoint a distance of ``linewidth/2``.
108: (8)                    'round'
109: (12)                       like *butt*, but a semicircular cap is added to the end of the
110: (12)                       line, of radius ``linewidth/2``.
111: (4)                .. plot::
112: (8)                    :alt: Demo of possible CapStyle's
113: (8)                    from matplotlib._enums import CapStyle
114: (8)                    CapStyle.demo()
115: (4)                """
116: (4)                butt = auto()
117: (4)                projecting = auto()
118: (4)                round = auto()
119: (4)                @staticmethod
120: (4)                def demo():
121: (8)                    """Demonstrate how each CapStyle looks for a thick line segment."""
122: (8)                    import matplotlib.pyplot as plt
123: (8)                    fig = plt.figure(figsize=(4, 1.2))
124: (8)                    ax = fig.add_axes([0, 0, 1, 0.8])
125: (8)                    ax.set_title('Cap style')
126: (8)                    for x, style in enumerate(['butt', 'round', 'projecting']):
127: (12)                       ax.text(x+0.25, 0.85, style, ha='center')
128: (12)                       xx = [x, x+0.5]
129: (12)                       yy = [0, 0]
130: (12)                       ax.plot(xx, yy, lw=12, color='tab:blue', solid_capstyle=style)
131: (12)                       ax.plot(xx, yy, lw=1, color='black')
132: (12)                       ax.plot(xx, yy, 'o', color='tab:red', markersize=3)
```

```
133: (8)                    ax.set_ylim(-.5, 1.5)
134: (8)                    ax.set_axis_off()
135: (8)                    fig.show()
136: (0)            CapStyle.input_description = "{" \
137: (8)                    + ", ".join([f"'{cs.name}'" for cs in CapStyle]) \
138: (8)                    + "}"
139: (0)            _docstring.interpd.update({'JoinStyle': JoinStyle.input_description,
140: (26)                                      'CapStyle': CapStyle.input_description})
```

File 11 - _fontconfig_pattern.py:

```
1: (0)              """
2: (0)              A module for parsing and generating `fontconfig patterns`_.
3: (0)              .. _fontconfig patterns:
4: (3)                 https://www.freedesktop.org/software/fontconfig/fontconfig-user.html
5: (0)              """
6: (0)              from functools import lru_cache, partial
7: (0)              import re
8: (0)              from pyparsing import (
9: (4)                  Group, Optional, ParseException, Regex, StringEnd, Suppress, ZeroOrMore,
oneOf)
10: (0)             _family_punc = r'\\\-:,'
11: (0)             _family_unescape = partial(re.compile(r'\\(?=[%s])' % _family_punc).sub, '')
12: (0)             _family_escape = partial(re.compile(r'(?=[%s])' % _family_punc).sub, r'\\')
13: (0)             _value_punc = r'\\=_:,'
14: (0)             _value_unescape = partial(re.compile(r'\\(?=[%s])' % _value_punc).sub, '')
15: (0)             _value_escape = partial(re.compile(r'(?=[%s])' % _value_punc).sub, r'\\')
16: (0)             _CONSTANTS = {
17: (4)                 'thin':           ('weight', 'light'),
18: (4)                 'extralight':     ('weight', 'light'),
19: (4)                 'ultralight':     ('weight', 'light'),
20: (4)                 'light':          ('weight', 'light'),
21: (4)                 'book':           ('weight', 'book'),
22: (4)                 'regular':        ('weight', 'regular'),
23: (4)                 'normal':         ('weight', 'normal'),
24: (4)                 'medium':         ('weight', 'medium'),
25: (4)                 'demibold':       ('weight', 'demibold'),
26: (4)                 'semibold':       ('weight', 'semibold'),
27: (4)                 'bold':           ('weight', 'bold'),
28: (4)                 'extrabold':      ('weight', 'extra bold'),
29: (4)                 'black':          ('weight', 'black'),
30: (4)                 'heavy':          ('weight', 'heavy'),
31: (4)                 'roman':          ('slant', 'normal'),
32: (4)                 'italic':         ('slant', 'italic'),
33: (4)                 'oblique':        ('slant', 'oblique'),
34: (4)                 'ultracondensed': ('width', 'ultra-condensed'),
35: (4)                 'extracondensed': ('width', 'extra-condensed'),
36: (4)                 'condensed':      ('width', 'condensed'),
37: (4)                 'semicondensed':  ('width', 'semi-condensed'),
38: (4)                 'expanded':       ('width', 'expanded'),
39: (4)                 'extraexpanded':  ('width', 'extra-expanded'),
40: (4)                 'ultraexpanded':  ('width', 'ultra-expanded'),
41: (0)             }
42: (0)             @lru_cache  # The parser instance is a singleton.
43: (0)             def _make_fontconfig_parser():
44: (4)                 def comma_separated(elem):
45: (8)                     return elem + ZeroOrMore(Suppress(",") + elem)
46: (4)                 family = Regex(fr"([^{_family_punc}]|(\\[{_family_punc}]))*")
47: (4)                 size = Regex(r"([0-9]+\.?[0-9]*|\.[0-9]+)")
48: (4)                 name = Regex(r"[a-z]+")
49: (4)                 value = Regex(fr"([^{_value_punc}]|(\\[{_value_punc}]))*")
50: (4)                 prop = Group((name + Suppress("=") + comma_separated(value)) |
oneOf(_CONSTANTS))
51: (4)                 return (
52: (8)                     Optional(comma_separated(family)("families"))
53: (8)                     + Optional("-" + comma_separated(size)("sizes"))
54: (8)                     + ZeroOrMore(":" + prop("properties*"))
```

```
55: (8)                     + StringEnd()
56: (4)                 )
57: (0)             @lru_cache
58: (0)             def parse_fontconfig_pattern(pattern):
59: (4)                 """
60: (4)                 Parse a fontconfig *pattern* into a dict that can initialize a
61: (4)                 `.font_manager.FontProperties` object.
62: (4)                 """
63: (4)                 parser = _make_fontconfig_parser()
64: (4)                 try:
65: (8)                     parse = parser.parseString(pattern)
66: (4)                 except ParseException as err:
67: (8)                     raise ValueError("\n" + ParseException.explain(err, 0)) from None
68: (4)                 parser.resetCache()
69: (4)                 props = {}
70: (4)                 if "families" in parse:
71: (8)                     props["family"] = [*map(_family_unescape, parse["families"])]
72: (4)                 if "sizes" in parse:
73: (8)                     props["size"] = [*parse["sizes"]]
74: (4)                 for prop in parse.get("properties", []):
75: (8)                     if len(prop) == 1:
76: (12)                        prop = _CONSTANTS[prop[0]]
77: (8)                     k, *v = prop
78: (8)                     props.setdefault(k, []).extend(map(_value_unescape, v))
79: (4)                 return props
80: (0)             def generate_fontconfig_pattern(d):
81: (4)                 """Convert a `.FontProperties` to a fontconfig pattern string."""
82: (4)                 kvs = [(k, getattr(d, f"get_{k}")())
83: (11)                       for k in ["style", "variant", "weight", "stretch", "file", "size"]]
84: (4)                 return (",".join(_family_escape(f) for f in d.get_family())
85: (12)                        + ":".join(f":{k}={_value_escape(str(v))}"
86: (22)                                   for k, v in kvs if v is not None))
```

File 12 - _internal_utils.py:

```
1: (0)              """
2: (0)              Internal debugging utilities, that are not expected to be used in the rest of
3: (0)              the codebase.
4: (0)              WARNING: Code in this module may change without prior notice!
5: (0)              """
6: (0)              from io import StringIO
7: (0)              from pathlib import Path
8: (0)              import subprocess
9: (0)              from matplotlib.transforms import TransformNode
10: (0)             def graphviz_dump_transform(transform, dest, *, highlight=None):
11: (4)                 """
12: (4)                 Generate a graphical representation of the transform tree for *transform*
13: (4)                 using the :program:`dot` program (which this function depends on).  The
14: (4)                 output format (png, dot, etc.) is determined from the suffix of *dest*.
15: (4)                 Parameters
16: (4)                 ----------
17: (4)                 transform : `~matplotlib.transform.Transform`
18: (8)                     The represented transform.
19: (4)                 dest : str
20: (8)                     Output filename.  The extension must be one of the formats supported
21: (8)                     by :program:`dot`, e.g. png, svg, dot, ...
22: (8)                     (see https://www.graphviz.org/doc/info/output.html).
23: (4)                 highlight : list of `~matplotlib.transform.Transform` or None
24: (8)                     The transforms in the tree to be drawn in bold.
25: (8)                     If *None*, *transform* is highlighted.
26: (4)                 """
27: (4)                 if highlight is None:
28: (8)                     highlight = [transform]
29: (4)                 seen = set()
30: (4)                 def recurse(root, buf):
31: (8)                     if id(root) in seen:
32: (12)                        return
```

```
33: (8)                     seen.add(id(root))
34: (8)                     props = {}
35: (8)                     label = type(root).__name__
36: (8)                     if root._invalid:
37: (12)                        label = f'[{label}]'
38: (8)                     if root in highlight:
39: (12)                        props['style'] = 'bold'
40: (8)                     props['shape'] = 'box'
41: (8)                     props['label'] = '"%s"' % label
42: (8)                     props = ' '.join(map('{0[0]}={0[1]}'.format, props.items()))
43: (8)                     buf.write(f'{id(root)} [{props}];\n')
44: (8)                     for key, val in vars(root).items():
45: (12)                        if isinstance(val, TransformNode) and id(root) in val._parents:
46: (16)                            buf.write(f'"{id(root)}" -> "{id(val)}" '
47: (26)                                      f'[label="{key}", fontsize=10];\n')
48: (16)                            recurse(val, buf)
49: (4)                 buf = StringIO()
50: (4)                 buf.write('digraph G {\n')
51: (4)                 recurse(transform, buf)
52: (4)                 buf.write('}\n')
53: (4)                 subprocess.run(
54: (8)                     ['dot', '-T', Path(dest).suffix[1:], '-o', dest],
55: (8)                     input=buf.getvalue().encode('utf-8'), check=True)

----------------------------------------

File 13 - _layoutgrid.py:

1: (0)              """
2: (0)              A layoutgrid is a nrows by ncols set of boxes, meant to be used by
3: (0)              `._constrained_layout`, each box is analogous to a subplotspec element of
4: (0)              a gridspec.
5: (0)              Each box is defined by left[ncols], right[ncols], bottom[nrows] and
top[nrows],
6: (0)              and by two editable margins for each side.  The main margin gets its value
7: (0)              set by the size of ticklabels, titles, etc on each Axes that is in the figure.
8: (0)              The outer margin is the padding around the Axes, and space for any
9: (0)              colorbars.
10: (0)             The "inner" widths and heights of these boxes are then constrained to be the
11: (0)             same (relative the values of `width_ratios[ncols]` and
`height_ratios[nrows]`).
12: (0)             The layoutgrid is then constrained to be contained within a parent layoutgrid,
13: (0)             its column(s) and row(s) specified when it is created.
14: (0)             """
15: (0)             import itertools
16: (0)             import kiwisolver as kiwi
17: (0)             import logging
18: (0)             import numpy as np
19: (0)             import matplotlib as mpl
20: (0)             import matplotlib.patches as mpatches
21: (0)             from matplotlib.transforms import Bbox
22: (0)             _log = logging.getLogger(__name__)
23: (0)             class LayoutGrid:
24: (4)                 """
25: (4)                 Analogous to a gridspec, and contained in another LayoutGrid.
26: (4)                 """
27: (4)                 def __init__(self, parent=None, parent_pos=(0, 0),
28: (17)                             parent_inner=False, name='', ncols=1, nrows=1,
29: (17)                             h_pad=None, w_pad=None, width_ratios=None,
30: (17)                             height_ratios=None):
31: (8)                     Variable = kiwi.Variable
32: (8)                     self.parent_pos = parent_pos
33: (8)                     self.parent_inner = parent_inner
34: (8)                     self.name = name + seq_id()
35: (8)                     if isinstance(parent, LayoutGrid):
36: (12)                        self.name = f'{parent.name}.{self.name}'
37: (8)                     self.nrows = nrows
38: (8)                     self.ncols = ncols
39: (8)                     self.height_ratios = np.atleast_1d(height_ratios)
```

```
(8)                     if height_ratios is None:
(12)                        self.height_ratios = np.ones(nrows)
(8)                     self.width_ratios = np.atleast_1d(width_ratios)
(8)                     if width_ratios is None:
(12)                        self.width_ratios = np.ones(ncols)
(8)                     sn = self.name + '_'
(8)                     if not isinstance(parent, LayoutGrid):
(12)                        self.solver = kiwi.Solver()
(8)                     else:
(12)                        parent.add_child(self, *parent_pos)
(12)                        self.solver = parent.solver
(8)                     self.artists = np.empty((nrows, ncols), dtype=object)
(8)                     self.children = np.empty((nrows, ncols), dtype=object)
(8)                     self.margins = {}
(8)                     self.margin_vals = {}
(8)                     for todo in ['left', 'right', 'leftcb', 'rightcb']:
(12)                        self.margin_vals[todo] = np.zeros(ncols)
(8)                     sol = self.solver
(8)                     self.lefts = [Variable(f'{sn}lefts[{i}]') for i in range(ncols)]
(8)                     self.rights = [Variable(f'{sn}rights[{i}]') for i in range(ncols)]
(8)                     for todo in ['left', 'right', 'leftcb', 'rightcb']:
(12)                        self.margins[todo] = [Variable(f'{sn}margins[{todo}][{i}]')
(34)                                              for i in range(ncols)]
(12)                        for i in range(ncols):
(16)                            sol.addEditVariable(self.margins[todo][i], 'strong')
(8)                     for todo in ['bottom', 'top', 'bottomcb', 'topcb']:
(12)                        self.margins[todo] = np.empty((nrows), dtype=object)
(12)                        self.margin_vals[todo] = np.zeros(nrows)
(8)                     self.bottoms = [Variable(f'{sn}bottoms[{i}]') for i in range(nrows)]
(8)                     self.tops = [Variable(f'{sn}tops[{i}]') for i in range(nrows)]
(8)                     for todo in ['bottom', 'top', 'bottomcb', 'topcb']:
(12)                        self.margins[todo] = [Variable(f'{sn}margins[{todo}][{i}]')
(34)                                              for i in range(nrows)]
(12)                        for i in range(nrows):
(16)                            sol.addEditVariable(self.margins[todo][i], 'strong')
(8)                     self.reset_margins()
(8)                     self.add_constraints(parent)
(8)                     self.h_pad = h_pad
(8)                     self.w_pad = w_pad
(4)                 def __repr__(self):
(8)                     str = f'LayoutBox: {self.name:25s} {self.nrows}x{self.ncols},\n'
(8)                     for i in range(self.nrows):
(12)                        for j in range(self.ncols):
(16)                            str += f'{i}, {j}: '\
(23)                                   f'L{self.lefts[j].value():1.3f}, ' \
(23)                                   f'B{self.bottoms[i].value():1.3f}, ' \
(23)                                   f'R{self.rights[j].value():1.3f}, ' \
(23)                                   f'T{self.tops[i].value():1.3f}, ' \
(23)                                   f'ML{self.margins["left"][j].value():1.3f}, ' \
(23)                                   f'MR{self.margins["right"][j].value():1.3f}, ' \
(23)                                   f'MB{self.margins["bottom"][i].value():1.3f}, ' \
(23)                                   f'MT{self.margins["top"][i].value():1.3f}, \n'
(8)                     return str
(4)                 def reset_margins(self):
(8)                     """
(8)                     Reset all the margins to zero.  Must do this after changing
(8)                     figure size, for instance, because the relative size of the
(8)                     axes labels etc changes.
(8)                     """
(8)                     for todo in ['left', 'right', 'bottom', 'top',
(21)                                 'leftcb', 'rightcb', 'bottomcb', 'topcb']:
(12)                        self.edit_margins(todo, 0.0)
(4)                 def add_constraints(self, parent):
(8)                     self.hard_constraints()
(8)                     self.parent_constraints(parent)
(8)                     self.grid_constraints()
(4)                 def hard_constraints(self):
(8)                     """
(8)                     These are the redundant constraints, plus ones that make the
```

```
109: (8)                    rest of the code easier.
110: (8)                    """
111: (8)                    for i in range(self.ncols):
112: (12)                       hc = [self.rights[i] >= self.lefts[i],
113: (18)                             (self.rights[i] - self.margins['right'][i] -
114: (20)                               self.margins['rightcb'][i] >=
115: (20)                               self.lefts[i] - self.margins['left'][i] -
116: (20)                               self.margins['leftcb'][i])
117: (18)                             ]
118: (12)                       for c in hc:
119: (16)                           self.solver.addConstraint(c | 'required')
120: (8)                    for i in range(self.nrows):
121: (12)                       hc = [self.tops[i] >= self.bottoms[i],
122: (18)                             (self.tops[i] - self.margins['top'][i] -
123: (20)                               self.margins['topcb'][i] >=
124: (20)                               self.bottoms[i] - self.margins['bottom'][i] -
125: (20)                               self.margins['bottomcb'][i])
126: (18)                             ]
127: (12)                       for c in hc:
128: (16)                           self.solver.addConstraint(c | 'required')
129: (4)                def add_child(self, child, i=0, j=0):
130: (8)                    self.children[np.ix_(np.atleast_1d(i), np.atleast_1d(j))] = child
131: (4)                def parent_constraints(self, parent):
132: (8)                    if not isinstance(parent, LayoutGrid):
133: (12)                       hc = [self.lefts[0] == parent[0],
134: (18)                             self.rights[-1] == parent[0] + parent[2],
135: (18)                             self.tops[0] == parent[1] + parent[3],
136: (18)                             self.bottoms[-1] == parent[1]]
137: (8)                    else:
138: (12)                       rows, cols = self.parent_pos
139: (12)                       rows = np.atleast_1d(rows)
140: (12)                       cols = np.atleast_1d(cols)
141: (12)                       left = parent.lefts[cols[0]]
142: (12)                       right = parent.rights[cols[-1]]
143: (12)                       top = parent.tops[rows[0]]
144: (12)                       bottom = parent.bottoms[rows[-1]]
145: (12)                       if self.parent_inner:
146: (16)                           left += parent.margins['left'][cols[0]]
147: (16)                           left += parent.margins['leftcb'][cols[0]]
148: (16)                           right -= parent.margins['right'][cols[-1]]
149: (16)                           right -= parent.margins['rightcb'][cols[-1]]
150: (16)                           top -= parent.margins['top'][rows[0]]
151: (16)                           top -= parent.margins['topcb'][rows[0]]
152: (16)                           bottom += parent.margins['bottom'][rows[-1]]
153: (16)                           bottom += parent.margins['bottomcb'][rows[-1]]
154: (12)                       hc = [self.lefts[0] == left,
155: (18)                             self.rights[-1] == right,
156: (18)                             self.tops[0] == top,
157: (18)                             self.bottoms[-1] == bottom]
158: (8)                    for c in hc:
159: (12)                       self.solver.addConstraint(c | 'required')
160: (4)                def grid_constraints(self):
161: (8)                    w = (self.rights[0] - self.margins['right'][0] -
162: (13)                         self.margins['rightcb'][0])
163: (8)                    w = (w - self.lefts[0] - self.margins['left'][0] -
164: (13)                         self.margins['leftcb'][0])
165: (8)                    w0 = w / self.width_ratios[0]
166: (8)                    for i in range(1, self.ncols):
167: (12)                       w = (self.rights[i] - self.margins['right'][i] -
168: (17)                            self.margins['rightcb'][i])
169: (12)                       w = (w - self.lefts[i] - self.margins['left'][i] -
170: (17)                            self.margins['leftcb'][i])
171: (12)                       c = (w == w0 * self.width_ratios[i])
172: (12)                       self.solver.addConstraint(c | 'strong')
173: (12)                       c = (self.rights[i - 1] == self.lefts[i])
174: (12)                       self.solver.addConstraint(c | 'strong')
175: (8)                    h = self.tops[0] - self.margins['top'][0] - self.margins['topcb'][0]
176: (8)                    h = (h - self.bottoms[0] - self.margins['bottom'][0] -
177: (13)                         self.margins['bottomcb'][0])
```

```
178: (8)                    h0 = h / self.height_ratios[0]
179: (8)                    for i in range(1, self.nrows):
180: (12)                       h = (self.tops[i] - self.margins['top'][i] -
181: (17)                            self.margins['topcb'][i])
182: (12)                       h = (h - self.bottoms[i] - self.margins['bottom'][i] -
183: (17)                            self.margins['bottomcb'][i])
184: (12)                       c = (h == h0 * self.height_ratios[i])
185: (12)                       self.solver.addConstraint(c | 'strong')
186: (12)                       c = (self.bottoms[i - 1] == self.tops[i])
187: (12)                       self.solver.addConstraint(c | 'strong')
188: (4)                def edit_margin(self, todo, size, cell):
189: (8)                    """
190: (8)                    Change the size of the margin for one cell.
191: (8)                    Parameters
192: (8)                    ----------
193: (8)                    todo : string (one of 'left', 'right', 'bottom', 'top')
194: (12)                       margin to alter.
195: (8)                    size : float
196: (12)                       Size of the margin.  If it is larger than the existing minimum it
197: (12)                       updates the margin size. Fraction of figure size.
198: (8)                    cell : int
199: (12)                       Cell column or row to edit.
200: (8)                    """
201: (8)                    self.solver.suggestValue(self.margins[todo][cell], size)
202: (8)                    self.margin_vals[todo][cell] = size
203: (4)                def edit_margin_min(self, todo, size, cell=0):
204: (8)                    """
205: (8)                    Change the minimum size of the margin for one cell.
206: (8)                    Parameters
207: (8)                    ----------
208: (8)                    todo : string (one of 'left', 'right', 'bottom', 'top')
209: (12)                       margin to alter.
210: (8)                    size : float
211: (12)                       Minimum size of the margin .  If it is larger than the
212: (12)                       existing minimum it updates the margin size. Fraction of
213: (12)                       figure size.
214: (8)                    cell : int
215: (12)                       Cell column or row to edit.
216: (8)                    """
217: (8)                    if size > self.margin_vals[todo][cell]:
218: (12)                       self.edit_margin(todo, size, cell)
219: (4)                def edit_margins(self, todo, size):
220: (8)                    """
221: (8)                    Change the size of all the margin of all the cells in the layout grid.
222: (8)                    Parameters
223: (8)                    ----------
224: (8)                    todo : string (one of 'left', 'right', 'bottom', 'top')
225: (12)                       margin to alter.
226: (8)                    size : float
227: (12)                       Size to set the margins.  Fraction of figure size.
228: (8)                    """
229: (8)                    for i in range(len(self.margin_vals[todo])):
230: (12)                       self.edit_margin(todo, size, i)
231: (4)                def edit_all_margins_min(self, todo, size):
232: (8)                    """
233: (8)                    Change the minimum size of all the margin of all
234: (8)                    the cells in the layout grid.
235: (8)                    Parameters
236: (8)                    ----------
237: (8)                    todo : {'left', 'right', 'bottom', 'top'}
238: (12)                       The margin to alter.
239: (8)                    size : float
240: (12)                       Minimum size of the margin.  If it is larger than the
241: (12)                       existing minimum it updates the margin size. Fraction of
242: (12)                       figure size.
243: (8)                    """
244: (8)                    for i in range(len(self.margin_vals[todo])):
245: (12)                       self.edit_margin_min(todo, size, i)
246: (4)                def edit_outer_margin_mins(self, margin, ss):
```

```
(8)                     """
(8)                     Edit all four margin minimums in one statement.
(8)                     Parameters
(8)                     ----------
(8)                     margin : dict
(12)                        size of margins in a dict with keys 'left', 'right', 'bottom',
(12)                        'top'
(8)                     ss : SubplotSpec
(12)                        defines the subplotspec these margins should be applied to
(8)                     """
(8)                     self.edit_margin_min('left', margin['left'], ss.colspan.start)
(8)                     self.edit_margin_min('leftcb', margin['leftcb'], ss.colspan.start)
(8)                     self.edit_margin_min('right', margin['right'], ss.colspan.stop - 1)
(8)                     self.edit_margin_min('rightcb', margin['rightcb'], ss.colspan.stop -
1)
(8)                     self.edit_margin_min('top', margin['top'], ss.rowspan.start)
(8)                     self.edit_margin_min('topcb', margin['topcb'], ss.rowspan.start)
(8)                     self.edit_margin_min('bottom', margin['bottom'], ss.rowspan.stop - 1)
(8)                     self.edit_margin_min('bottomcb', margin['bottomcb'],
(29)                                         ss.rowspan.stop - 1)
(4)                 def get_margins(self, todo, col):
(8)                     """Return the margin at this position"""
(8)                     return self.margin_vals[todo][col]
(4)                 def get_outer_bbox(self, rows=0, cols=0):
(8)                     """
(8)                     Return the outer bounding box of the subplot specs
(8)                     given by rows and cols.  rows and cols can be spans.
(8)                     """
(8)                     rows = np.atleast_1d(rows)
(8)                     cols = np.atleast_1d(cols)
(8)                     bbox = Bbox.from_extents(
(12)                        self.lefts[cols[0]].value(),
(12)                        self.bottoms[rows[-1]].value(),
(12)                        self.rights[cols[-1]].value(),
(12)                        self.tops[rows[0]].value())
(8)                     return bbox
(4)                 def get_inner_bbox(self, rows=0, cols=0):
(8)                     """
(8)                     Return the inner bounding box of the subplot specs
(8)                     given by rows and cols.  rows and cols can be spans.
(8)                     """
(8)                     rows = np.atleast_1d(rows)
(8)                     cols = np.atleast_1d(cols)
(8)                     bbox = Bbox.from_extents(
(12)                        (self.lefts[cols[0]].value() +
(16)                            self.margins['left'][cols[0]].value() +
(16)                            self.margins['leftcb'][cols[0]].value()),
(12)                        (self.bottoms[rows[-1]].value() +
(16)                            self.margins['bottom'][rows[-1]].value() +
(16)                            self.margins['bottomcb'][rows[-1]].value()),
(12)                        (self.rights[cols[-1]].value() -
(16)                            self.margins['right'][cols[-1]].value() -
(16)                            self.margins['rightcb'][cols[-1]].value()),
(12)                        (self.tops[rows[0]].value() -
(16)                            self.margins['top'][rows[0]].value() -
(16)                            self.margins['topcb'][rows[0]].value()
(8)                     )
(8)                     return bbox
(4)                 def get_bbox_for_cb(self, rows=0, cols=0):
(8)                     """
(8)                     Return the bounding box that includes the
(8)                     decorations but, *not* the colorbar...
(8)                     """
(8)                     rows = np.atleast_1d(rows)
(8)                     cols = np.atleast_1d(cols)
(8)                     bbox = Bbox.from_extents(
(12)                        (self.lefts[cols[0]].value() +
(16)                            self.margins['leftcb'][cols[0]].value()),
(12)                        (self.bottoms[rows[-1]].value() +
```

```
(16)                            self.margins['bottomcb'][rows[-1]].value()),
(12)                        (self.rights[cols[-1]].value() -
(16)                            self.margins['rightcb'][cols[-1]].value()),
(12)                        (self.tops[rows[0]].value() -
(16)                            self.margins['topcb'][rows[0]].value())
(8)                     )
(8)                     return bbox
(4)                 def get_left_margin_bbox(self, rows=0, cols=0):
(8)                     """
(8)                     Return the left margin bounding box of the subplot specs
(8)                     given by rows and cols.  rows and cols can be spans.
(8)                     """
(8)                     rows = np.atleast_1d(rows)
(8)                     cols = np.atleast_1d(cols)
(8)                     bbox = Bbox.from_extents(
(12)                        (self.lefts[cols[0]].value() +
(16)                            self.margins['leftcb'][cols[0]].value()),
(12)                        (self.bottoms[rows[-1]].value()),
(12)                        (self.lefts[cols[0]].value() +
(16)                            self.margins['leftcb'][cols[0]].value() +
(16)                            self.margins['left'][cols[0]].value()),
(12)                        (self.tops[rows[0]].value()))
(8)                     return bbox
(4)                 def get_bottom_margin_bbox(self, rows=0, cols=0):
(8)                     """
(8)                     Return the left margin bounding box of the subplot specs
(8)                     given by rows and cols.  rows and cols can be spans.
(8)                     """
(8)                     rows = np.atleast_1d(rows)
(8)                     cols = np.atleast_1d(cols)
(8)                     bbox = Bbox.from_extents(
(12)                        (self.lefts[cols[0]].value()),
(12)                        (self.bottoms[rows[-1]].value() +
(13)                         self.margins['bottomcb'][rows[-1]].value()),
(12)                        (self.rights[cols[-1]].value()),
(12)                        (self.bottoms[rows[-1]].value() +
(16)                            self.margins['bottom'][rows[-1]].value() +
(13)                         self.margins['bottomcb'][rows[-1]].value()
(13)                         ))
(8)                     return bbox
(4)                 def get_right_margin_bbox(self, rows=0, cols=0):
(8)                     """
(8)                     Return the left margin bounding box of the subplot specs
(8)                     given by rows and cols.  rows and cols can be spans.
(8)                     """
(8)                     rows = np.atleast_1d(rows)
(8)                     cols = np.atleast_1d(cols)
(8)                     bbox = Bbox.from_extents(
(12)                        (self.rights[cols[-1]].value() -
(16)                            self.margins['right'][cols[-1]].value() -
(16)                            self.margins['rightcb'][cols[-1]].value()),
(12)                        (self.bottoms[rows[-1]].value()),
(12)                        (self.rights[cols[-1]].value() -
(16)                            self.margins['rightcb'][cols[-1]].value()),
(12)                        (self.tops[rows[0]].value()))
(8)                     return bbox
(4)                 def get_top_margin_bbox(self, rows=0, cols=0):
(8)                     """
(8)                     Return the left margin bounding box of the subplot specs
(8)                     given by rows and cols.  rows and cols can be spans.
(8)                     """
(8)                     rows = np.atleast_1d(rows)
(8)                     cols = np.atleast_1d(cols)
(8)                     bbox = Bbox.from_extents(
(12)                        (self.lefts[cols[0]].value()),
(12)                        (self.tops[rows[0]].value() -
(16)                            self.margins['topcb'][rows[0]].value()),
(12)                        (self.rights[cols[-1]].value()),
(12)                        (self.tops[rows[0]].value() -
```

```
384: (16)                            self.margins['topcb'][rows[0]].value() -
385: (16)                            self.margins['top'][rows[0]].value()))
386: (8)                    return bbox
387: (4)                def update_variables(self):
388: (8)                    """
389: (8)                    Update the variables for the solver attached to this layoutgrid.
390: (8)                    """
391: (8)                    self.solver.updateVariables()
392: (0)            _layoutboxobjnum = itertools.count()
393: (0)            def seq_id():
394: (4)                """Generate a short sequential id for layoutbox objects."""
395: (4)                return '%06d' % next(_layoutboxobjnum)
396: (0)            def plot_children(fig, lg=None, level=0):
397: (4)                """Simple plotting to show where boxes are."""
398: (4)                if lg is None:
399: (8)                    _layoutgrids = fig.get_layout_engine().execute(fig)
400: (8)                    lg = _layoutgrids[fig]
401: (4)                colors = mpl.rcParams["axes.prop_cycle"].by_key()["color"]
402: (4)                col = colors[level]
403: (4)                for i in range(lg.nrows):
404: (8)                    for j in range(lg.ncols):
405: (12)                       bb = lg.get_outer_bbox(rows=i, cols=j)
406: (12)                       fig.add_artist(
407: (16)                           mpatches.Rectangle(bb.p0, bb.width, bb.height, linewidth=1,
408: (35)                                              edgecolor='0.7', facecolor='0.7',
409: (35)                                              alpha=0.2, transform=fig.transFigure,
410: (35)                                              zorder=-3))
411: (12)                       bbi = lg.get_inner_bbox(rows=i, cols=j)
412: (12)                       fig.add_artist(
413: (16)                           mpatches.Rectangle(bbi.p0, bbi.width, bbi.height, linewidth=2,
414: (35)                                              edgecolor=col, facecolor='none',
415: (35)                                              transform=fig.transFigure, zorder=-2))
416: (12)                       bbi = lg.get_left_margin_bbox(rows=i, cols=j)
417: (12)                       fig.add_artist(
418: (16)                           mpatches.Rectangle(bbi.p0, bbi.width, bbi.height, linewidth=0,
419: (35)                                              edgecolor='none', alpha=0.2,
420: (35)                                              facecolor=[0.5, 0.7, 0.5],
421: (35)                                              transform=fig.transFigure, zorder=-2))
422: (12)                       bbi = lg.get_right_margin_bbox(rows=i, cols=j)
423: (12)                       fig.add_artist(
424: (16)                           mpatches.Rectangle(bbi.p0, bbi.width, bbi.height, linewidth=0,
425: (35)                                              edgecolor='none', alpha=0.2,
426: (35)                                              facecolor=[0.7, 0.5, 0.5],
427: (35)                                              transform=fig.transFigure, zorder=-2))
428: (12)                       bbi = lg.get_bottom_margin_bbox(rows=i, cols=j)
429: (12)                       fig.add_artist(
430: (16)                           mpatches.Rectangle(bbi.p0, bbi.width, bbi.height, linewidth=0,
431: (35)                                              edgecolor='none', alpha=0.2,
432: (35)                                              facecolor=[0.5, 0.5, 0.7],
433: (35)                                              transform=fig.transFigure, zorder=-2))
434: (12)                       bbi = lg.get_top_margin_bbox(rows=i, cols=j)
435: (12)                       fig.add_artist(
436: (16)                           mpatches.Rectangle(bbi.p0, bbi.width, bbi.height, linewidth=0,
437: (35)                                              edgecolor='none', alpha=0.2,
438: (35)                                              facecolor=[0.7, 0.2, 0.7],
439: (35)                                              transform=fig.transFigure, zorder=-2))
440: (4)                for ch in lg.children.flat:
441: (8)                    if ch is not None:
442: (12)                       plot_children(fig, ch, level=level+1)
```

File 14 - _mathtext.py:

```
1: (0)              """
2: (0)              Implementation details for :mod:`.mathtext`.
3: (0)              """
4: (0)              from __future__ import annotations
5: (0)              import abc
```

```
import copy
import enum
import functools
import logging
import os
import re
import types
import unicodedata
import string
import typing as T
from typing import NamedTuple
import numpy as np
from pyparsing import (
    Empty, Forward, Literal, NotAny, oneOf, OneOrMore, Optional,
    ParseBaseException, ParseException, ParseExpression, ParseFatalException,
    ParserElement, ParseResults, QuotedString, Regex, StringEnd, ZeroOrMore,
    pyparsing_common, Group)
import matplotlib as mpl
from . import cbook
from ._mathtext_data import (
    latex_to_bakoma, stix_glyph_fixes, stix_virtual_fonts, tex2uni)
from .font_manager import FontProperties, findfont, get_font
from .ft2font import FT2Font, FT2Image, KERNING_DEFAULT
from packaging.version import parse as parse_version
from pyparsing import __version__ as pyparsing_version
if parse_version(pyparsing_version).major < 3:
    from pyparsing import nestedExpr as nested_expr
else:
    from pyparsing import nested_expr
if T.TYPE_CHECKING:
    from collections.abc import Iterable
    from .ft2font import Glyph
ParserElement.enablePackrat()
_log = logging.getLogger("matplotlib.mathtext")
def get_unicode_index(symbol: str) -> int:  # Publicly exported.
    r"""
    Return the integer index (from the Unicode table) of *symbol*.
    Parameters
    ----------
    symbol : str
        A single (Unicode) character, a TeX command (e.g. r'\pi') or a Type1
        symbol name (e.g. 'phi').
    """
    try:  # This will succeed if symbol is a single Unicode char
        return ord(symbol)
    except TypeError:
        pass
    try:  # Is symbol a TeX symbol (i.e. \alpha)
        return tex2uni[symbol.strip("\\")]
    except KeyError as err:
        raise ValueError(
            f"{symbol!r} is not a valid Unicode character or TeX/Type1 symbol"
            ) from err
class VectorParse(NamedTuple):
    """
    The namedtuple type returned by ``MathTextParser("path").parse(...)``.
    Attributes
    ----------
    width, height, depth : float
        The global metrics.
    glyphs : list
        The glyphs including their positions.
    rect : list
        The list of rectangles.
    """
    width: float
    height: float
    depth: float
    glyphs: list[tuple[FT2Font, float, int, float, float]]
```

```
(4)                 rects: list[tuple[float, float, float, float]]
(0)             VectorParse.__module__ = "matplotlib.mathtext"
(0)             class RasterParse(NamedTuple):
(4)                 """
(4)                 The namedtuple type returned by ``MathTextParser("agg").parse(...)``.
(4)                 Attributes
(4)                 ----------
(4)                 ox, oy : float
(8)                     The offsets are always zero.
(4)                 width, height, depth : float
(8)                     The global metrics.
(4)                 image : FT2Image
(8)                     A raster image.
(4)                 """
(4)                 ox: float
(4)                 oy: float
(4)                 width: float
(4)                 height: float
(4)                 depth: float
(4)                 image: FT2Image
(0)             RasterParse.__module__ = "matplotlib.mathtext"
(0)             class Output:
(4)                 r"""
(4)                 Result of `ship`\ping a box: lists of positioned glyphs and rectangles.
(4)                 This class is not exposed to end users, but converted to a `VectorParse`
or
(4)                 a `RasterParse` by `.MathTextParser.parse`.
(4)                 """
(4)                 def __init__(self, box: Box):
(8)                     self.box = box
(8)                     self.glyphs: list[tuple[float, float, FontInfo]] = []  # (ox, oy,
info)
(8)                     self.rects: list[tuple[float, float, float, float]] = []  # (x1, y1,
x2, y2)
(4)                 def to_vector(self) -> VectorParse:
(8)                     w, h, d = map(
(12)                        np.ceil, [self.box.width, self.box.height, self.box.depth])
(8)                     gs = [(info.font, info.fontsize, info.num, ox, h - oy + info.offset)
(14)                          for ox, oy, info in self.glyphs]
(8)                     rs = [(x1, h - y2, x2 - x1, y2 - y1)
(14)                          for x1, y1, x2, y2 in self.rects]
(8)                     return VectorParse(w, h + d, d, gs, rs)
(4)                 def to_raster(self, *, antialiased: bool) -> RasterParse:
(8)                     xmin = min([*[ox + info.metrics.xmin for ox, oy, info in self.glyphs],
(20)                            *[x1 for x1, y1, x2, y2 in self.rects], 0]) - 1
(8)                     ymin = min([*[oy - info.metrics.ymax for ox, oy, info in self.glyphs],
(20)                            *[y1 for x1, y1, x2, y2 in self.rects], 0]) - 1
(8)                     xmax = max([*[ox + info.metrics.xmax for ox, oy, info in self.glyphs],
(20)                            *[x2 for x1, y1, x2, y2 in self.rects], 0]) + 1
(8)                     ymax = max([*[oy - info.metrics.ymin for ox, oy, info in self.glyphs],
(20)                            *[y2 for x1, y1, x2, y2 in self.rects], 0]) + 1
(8)                     w = xmax - xmin
(8)                     h = ymax - ymin - self.box.depth
(8)                     d = ymax - ymin - self.box.height
(8)                     image = FT2Image(np.ceil(w), np.ceil(h + max(d, 0)))
(8)                     shifted = ship(self.box, (-xmin, -ymin))
(8)                     for ox, oy, info in shifted.glyphs:
(12)                        info.font.draw_glyph_to_bitmap(
(16)                            image, ox, oy - info.metrics.iceberg, info.glyph,
(16)                            antialiased=antialiased)
(8)                     for x1, y1, x2, y2 in shifted.rects:
(12)                        height = max(int(y2 - y1) - 1, 0)
(12)                        if height == 0:
(16)                            center = (y2 + y1) / 2
(16)                            y = int(center - (height + 1) / 2)
(12)                        else:
(16)                            y = int(y1)
(12)                        image.draw_rect_filled(int(x1), y, np.ceil(x2), y + height)
(8)                     return RasterParse(0, 0, w, h + d, d, image)
```

```
(0)            class FontMetrics(NamedTuple):
(4)                """
(4)                Metrics of a font.
(4)                Attributes
(4)                ----------
(4)                advance : float
(8)                    The advance distance (in points) of the glyph.
(4)                height : float
(8)                    The height of the glyph in points.
(4)                width : float
(8)                    The width of the glyph in points.
(4)                xmin, xmax, ymin, ymax : float
(8)                    The ink rectangle of the glyph.
(4)                iceberg : float
(8)                    The distance from the baseline to the top of the glyph. (This corresponds to
(8)                    TeX's definition of "height".)
(4)                slanted : bool
(8)                    Whether the glyph should be considered as "slanted" (currently used for kerning
(8)                    sub/superscripts).
(4)                """
(4)                advance: float
(4)                height: float
(4)                width: float
(4)                xmin: float
(4)                xmax: float
(4)                ymin: float
(4)                ymax: float
(4)                iceberg: float
(4)                slanted: bool
(0)            class FontInfo(NamedTuple):
(4)                font: FT2Font
(4)                fontsize: float
(4)                postscript_name: str
(4)                metrics: FontMetrics
(4)                num: int
(4)                glyph: Glyph
(4)                offset: float
(0)            class Fonts(abc.ABC):
(4)                """
(4)                An abstract base class for a system of fonts to use for mathtext.
(4)                The class must be able to take symbol keys and font file names and
(4)                return the character metrics.  It also delegates to a backend class
(4)                to do the actual drawing.
(4)                """
(4)                def __init__(self, default_font_prop: FontProperties, load_glyph_flags: int):
(8)                    """
(8)                    Parameters
(8)                    ----------
(8)                    default_font_prop : `~.font_manager.FontProperties`
(12)                       The default non-math font, or the base font for Unicode (generic)
(12)                       font rendering.
(8)                    load_glyph_flags : int
(12)                       Flags passed to the glyph loader (e.g. ``FT_Load_Glyph`` and
(12)                       ``FT_Load_Char`` for FreeType-based fonts).
(8)                    """
(8)                    self.default_font_prop = default_font_prop
(8)                    self.load_glyph_flags = load_glyph_flags
(4)                def get_kern(self, font1: str, fontclass1: str, sym1: str, fontsize1: float,
(17)                            font2: str, fontclass2: str, sym2: str, fontsize2: float,
(17)                            dpi: float) -> float:
(8)                    """
(8)                    Get the kerning distance for font between *sym1* and *sym2*.
(8)                    See `~.Fonts.get_metrics` for a detailed description of the parameters.
(8)                    """
```

```
(8)                 return 0.
(4)             def _get_font(self, font: str) -> FT2Font:
(8)                 raise NotImplementedError
(4)             def _get_info(self, font: str, font_class: str, sym: str, fontsize: float,
(18)                          dpi: float) -> FontInfo:
(8)                 raise NotImplementedError
(4)             def get_metrics(self, font: str, font_class: str, sym: str, fontsize:
float,
(20)                            dpi: float) -> FontMetrics:
(8)                 r"""
(8)                 Parameters
(8)                 ----------
(8)                 font : str
(12)                    One of the TeX font names: "tt", "it", "rm", "cal", "sf", "bf",
(12)                    "default", "regular", "bb", "frak", "scr".  "default" and
"regular"
(12)                    are synonyms and use the non-math font.
(8)                 font_class : str
(12)                    One of the TeX font names (as for *font*), but **not** "bb",
(12)                    "frak", or "scr".  This is used to combine two font classes.  The
(12)                    only supported combination currently is ``get_metrics("frak",
"bf",
(12)                    ...)``.
(8)                 sym : str
(12)                    A symbol in raw TeX form, e.g., "1", "x", or "\sigma".
(8)                 fontsize : float
(12)                    Font size in points.
(8)                 dpi : float
(12)                    Rendering dots-per-inch.
(8)                 Returns
(8)                 -------
(8)                 FontMetrics
(8)                 """
(8)                 info = self._get_info(font, font_class, sym, fontsize, dpi)
(8)                 return info.metrics
(4)             def render_glyph(self, output: Output, ox: float, oy: float, font: str,
(21)                             font_class: str, sym: str, fontsize: float, dpi: float) -
> None:
(8)                 """
(8)                 At position (*ox*, *oy*), draw the glyph specified by the remaining
(8)                 parameters (see `get_metrics` for their detailed description).
(8)                 """
(8)                 info = self._get_info(font, font_class, sym, fontsize, dpi)
(8)                 output.glyphs.append((ox, oy, info))
(4)             def render_rect_filled(self, output: Output,
(27)                                   x1: float, y1: float, x2: float, y2: float) ->
None:
(8)                 """
(8)                 Draw a filled rectangle from (*x1*, *y1*) to (*x2*, *y2*).
(8)                 """
(8)                 output.rects.append((x1, y1, x2, y2))
(4)             def get_xheight(self, font: str, fontsize: float, dpi: float) -> float:
(8)                 """
(8)                 Get the xheight for the given *font* and *fontsize*.
(8)                 """
(8)                 raise NotImplementedError()
(4)             def get_underline_thickness(self, font: str, fontsize: float, dpi: float)
-> float:
(8)                 """
(8)                 Get the line thickness that matches the given font.  Used as a
(8)                 base unit for drawing lines such as in a fraction or radical.
(8)                 """
(8)                 raise NotImplementedError()
(4)             def get_sized_alternatives_for_symbol(self, fontname: str,
(42)                                                   sym: str) -> list[tuple[str, str]]:
(8)                 """
(8)                 Override if your font provides multiple sizes of the same
(8)                 symbol.  Should return a list of symbols matching *sym* in
(8)                 various sizes.  The expression renderer will select the most
```

```
268: (8)                    appropriate size for a given situation from this list.
269: (8)                    """
270: (8)                    return [(fontname, sym)]
271: (0)            class TruetypeFonts(Fonts, metaclass=abc.ABCMeta):
272: (4)                """
273: (4)                A generic base class for all font setups that use Truetype fonts
274: (4)                (through FT2Font).
275: (4)                """
276: (4)                def __init__(self, default_font_prop: FontProperties, load_glyph_flags:
int):
277: (8)                    super().__init__(default_font_prop, load_glyph_flags)
278: (8)                    self._get_info = functools.cache(self._get_info)  # type:
ignore[method-assign]
279: (8)                    self._fonts = {}
280: (8)                    self.fontmap: dict[str | int, str] = {}
281: (8)                    filename = findfont(self.default_font_prop)
282: (8)                    default_font = get_font(filename)
283: (8)                    self._fonts['default'] = default_font
284: (8)                    self._fonts['regular'] = default_font
285: (4)                def _get_font(self, font: str | int) -> FT2Font:
286: (8)                    if font in self.fontmap:
287: (12)                       basename = self.fontmap[font]
288: (8)                    else:
289: (12)                       basename = T.cast(str, font)
290: (8)                    cached_font = self._fonts.get(basename)
291: (8)                    if cached_font is None and os.path.exists(basename):
292: (12)                       cached_font = get_font(basename)
293: (12)                       self._fonts[basename] = cached_font
294: (12)                       self._fonts[cached_font.postscript_name] = cached_font
295: (12)                       self._fonts[cached_font.postscript_name.lower()] = cached_font
296: (8)                    return T.cast(FT2Font, cached_font)  # FIXME: Not sure this is
guaranteed.
297: (4)                def _get_offset(self, font: FT2Font, glyph: Glyph, fontsize: float,
298: (20)                               dpi: float) -> float:
299: (8)                    if font.postscript_name == 'Cmex10':
300: (12)                       return (glyph.height / 64 / 2) + (fontsize/3 * dpi/72)
301: (8)                    return 0.
302: (4)                def _get_glyph(self, fontname: str, font_class: str,
303: (19)                              sym: str) -> tuple[FT2Font, int, bool]:
304: (8)                    raise NotImplementedError
305: (4)                def _get_info(self, fontname: str, font_class: str, sym: str, fontsize:
float,
306: (18)                             dpi: float) -> FontInfo:
307: (8)                    font, num, slanted = self._get_glyph(fontname, font_class, sym)
308: (8)                    font.set_size(fontsize, dpi)
309: (8)                    glyph = font.load_char(num, flags=self.load_glyph_flags)
310: (8)                    xmin, ymin, xmax, ymax = [val/64.0 for val in glyph.bbox]
311: (8)                    offset = self._get_offset(font, glyph, fontsize, dpi)
312: (8)                    metrics = FontMetrics(
313: (12)                       advance = glyph.linearHoriAdvance/65536.0,
314: (12)                       height  = glyph.height/64.0,
315: (12)                       width   = glyph.width/64.0,
316: (12)                       xmin    = xmin,
317: (12)                       xmax    = xmax,
318: (12)                       ymin    = ymin+offset,
319: (12)                       ymax    = ymax+offset,
320: (12)                       iceberg = glyph.horiBearingY/64.0 + offset,
321: (12)                       slanted = slanted
322: (12)                       )
323: (8)                    return FontInfo(
324: (12)                       font            = font,
325: (12)                       fontsize        = fontsize,
326: (12)                       postscript_name = font.postscript_name,
327: (12)                       metrics         = metrics,
328: (12)                       num             = num,
329: (12)                       glyph           = glyph,
330: (12)                       offset          = offset
331: (12)                       )
332: (4)                def get_xheight(self, fontname: str, fontsize: float, dpi: float) ->
```

```
float:
333: (8)                    font = self._get_font(fontname)
334: (8)                    font.set_size(fontsize, dpi)
335: (8)                    pclt = font.get_sfnt_table('pclt')
336: (8)                    if pclt is None:
337: (12)                       metrics = self.get_metrics(
338: (16)                           fontname, mpl.rcParams['mathtext.default'], 'x', fontsize,
dpi)
339: (12)                       return metrics.iceberg
340: (8)                    xHeight = (pclt['xHeight'] / 64.0) * (fontsize / 12.0) * (dpi / 100.0)
341: (8)                    return xHeight
342: (4)                def get_underline_thickness(self, font: str, fontsize: float, dpi: float)
-> float:
343: (8)                    return ((0.75 / 12.0) * fontsize * dpi) / 72.0
344: (4)                def get_kern(self, font1: str, fontclass1: str, sym1: str, fontsize1:
float,
345: (17)                            font2: str, fontclass2: str, sym2: str, fontsize2: float,
346: (17)                            dpi: float) -> float:
347: (8)                    if font1 == font2 and fontsize1 == fontsize2:
348: (12)                       info1 = self._get_info(font1, fontclass1, sym1, fontsize1, dpi)
349: (12)                       info2 = self._get_info(font2, fontclass2, sym2, fontsize2, dpi)
350: (12)                       font = info1.font
351: (12)                       return font.get_kerning(info1.num, info2.num, KERNING_DEFAULT) /
64
352: (8)                    return super().get_kern(font1, fontclass1, sym1, fontsize1,
353: (32)                                           font2, fontclass2, sym2, fontsize2, dpi)
354: (0)            class BakomaFonts(TruetypeFonts):
355: (4)                """
356: (4)                Use the Bakoma TrueType fonts for rendering.
357: (4)                Symbols are strewn about a number of font files, each of which has
358: (4)                its own proprietary 8-bit encoding.
359: (4)                """
360: (4)                _fontmap = {
361: (8)                    'cal': 'cmsy10',
362: (8)                    'rm':  'cmr10',
363: (8)                    'tt':  'cmtt10',
364: (8)                    'it':  'cmmi10',
365: (8)                    'bf':  'cmb10',
366: (8)                    'sf':  'cmss10',
367: (8)                    'ex':  'cmex10',
368: (4)                }
369: (4)                def __init__(self, default_font_prop: FontProperties, load_glyph_flags:
int):
370: (8)                    self._stix_fallback = StixFonts(default_font_prop, load_glyph_flags)
371: (8)                    super().__init__(default_font_prop, load_glyph_flags)
372: (8)                    for key, val in self._fontmap.items():
373: (12)                       fullpath = findfont(val)
374: (12)                       self.fontmap[key] = fullpath
375: (12)                       self.fontmap[val] = fullpath
376: (4)                _slanted_symbols = set(r"\int \oint".split())
377: (4)                def _get_glyph(self, fontname: str, font_class: str,
378: (19)                               sym: str) -> tuple[FT2Font, int, bool]:
379: (8)                    font = None
380: (8)                    if fontname in self.fontmap and sym in latex_to_bakoma:
381: (12)                       basename, num = latex_to_bakoma[sym]
382: (12)                       slanted = (basename == "cmmi10") or sym in self._slanted_symbols
383: (12)                       font = self._get_font(basename)
384: (8)                    elif len(sym) == 1:
385: (12)                       slanted = (fontname == "it")
386: (12)                       font = self._get_font(fontname)
387: (12)                       if font is not None:
388: (16)                           num = ord(sym)
389: (8)                    if font is not None and font.get_char_index(num) != 0:
390: (12)                       return font, num, slanted
391: (8)                    else:
392: (12)                       return self._stix_fallback._get_glyph(fontname, font_class, sym)
393: (4)                _size_alternatives = {
394: (8)                    '(':           [('rm', '('), ('ex', '\xa1'), ('ex', '\xb3'),
395: (24)                                   ('ex', '\xb5'), ('ex', '\xc3')],
```

```
(8)                 ')':          [('rm', ')'), ('ex', '\xa2'), ('ex', '\xb4'),
(24)                               ('ex', '\xb6'), ('ex', '\x21')],
(8)                 '{':          [('cal', '{'), ('ex', '\xa9'), ('ex', '\x6e'),
(24)                               ('ex', '\xbd'), ('ex', '\x28')],
(8)                 '}':          [('cal', '}'), ('ex', '\xaa'), ('ex', '\x6f'),
(24)                               ('ex', '\xbe'), ('ex', '\x29')],
(8)                 '[':          [('rm', '['), ('ex', '\xa3'), ('ex', '\x68'),
(24)                               ('ex', '\x22')],
(8)                 ']':          [('rm', ']'), ('ex', '\xa4'), ('ex', '\x69'),
(24)                               ('ex', '\x23')],
(8)                 r'\lfloor':   [('ex', '\xa5'), ('ex', '\x6a'),
(24)                               ('ex', '\xb9'), ('ex', '\x24')],
(8)                 r'\rfloor':   [('ex', '\xa6'), ('ex', '\x6b'),
(24)                               ('ex', '\xba'), ('ex', '\x25')],
(8)                 r'\lceil':    [('ex', '\xa7'), ('ex', '\x6c'),
(24)                               ('ex', '\xbb'), ('ex', '\x26')],
(8)                 r'\rceil':    [('ex', '\xa8'), ('ex', '\x6d'),
(24)                               ('ex', '\xbc'), ('ex', '\x27')],
(8)                 r'\langle':   [('ex', '\xad'), ('ex', '\x44'),
(24)                               ('ex', '\xbf'), ('ex', '\x2a')],
(8)                 r'\rangle':   [('ex', '\xae'), ('ex', '\x45'),
(24)                               ('ex', '\xc0'), ('ex', '\x2b')],
(8)                 r'\__sqrt__': [('ex', '\x70'), ('ex', '\x71'),
(24)                               ('ex', '\x72'), ('ex', '\x73')],
(8)                 r'\backslash': [('ex', '\xb2'), ('ex', '\x2f'),
(24)                               ('ex', '\xc2'), ('ex', '\x2d')],
(8)                 r'/':         [('rm', '/'), ('ex', '\xb1'), ('ex', '\x2e'),
(24)                               ('ex', '\xcb'), ('ex', '\x2c')],
(8)                 r'\widehat':  [('rm', '\x5e'), ('ex', '\x62'), ('ex', '\x63'),
(24)                               ('ex', '\x64')],
(8)                 r'\widetilde': [('rm', '\x7e'), ('ex', '\x65'), ('ex', '\x66'),
(24)                               ('ex', '\x67')],
(8)                 r'<':         [('cal', 'h'), ('ex', 'D')],
(8)                 r'>':         [('cal', 'i'), ('ex', 'E')]
(8)                 }
(4)             for alias, target in [(r'\leftparen', '('),
(26)                                  (r'\rightparent', ')'),
(26)                                  (r'\leftbrace', '{'),
(26)                                  (r'\rightbrace', '}'),
(26)                                  (r'\leftbracket', '['),
(26)                                  (r'\rightbracket', ']'),
(26)                                  (r'\{', '{'),
(26)                                  (r'\}', '}'),
(26)                                  (r'\[', '['),
(26)                                  (r'\]', ']')]:
(8)                 _size_alternatives[alias] = _size_alternatives[target]
(4)             def get_sized_alternatives_for_symbol(self, fontname: str,
(42)                                                  sym: str) -> list[tuple[str, str]]:
(8)                 return self._size_alternatives.get(sym, [(fontname, sym)])
(0)         class UnicodeFonts(TruetypeFonts):
(4)             """
(4)             An abstract base class for handling Unicode fonts.
(4)             While some reasonably complete Unicode fonts (such as DejaVu) may
(4)             work in some situations, the only Unicode font I'm aware of with a
(4)             complete set of math symbols is STIX.
(4)             This class will "fallback" on the Bakoma fonts when a required
(4)             symbol cannot be found in the font.
(4)             """
(4)             _cmr10_substitutions = {
(8)                 0x00D7: 0x00A3,  # Multiplication sign.
(8)                 0x2212: 0x00A1,  # Minus sign.
(4)             }
(4)             def __init__(self, default_font_prop: FontProperties, load_glyph_flags:
int):
(8)                 fallback_rc = mpl.rcParams['mathtext.fallback']
(8)                 font_cls: type[TruetypeFonts] | None = {
(12)                    'stix': StixFonts,
(12)                    'stixsans': StixSansFonts,
(12)                    'cm': BakomaFonts
```

```
464: (8)                    }.get(fallback_rc)
465: (8)                    self._fallback_font = (font_cls(default_font_prop, load_glyph_flags)
466: (31)                                          if font_cls else None)
467: (8)                    super().__init__(default_font_prop, load_glyph_flags)
468: (8)                    for texfont in "cal rm tt it bf sf bfit".split():
469: (12)                       prop = mpl.rcParams['mathtext.' + texfont]
470: (12)                       font = findfont(prop)
471: (12)                       self.fontmap[texfont] = font
472: (8)                    prop = FontProperties('cmex10')
473: (8)                    font = findfont(prop)
474: (8)                    self.fontmap['ex'] = font
475: (8)                    if isinstance(self._fallback_font, StixFonts):
476: (12)                       stixsizedaltfonts = {
477: (17)                            0: 'STIXGeneral',
478: (17)                            1: 'STIXSizeOneSym',
479: (17)                            2: 'STIXSizeTwoSym',
480: (17)                            3: 'STIXSizeThreeSym',
481: (17)                            4: 'STIXSizeFourSym',
482: (17)                            5: 'STIXSizeFiveSym'}
483: (12)                       for size, name in stixsizedaltfonts.items():
484: (16)                           fullpath = findfont(name)
485: (16)                           self.fontmap[size] = fullpath
486: (16)                           self.fontmap[name] = fullpath
487: (4)                _slanted_symbols = set(r"\int \oint".split())
488: (4)                def _map_virtual_font(self, fontname: str, font_class: str,
489: (26)                                     uniindex: int) -> tuple[str, int]:
490: (8)                    return fontname, uniindex
491: (4)                def _get_glyph(self, fontname: str, font_class: str,
492: (19)                              sym: str) -> tuple[FT2Font, int, bool]:
493: (8)                    try:
494: (12)                       uniindex = get_unicode_index(sym)
495: (12)                       found_symbol = True
496: (8)                    except ValueError:
497: (12)                       uniindex = ord('?')
498: (12)                       found_symbol = False
499: (12)                       _log.warning("No TeX to Unicode mapping for %a.", sym)
500: (8)                    fontname, uniindex = self._map_virtual_font(
501: (12)                       fontname, font_class, uniindex)
502: (8)                    new_fontname = fontname
503: (8)                    if found_symbol:
504: (12)                       if fontname == 'it' and uniindex < 0x10000:
505: (16)                           char = chr(uniindex)
506: (16)                           if (unicodedata.category(char)[0] != "L"
507: (24)                                   or unicodedata.name(char).startswith("GREEK
CAPITAL")):
508: (20)                               new_fontname = 'rm'
509: (12)                       slanted = (new_fontname == 'it') or sym in self._slanted_symbols
510: (12)                       found_symbol = False
511: (12)                       font = self._get_font(new_fontname)
512: (12)                       if font is not None:
513: (16)                           if (uniindex in self._cmr10_substitutions
514: (24)                                   and font.family_name == "cmr10"):
515: (20)                               font = get_font(
516: (24)                                   cbook._get_data_path("fonts/ttf/cmsy10.ttf"))
517: (20)                               uniindex = self._cmr10_substitutions[uniindex]
518: (16)                           glyphindex = font.get_char_index(uniindex)
519: (16)                           if glyphindex != 0:
520: (20)                               found_symbol = True
521: (8)                    if not found_symbol:
522: (12)                       if self._fallback_font:
523: (16)                           if (fontname in ('it', 'regular')
524: (24)                                   and isinstance(self._fallback_font, StixFonts)):
525: (20)                               fontname = 'rm'
526: (16)                           g = self._fallback_font._get_glyph(fontname, font_class, sym)
527: (16)                           family = g[0].family_name
528: (16)                           if family in list(BakomaFonts._fontmap.values()):
529: (20)                               family = "Computer Modern"
530: (16)                           _log.info("Substituting symbol %s from %s", sym, family)
531: (16)                           return g
```

```
532: (12)                        else:
533: (16)                            if (fontname in ('it', 'regular')
534: (24)                                    and isinstance(self, StixFonts)):
535: (20)                                return self._get_glyph('rm', font_class, sym)
536: (16)                            _log.warning("Font %r does not have a glyph for %a [U+%x], "
537: (29)                                         "substituting with a dummy symbol.",
538: (29)                                         new_fontname, sym, uniindex)
539: (16)                            font = self._get_font('rm')
540: (16)                            uniindex = 0xA4  # currency char, for lack of anything better
541: (16)                            slanted = False
542: (8)                    return font, uniindex, slanted
543: (4)                def get_sized_alternatives_for_symbol(self, fontname: str,
544: (42)                                                      sym: str) -> list[tuple[str, str]]:
545: (8)                    if self._fallback_font:
546: (12)                       return self._fallback_font.get_sized_alternatives_for_symbol(
547: (16)                           fontname, sym)
548: (8)                    return [(fontname, sym)]
549: (0)            class DejaVuFonts(UnicodeFonts, metaclass=abc.ABCMeta):
550: (4)                _fontmap: dict[str | int, str] = {}
551: (4)                def __init__(self, default_font_prop: FontProperties, load_glyph_flags:
int):
552: (8)                    if isinstance(self, DejaVuSerifFonts):
553: (12)                       self._fallback_font = StixFonts(default_font_prop,
load_glyph_flags)
554: (8)                    else:
555: (12)                       self._fallback_font = StixSansFonts(default_font_prop,
load_glyph_flags)
556: (8)                    self.bakoma = BakomaFonts(default_font_prop, load_glyph_flags)
557: (8)                    TruetypeFonts.__init__(self, default_font_prop, load_glyph_flags)
558: (8)                    self._fontmap.update({
559: (12)                       1: 'STIXSizeOneSym',
560: (12)                       2: 'STIXSizeTwoSym',
561: (12)                       3: 'STIXSizeThreeSym',
562: (12)                       4: 'STIXSizeFourSym',
563: (12)                       5: 'STIXSizeFiveSym',
564: (8)                    })
565: (8)                    for key, name in self._fontmap.items():
566: (12)                       fullpath = findfont(name)
567: (12)                       self.fontmap[key] = fullpath
568: (12)                       self.fontmap[name] = fullpath
569: (4)                def _get_glyph(self, fontname: str, font_class: str,
570: (19)                              sym: str) -> tuple[FT2Font, int, bool]:
571: (8)                    if sym == r'\prime':
572: (12)                       return self.bakoma._get_glyph(fontname, font_class, sym)
573: (8)                    else:
574: (12)                       uniindex = get_unicode_index(sym)
575: (12)                       font = self._get_font('ex')
576: (12)                       if font is not None:
577: (16)                           glyphindex = font.get_char_index(uniindex)
578: (16)                           if glyphindex != 0:
579: (20)                               return super()._get_glyph('ex', font_class, sym)
580: (12)                       return super()._get_glyph(fontname, font_class, sym)
581: (0)            class DejaVuSerifFonts(DejaVuFonts):
582: (4)                """
583: (4)                A font handling class for the DejaVu Serif fonts
584: (4)                If a glyph is not found it will fallback to Stix Serif
585: (4)                """
586: (4)                _fontmap = {
587: (8)                    'rm': 'DejaVu Serif',
588: (8)                    'it': 'DejaVu Serif:italic',
589: (8)                    'bf': 'DejaVu Serif:weight=bold',
590: (8)                    'bfit': 'DejaVu Serif:italic:bold',
591: (8)                    'sf': 'DejaVu Sans',
592: (8)                    'tt': 'DejaVu Sans Mono',
593: (8)                    'ex': 'DejaVu Serif Display',
594: (8)                    0:    'DejaVu Serif',
595: (4)                }
596: (0)            class DejaVuSansFonts(DejaVuFonts):
597: (4)                """
```

```
598: (4)                A font handling class for the DejaVu Sans fonts
599: (4)                If a glyph is not found it will fallback to Stix Sans
600: (4)                """
601: (4)                _fontmap = {
602: (8)                    'rm': 'DejaVu Sans',
603: (8)                    'it': 'DejaVu Sans:italic',
604: (8)                    'bf': 'DejaVu Sans:weight=bold',
605: (8)                    'bfit': 'DejaVu Sans:italic:bold',
606: (8)                    'sf': 'DejaVu Sans',
607: (8)                    'tt': 'DejaVu Sans Mono',
608: (8)                    'ex': 'DejaVu Sans Display',
609: (8)                    0:    'DejaVu Sans',
610: (4)                }
611: (0)            class StixFonts(UnicodeFonts):
612: (4)                """
613: (4)                A font handling class for the STIX fonts.
614: (4)                In addition to what UnicodeFonts provides, this class:
615: (4)                - supports "virtual fonts" which are complete alpha numeric
616: (6)                  character sets with different font styles at special Unicode
617: (6)                  code points, such as "Blackboard".
618: (4)                - handles sized alternative characters for the STIXSizeX fonts.
619: (4)                """
620: (4)                _fontmap: dict[str | int, str] = {
621: (8)                    'rm': 'STIXGeneral',
622: (8)                    'it': 'STIXGeneral:italic',
623: (8)                    'bf': 'STIXGeneral:weight=bold',
624: (8)                    'bfit': 'STIXGeneral:italic:bold',
625: (8)                    'nonunirm': 'STIXNonUnicode',
626: (8)                    'nonuniit': 'STIXNonUnicode:italic',
627: (8)                    'nonunibf': 'STIXNonUnicode:weight=bold',
628: (8)                    0: 'STIXGeneral',
629: (8)                    1: 'STIXSizeOneSym',
630: (8)                    2: 'STIXSizeTwoSym',
631: (8)                    3: 'STIXSizeThreeSym',
632: (8)                    4: 'STIXSizeFourSym',
633: (8)                    5: 'STIXSizeFiveSym',
634: (4)                }
635: (4)                _fallback_font = None
636: (4)                _sans = False
637: (4)                def __init__(self, default_font_prop: FontProperties, load_glyph_flags:
int):
638: (8)                    TruetypeFonts.__init__(self, default_font_prop, load_glyph_flags)
639: (8)                    for key, name in self._fontmap.items():
640: (12)                       fullpath = findfont(name)
641: (12)                       self.fontmap[key] = fullpath
642: (12)                       self.fontmap[name] = fullpath
643: (4)                def _map_virtual_font(self, fontname: str, font_class: str,
644: (26)                                     uniindex: int) -> tuple[str, int]:
645: (8)                    font_mapping = stix_virtual_fonts.get(fontname)
646: (8)                    if (self._sans and font_mapping is None
647: (16)                           and fontname not in ('regular', 'default')):
648: (12)                       font_mapping = stix_virtual_fonts['sf']
649: (12)                       doing_sans_conversion = True
650: (8)                    else:
651: (12)                       doing_sans_conversion = False
652: (8)                    if isinstance(font_mapping, dict):
653: (12)                       try:
654: (16)                           mapping = font_mapping[font_class]
655: (12)                       except KeyError:
656: (16)                           mapping = font_mapping['rm']
657: (8)                    elif isinstance(font_mapping, list):
658: (12)                       mapping = font_mapping
659: (8)                    else:
660: (12)                       mapping = None
661: (8)                    if mapping is not None:
662: (12)                       lo = 0
663: (12)                       hi = len(mapping)
664: (12)                       while lo < hi:
665: (16)                           mid = (lo+hi)//2
```

```
666: (16)                            range = mapping[mid]
667: (16)                            if uniindex < range[0]:
668: (20)                                hi = mid
669: (16)                            elif uniindex <= range[1]:
670: (20)                                break
671: (16)                            else:
672: (20)                                lo = mid + 1
673: (12)                        if range[0] <= uniindex <= range[1]:
674: (16)                            uniindex = uniindex - range[0] + range[3]
675: (16)                            fontname = range[2]
676: (12)                        elif not doing_sans_conversion:
677: (16)                            uniindex = 0x1
678: (16)                            fontname = mpl.rcParams['mathtext.default']
679: (8)                    if fontname in ('rm', 'it'):
680: (12)                        uniindex = stix_glyph_fixes.get(uniindex, uniindex)
681: (8)                    if fontname in ('it', 'rm', 'bf', 'bfit') and 0xe000 <= uniindex <=
0xf8ff:
682: (12)                        fontname = 'nonuni' + fontname
683: (8)                    return fontname, uniindex
684: (4)                @functools.cache
685: (4)                def get_sized_alternatives_for_symbol(  # type: ignore[override]
686: (12)                        self,
687: (12)                        fontname: str,
688: (12)                        sym: str) -> list[tuple[str, str]] | list[tuple[int, str]]:
689: (8)                    fixes = {
690: (12)                        '\\{': '{', '\\}': '}', '\\[': '[', '\\]': ']',
691: (12)                        '<': '\N{MATHEMATICAL LEFT ANGLE BRACKET}',
692: (12)                        '>': '\N{MATHEMATICAL RIGHT ANGLE BRACKET}',
693: (8)                    }
694: (8)                    sym = fixes.get(sym, sym)
695: (8)                    try:
696: (12)                        uniindex = get_unicode_index(sym)
697: (8)                    except ValueError:
698: (12)                        return [(fontname, sym)]
699: (8)                    alternatives = [(i, chr(uniindex)) for i in range(6)
700: (24)                                    if self._get_font(i).get_char_index(uniindex) != 0]
701: (8)                    if sym == r'\__sqrt__':
702: (12)                        alternatives = alternatives[:-1]
703: (8)                    return alternatives
704: (0)            class StixSansFonts(StixFonts):
705: (4)                """
706: (4)                A font handling class for the STIX fonts (that uses sans-serif
707: (4)                characters by default).
708: (4)                """
709: (4)                _sans = True
710: (0)            SHRINK_FACTOR   = 0.7
711: (0)            NUM_SIZE_LEVELS = 6
712: (0)            class FontConstantsBase:
713: (4)                """
714: (4)                A set of constants that controls how certain things, such as sub-
715: (4)                and superscripts are laid out.  These are all metrics that can't
716: (4)                be reliably retrieved from the font metrics in the font itself.
717: (4)                """
718: (4)                script_space: T.ClassVar[float] = 0.05
719: (4)                subdrop: T.ClassVar[float] = 0.4
720: (4)                sup1: T.ClassVar[float] = 0.7
721: (4)                sub1: T.ClassVar[float] = 0.3
722: (4)                sub2: T.ClassVar[float] = 0.5
723: (4)                delta: T.ClassVar[float] = 0.025
724: (4)                delta_slanted: T.ClassVar[float] = 0.2
725: (4)                delta_integral: T.ClassVar[float] = 0.1
726: (0)            class ComputerModernFontConstants(FontConstantsBase):
727: (4)                script_space = 0.075
728: (4)                subdrop = 0.2
729: (4)                sup1 = 0.45
730: (4)                sub1 = 0.2
731: (4)                sub2 = 0.3
732: (4)                delta = 0.075
733: (4)                delta_slanted = 0.3
```

```
(4)                 delta_integral = 0.3
(0)             class STIXFontConstants(FontConstantsBase):
(4)                 script_space = 0.1
(4)                 sup1 = 0.8
(4)                 sub2 = 0.6
(4)                 delta = 0.05
(4)                 delta_slanted = 0.3
(4)                 delta_integral = 0.3
(0)             class STIXSansFontConstants(FontConstantsBase):
(4)                 script_space = 0.05
(4)                 sup1 = 0.8
(4)                 delta_slanted = 0.6
(4)                 delta_integral = 0.3
(0)             class DejaVuSerifFontConstants(FontConstantsBase):
(4)                 pass
(0)             class DejaVuSansFontConstants(FontConstantsBase):
(4)                 pass
(0)             _font_constant_mapping = {
(4)                 'DejaVu Sans': DejaVuSansFontConstants,
(4)                 'DejaVu Sans Mono': DejaVuSansFontConstants,
(4)                 'DejaVu Serif': DejaVuSerifFontConstants,
(4)                 'cmb10': ComputerModernFontConstants,
(4)                 'cmex10': ComputerModernFontConstants,
(4)                 'cmmi10': ComputerModernFontConstants,
(4)                 'cmr10': ComputerModernFontConstants,
(4)                 'cmss10': ComputerModernFontConstants,
(4)                 'cmsy10': ComputerModernFontConstants,
(4)                 'cmtt10': ComputerModernFontConstants,
(4)                 'STIXGeneral': STIXFontConstants,
(4)                 'STIXNonUnicode': STIXFontConstants,
(4)                 'STIXSizeFiveSym': STIXFontConstants,
(4)                 'STIXSizeFourSym': STIXFontConstants,
(4)                 'STIXSizeThreeSym': STIXFontConstants,
(4)                 'STIXSizeTwoSym': STIXFontConstants,
(4)                 'STIXSizeOneSym': STIXFontConstants,
(4)                 'Bitstream Vera Sans': DejaVuSansFontConstants,
(4)                 'Bitstream Vera': DejaVuSansFontConstants,
(4)                 }
(0)             def _get_font_constant_set(state: ParserState) -> type[FontConstantsBase]:
(4)                 constants = _font_constant_mapping.get(
(8)                     state.fontset._get_font(state.font).family_name, FontConstantsBase)
(4)                 if (constants is STIXFontConstants and
(12)                        isinstance(state.fontset, StixSansFonts)):
(8)                     return STIXSansFontConstants
(4)                 return constants
(0)             class Node:
(4)                 """A node in the TeX box model."""
(4)                 def __init__(self) -> None:
(8)                     self.size = 0
(4)                 def __repr__(self) -> str:
(8)                     return type(self).__name__
(4)                 def get_kerning(self, next: Node | None) -> float:
(8)                     return 0.0
(4)                 def shrink(self) -> None:
(8)                     """
(8)                     Shrinks one level smaller.  There are only three levels of
(8)                     sizes, after which things will no longer get smaller.
(8)                     """
(8)                     self.size += 1
(4)                 def render(self, output: Output, x: float, y: float) -> None:
(8)                     """Render this node."""
(0)             class Box(Node):
(4)                 """A node with a physical location."""
(4)                 def __init__(self, width: float, height: float, depth: float) -> None:
(8)                     super().__init__()
(8)                     self.width  = width
(8)                     self.height = height
(8)                     self.depth  = depth
(4)                 def shrink(self) -> None:
```

```
803: (8)                    super().shrink()
804: (8)                    if self.size < NUM_SIZE_LEVELS:
805: (12)                       self.width  *= SHRINK_FACTOR
806: (12)                       self.height *= SHRINK_FACTOR
807: (12)                       self.depth  *= SHRINK_FACTOR
808: (4)                def render(self, output: Output,  # type: ignore[override]
809: (15)                           x1: float, y1: float, x2: float, y2: float) -> None:
810: (8)                    pass
811: (0)            class Vbox(Box):
812: (4)                """A box with only height (zero width)."""
813: (4)                def __init__(self, height: float, depth: float):
814: (8)                    super().__init__(0., height, depth)
815: (0)            class Hbox(Box):
816: (4)                """A box with only width (zero height and depth)."""
817: (4)                def __init__(self, width: float):
818: (8)                    super().__init__(width, 0., 0.)
819: (0)            class Char(Node):
820: (4)                """
821: (4)                A single character.
822: (4)                Unlike TeX, the font information and metrics are stored with each `Char`
823: (4)                to make it easier to lookup the font metrics when needed.  Note that TeX
824: (4)                boxes have a width, height, and depth, unlike Type1 and TrueType which use
825: (4)                a full bounding box and an advance in the x-direction.  The metrics must
826: (4)                be converted to the TeX model, and the advance (if different from width)
827: (4)                must be converted into a `Kern` node when the `Char` is added to its
parent
828: (4)                `Hlist`.
829: (4)                """
830: (4)                def __init__(self, c: str, state: ParserState):
831: (8)                    super().__init__()
832: (8)                    self.c = c
833: (8)                    self.fontset = state.fontset
834: (8)                    self.font = state.font
835: (8)                    self.font_class = state.font_class
836: (8)                    self.fontsize = state.fontsize
837: (8)                    self.dpi = state.dpi
838: (8)                    self._update_metrics()
839: (4)                def __repr__(self) -> str:
840: (8)                    return '`%s`' % self.c
841: (4)                def _update_metrics(self) -> None:
842: (8)                    metrics = self._metrics = self.fontset.get_metrics(
843: (12)                       self.font, self.font_class, self.c, self.fontsize, self.dpi)
844: (8)                    if self.c == ' ':
845: (12)                       self.width = metrics.advance
846: (8)                    else:
847: (12)                       self.width = metrics.width
848: (8)                    self.height = metrics.iceberg
849: (8)                    self.depth = -(metrics.iceberg - metrics.height)
850: (4)                def is_slanted(self) -> bool:
851: (8)                    return self._metrics.slanted
852: (4)                def get_kerning(self, next: Node | None) -> float:
853: (8)                    """
854: (8)                    Return the amount of kerning between this and the given character.
855: (8)                    This method is called when characters are strung together into `Hlist`
856: (8)                    to create `Kern` nodes.
857: (8)                    """
858: (8)                    advance = self._metrics.advance - self.width
859: (8)                    kern = 0.
860: (8)                    if isinstance(next, Char):
861: (12)                       kern = self.fontset.get_kern(
862: (16)                           self.font, self.font_class, self.c, self.fontsize,
863: (16)                           next.font, next.font_class, next.c, next.fontsize,
864: (16)                           self.dpi)
865: (8)                    return advance + kern
866: (4)                def render(self, output: Output, x: float, y: float) -> None:
867: (8)                    self.fontset.render_glyph(
868: (12)                       output, x, y,
869: (12)                       self.font, self.font_class, self.c, self.fontsize, self.dpi)
870: (4)                def shrink(self) -> None:
```

```
(8)                 super().shrink()
(8)                 if self.size < NUM_SIZE_LEVELS:
(12)                    self.fontsize *= SHRINK_FACTOR
(12)                    self.width    *= SHRINK_FACTOR
(12)                    self.height   *= SHRINK_FACTOR
(12)                    self.depth    *= SHRINK_FACTOR
(0)         class Accent(Char):
(4)             """
(4)             The font metrics need to be dealt with differently for accents,
(4)             since they are already offset correctly from the baseline in
(4)             TrueType fonts.
(4)             """
(4)             def _update_metrics(self) -> None:
(8)                 metrics = self._metrics = self.fontset.get_metrics(
(12)                    self.font, self.font_class, self.c, self.fontsize, self.dpi)
(8)                 self.width = metrics.xmax - metrics.xmin
(8)                 self.height = metrics.ymax - metrics.ymin
(8)                 self.depth = 0
(4)             def shrink(self) -> None:
(8)                 super().shrink()
(8)                 self._update_metrics()
(4)             def render(self, output: Output, x: float, y: float) -> None:
(8)                 self.fontset.render_glyph(
(12)                    output, x - self._metrics.xmin, y + self._metrics.ymin,
(12)                    self.font, self.font_class, self.c, self.fontsize, self.dpi)
(0)         class List(Box):
(4)             """A list of nodes (either horizontal or vertical)."""
(4)             def __init__(self, elements: T.Sequence[Node]):
(8)                 super().__init__(0., 0., 0.)
(8)                 self.shift_amount = 0.   # An arbitrary offset
(8)                 self.children = [*elements]  # The child nodes of this list
(8)                 self.glue_set     = 0.   # The glue setting of this list
(8)                 self.glue_sign    = 0    # 0: normal, -1: shrinking, 1: stretching
(8)                 self.glue_order   = 0    # The order of infinity (0 - 3) for the glue
(4)             def __repr__(self) -> str:
(8)                 return '{}<w={:.02f} h={:.02f} d={:.02f} s={:.02f}>[{}]'.format(
(12)                    super().__repr__(),
(12)                    self.width, self.height,
(12)                    self.depth, self.shift_amount,
(12)                    ', '.join([repr(x) for x in self.children]))
(4)             def _set_glue(self, x: float, sign: int, totals: list[float],
(18)                          error_type: str) -> None:
(8)                 self.glue_order = o = next(
(12)                    (i for i in range(len(totals))[::-1] if totals[i] != 0), 0)
(8)                 self.glue_sign = sign
(8)                 if totals[o] != 0.:
(12)                    self.glue_set = x / totals[o]
(8)                 else:
(12)                    self.glue_sign = 0
(12)                    self.glue_ratio = 0.
(8)                 if o == 0:
(12)                    if len(self.children):
(16)                        _log.warning("%s %s: %r",
(29)                                     error_type, type(self).__name__, self)
(4)             def shrink(self) -> None:
(8)                 for child in self.children:
(12)                    child.shrink()
(8)                 super().shrink()
(8)                 if self.size < NUM_SIZE_LEVELS:
(12)                    self.shift_amount *= SHRINK_FACTOR
(12)                    self.glue_set     *= SHRINK_FACTOR
(0)         class Hlist(List):
(4)             """A horizontal list of boxes."""
(4)             def __init__(self, elements: T.Sequence[Node], w: float = 0.0,
(17)                         m: T.Literal['additional', 'exactly'] = 'additional',
(17)                         do_kern: bool = True):
(8)                 super().__init__(elements)
(8)                 if do_kern:
(12)                    self.kern()
```

```
(8)                 self.hpack(w=w, m=m)
(4)             def kern(self) -> None:
(8)                 """
(8)                 Insert `Kern` nodes between `Char` nodes to set kerning.
(8)                 The `Char` nodes themselves determine the amount of kerning they need
(8)                 (in `~Char.get_kerning`), and this function just creates the correct
(8)                 linked list.
(8)                 """
(8)                 new_children = []
(8)                 num_children = len(self.children)
(8)                 if num_children:
(12)                    for i in range(num_children):
(16)                        elem = self.children[i]
(16)                        if i < num_children - 1:
(20)                            next = self.children[i + 1]
(16)                        else:
(20)                            next = None
(16)                        new_children.append(elem)
(16)                        kerning_distance = elem.get_kerning(next)
(16)                        if kerning_distance != 0.:
(20)                            kern = Kern(kerning_distance)
(20)                            new_children.append(kern)
(12)                    self.children = new_children
(4)             def hpack(self, w: float = 0.0,
(14)                      m: T.Literal['additional', 'exactly'] = 'additional') -> None:
(8)                 r"""
(8)                 Compute the dimensions of the resulting boxes, and adjust the glue if
(8)                 one of those dimensions is pre-specified.  The computed sizes normally
(8)                 enclose all of the material inside the new box; but some items may
(8)                 stick out if negative glue is used, if the box is overfull, or if a
(8)                 ``\vbox`` includes other boxes that have been shifted left.
(8)                 Parameters
(8)                 ----------
(8)                 w : float, default: 0
(12)                    A width.
(8)                 m : {'exactly', 'additional'}, default: 'additional'
(12)                    Whether to produce a box whose width is 'exactly' *w*; or a box
(12)                    with the natural width of the contents, plus *w* ('additional').
(8)                 Notes
(8)                 -----
(8)                 The defaults produce a box with the natural width of the contents.
(8)                 """
(8)                 h = 0.
(8)                 d = 0.
(8)                 x = 0.
(8)                 total_stretch = [0.] * 4
(8)                 total_shrink = [0.] * 4
(8)                 for p in self.children:
(12)                    if isinstance(p, Char):
(16)                        x += p.width
(16)                        h = max(h, p.height)
(16)                        d = max(d, p.depth)
(12)                    elif isinstance(p, Box):
(16)                        x += p.width
(16)                        if not np.isinf(p.height) and not np.isinf(p.depth):
(20)                            s = getattr(p, 'shift_amount', 0.)
(20)                            h = max(h, p.height - s)
(20)                            d = max(d, p.depth + s)
(12)                    elif isinstance(p, Glue):
(16)                        glue_spec = p.glue_spec
(16)                        x += glue_spec.width
(16)                        total_stretch[glue_spec.stretch_order] += glue_spec.stretch
(16)                        total_shrink[glue_spec.shrink_order] += glue_spec.shrink
(12)                    elif isinstance(p, Kern):
(16)                        x += p.width
(8)                 self.height = h
(8)                 self.depth = d
(8)                 if m == 'additional':
(12)                    w += x
```

```
(8)                 self.width = w
(8)                 x = w - x
(8)                 if x == 0.:
(12)                    self.glue_sign = 0
(12)                    self.glue_order = 0
(12)                    self.glue_ratio = 0.
(12)                    return
(8)                 if x > 0.:
(12)                    self._set_glue(x, 1, total_stretch, "Overful")
(8)                 else:
(12)                    self._set_glue(x, -1, total_shrink, "Underful")
(0)         class Vlist(List):
(4)             """A vertical list of boxes."""
(4)             def __init__(self, elements: T.Sequence[Node], h: float = 0.0,
(17)                         m: T.Literal['additional', 'exactly'] = 'additional'):
(8)                 super().__init__(elements)
(8)                 self.vpack(h=h, m=m)
(4)             def vpack(self, h: float = 0.0,
(14)                      m: T.Literal['additional', 'exactly'] = 'additional',
(14)                      l: float = np.inf) -> None:
(8)                 """
(8)                 Compute the dimensions of the resulting boxes, and to adjust the glue
(8)                 if one of those dimensions is pre-specified.
(8)                 Parameters
(8)                 ----------
(8)                 h : float, default: 0
(12)                    A height.
(8)                 m : {'exactly', 'additional'}, default: 'additional'
(12)                    Whether to produce a box whose height is 'exactly' *h*; or a box
(12)                    with the natural height of the contents, plus *h* ('additional').
(8)                 l : float, default: np.inf
(12)                    The maximum height.
(8)                 Notes
(8)                 -----
(8)                 The defaults produce a box with the natural height of the contents.
(8)                 """
(8)                 w = 0.
(8)                 d = 0.
(8)                 x = 0.
(8)                 total_stretch = [0.] * 4
(8)                 total_shrink = [0.] * 4
(8)                 for p in self.children:
(12)                    if isinstance(p, Box):
(16)                        x += d + p.height
(16)                        d = p.depth
(16)                        if not np.isinf(p.width):
(20)                            s = getattr(p, 'shift_amount', 0.)
(20)                            w = max(w, p.width + s)
(12)                    elif isinstance(p, Glue):
(16)                        x += d
(16)                        d = 0.
(16)                        glue_spec = p.glue_spec
(16)                        x += glue_spec.width
(16)                        total_stretch[glue_spec.stretch_order] += glue_spec.stretch
(16)                        total_shrink[glue_spec.shrink_order] += glue_spec.shrink
(12)                    elif isinstance(p, Kern):
(16)                        x += d + p.width
(16)                        d = 0.
(12)                    elif isinstance(p, Char):
(16)                        raise RuntimeError(
(20)                            "Internal mathtext error: Char node found in Vlist")
(8)                 self.width = w
(8)                 if d > l:
(12)                    x += d - l
(12)                    self.depth = l
(8)                 else:
(12)                    self.depth = d
(8)                 if m == 'additional':
(12)                    h += x
```

```
1078: (8)                   self.height = h
1079: (8)                   x = h - x
1080: (8)                   if x == 0:
1081: (12)                      self.glue_sign = 0
1082: (12)                      self.glue_order = 0
1083: (12)                      self.glue_ratio = 0.
1084: (12)                      return
1085: (8)                   if x > 0.:
1086: (12)                      self._set_glue(x, 1, total_stretch, "Overful")
1087: (8)                   else:
1088: (12)                      self._set_glue(x, -1, total_shrink, "Underful")
1089: (0)           class Rule(Box):
1090: (4)               """
1091: (4)               A solid black rectangle.
1092: (4)               It has *width*, *depth*, and *height* fields just as in an `Hlist`.
1093: (4)               However, if any of these dimensions is inf, the actual value will be
1094: (4)               determined by running the rule up to the boundary of the innermost
1095: (4)               enclosing box.  This is called a "running dimension".  The width is never
1096: (4)               running in an `Hlist`; the height and depth are never running in a
`Vlist`.
1097: (4)               """
1098: (4)               def __init__(self, width: float, height: float, depth: float, state:
ParserState):
1099: (8)                   super().__init__(width, height, depth)
1100: (8)                   self.fontset = state.fontset
1101: (4)               def render(self, output: Output,  # type: ignore[override]
1102: (15)                         x: float, y: float, w: float, h: float) -> None:
1103: (8)                   self.fontset.render_rect_filled(output, x, y, x + w, y + h)
1104: (0)           class Hrule(Rule):
1105: (4)               """Convenience class to create a horizontal rule."""
1106: (4)               def __init__(self, state: ParserState, thickness: float | None = None):
1107: (8)                   if thickness is None:
1108: (12)                      thickness = state.get_current_underline_thickness()
1109: (8)                   height = depth = thickness * 0.5
1110: (8)                   super().__init__(np.inf, height, depth, state)
1111: (0)           class Vrule(Rule):
1112: (4)               """Convenience class to create a vertical rule."""
1113: (4)               def __init__(self, state: ParserState):
1114: (8)                   thickness = state.get_current_underline_thickness()
1115: (8)                   super().__init__(thickness, np.inf, np.inf, state)
1116: (0)           class _GlueSpec(NamedTuple):
1117: (4)               width: float
1118: (4)               stretch: float
1119: (4)               stretch_order: int
1120: (4)               shrink: float
1121: (4)               shrink_order: int
1122: (0)           _GlueSpec._named = {  # type: ignore[attr-defined]
1123: (4)               'fil':         _GlueSpec(0., 1., 1, 0., 0),
1124: (4)               'fill':        _GlueSpec(0., 1., 2, 0., 0),
1125: (4)               'filll':       _GlueSpec(0., 1., 3, 0., 0),
1126: (4)               'neg_fil':     _GlueSpec(0., 0., 0, 1., 1),
1127: (4)               'neg_fill':    _GlueSpec(0., 0., 0, 1., 2),
1128: (4)               'neg_filll':   _GlueSpec(0., 0., 0, 1., 3),
1129: (4)               'empty':       _GlueSpec(0., 0., 0, 0., 0),
1130: (4)               'ss':          _GlueSpec(0., 1., 1, -1., 1),
1131: (0)           }
1132: (0)           class Glue(Node):
1133: (4)               """
1134: (4)               Most of the information in this object is stored in the underlying
1135: (4)               ``_GlueSpec`` class, which is shared between multiple glue objects.
1136: (4)               (This is a memory optimization which probably doesn't matter anymore, but
1137: (4)               it's easier to stick to what TeX does.)
1138: (4)               """
1139: (4)               def __init__(self,
1140: (17)                           glue_type: _GlueSpec | T.Literal["fil", "fill", "filll",
1141: (50)                                                            "neg_fil", "neg_fill",
"neg_filll",
1142: (50)                                                            "empty", "ss"]):
1143: (8)                   super().__init__()
```

```
(8)                 if isinstance(glue_type, str):
(12)                    glue_spec = _GlueSpec._named[glue_type]  # type: ignore[attr-defined]
(8)                 elif isinstance(glue_type, _GlueSpec):
(12)                    glue_spec = glue_type
(8)                 else:
(12)                    raise ValueError("glue_type must be a glue spec name or instance")
(8)                 self.glue_spec = glue_spec
(4)             def shrink(self) -> None:
(8)                 super().shrink()
(8)                 if self.size < NUM_SIZE_LEVELS:
(12)                    g = self.glue_spec
(12)                    self.glue_spec = g._replace(width=g.width * SHRINK_FACTOR)
(0)         class HCentered(Hlist):
(4)             """
(4)             A convenience class to create an `Hlist` whose contents are
(4)             centered within its enclosing box.
(4)             """
(4)             def __init__(self, elements: list[Node]):
(8)                 super().__init__([Glue('ss'), *elements, Glue('ss')], do_kern=False)
(0)         class VCentered(Vlist):
(4)             """
(4)             A convenience class to create a `Vlist` whose contents are
(4)             centered within its enclosing box.
(4)             """
(4)             def __init__(self, elements: list[Node]):
(8)                 super().__init__([Glue('ss'), *elements, Glue('ss')])
(0)         class Kern(Node):
(4)             """
(4)             A `Kern` node has a width field to specify a (normally
(4)             negative) amount of spacing. This spacing correction appears in
(4)             horizontal lists between letters like A and V when the font
(4)             designer said that it looks better to move them closer together or
(4)             further apart. A kern node can also appear in a vertical list,
(4)             when its *width* denotes additional spacing in the vertical
(4)             direction.
(4)             """
(4)             height = 0
(4)             depth = 0
(4)             def __init__(self, width: float):
(8)                 super().__init__()
(8)                 self.width = width
(4)             def __repr__(self) -> str:
(8)                 return "k%.02f" % self.width
(4)             def shrink(self) -> None:
(8)                 super().shrink()
(8)                 if self.size < NUM_SIZE_LEVELS:
(12)                    self.width *= SHRINK_FACTOR
(0)         class AutoHeightChar(Hlist):
(4)             """
(4)             A character as close to the given height and depth as possible.
(4)             When using a font with multiple height versions of some characters (such as
(4)             the BaKoMa fonts), the correct glyph will be selected, otherwise this will
(4)             always just return a scaled version of the glyph.
(4)             """
(4)             def __init__(self, c: str, height: float, depth: float, state: ParserState,
(17)                         always: bool = False, factor: float | None = None):
(8)                 alternatives = state.fontset.get_sized_alternatives_for_symbol(
(12)                    state.font, c)
(8)                 xHeight = state.fontset.get_xheight(
(12)                    state.font, state.fontsize, state.dpi)
(8)                 state = state.copy()
(8)                 target_total = height + depth
(8)                 for fontname, sym in alternatives:
(12)                    state.font = fontname
(12)                    char = Char(sym, state)
(12)                    if char.height + char.depth >= target_total - 0.2 * xHeight:
```

```
(16)                        break
(8)                 shift = 0.0
(8)                 if state.font != 0 or len(alternatives) == 1:
(12)                    if factor is None:
(16)                        factor = target_total / (char.height + char.depth)
(12)                    state.fontsize *= factor
(12)                    char = Char(sym, state)
(12)                    shift = (depth - char.depth)
(8)                 super().__init__([char])
(8)                 self.shift_amount = shift
(0)         class AutoWidthChar(Hlist):
(4)             """
(4)             A character as close to the given width as possible.
(4)             When using a font with multiple width versions of some characters (such as
(4)             the BaKoMa fonts), the correct glyph will be selected, otherwise this will
(4)             always just return a scaled version of the glyph.
(4)             """
(4)             def __init__(self, c: str, width: float, state: ParserState, always: bool
= False,
(17)                         char_class: type[Char] = Char):
(8)                 alternatives = state.fontset.get_sized_alternatives_for_symbol(
(12)                    state.font, c)
(8)                 state = state.copy()
(8)                 for fontname, sym in alternatives:
(12)                    state.font = fontname
(12)                    char = char_class(sym, state)
(12)                    if char.width >= width:
(16)                        break
(8)                 factor = width / char.width
(8)                 state.fontsize *= factor
(8)                 char = char_class(sym, state)
(8)                 super().__init__([char])
(8)                 self.width = char.width
(0)         def ship(box: Box, xy: tuple[float, float] = (0, 0)) -> Output:
(4)             """
(4)             Ship out *box* at offset *xy*, converting it to an `Output`.
(4)             Since boxes can be inside of boxes inside of boxes, the main work of
`ship`
(4)             is done by two mutually recursive routines, `hlist_out` and `vlist_out`,
(4)             which traverse the `Hlist` nodes and `Vlist` nodes inside of horizontal
(4)             and vertical boxes.  The global variables used in TeX to store state as it
(4)             processes have become local variables here.
(4)             """
(4)             ox, oy = xy
(4)             cur_v = 0.
(4)             cur_h = 0.
(4)             off_h = ox
(4)             off_v = oy + box.height
(4)             output = Output(box)
(4)             def clamp(value: float) -> float:
(8)                 return -1e9 if value < -1e9 else +1e9 if value > +1e9 else value
(4)             def hlist_out(box: Hlist) -> None:
(8)                 nonlocal cur_v, cur_h, off_h, off_v
(8)                 cur_g = 0
(8)                 cur_glue = 0.
(8)                 glue_order = box.glue_order
(8)                 glue_sign = box.glue_sign
(8)                 base_line = cur_v
(8)                 left_edge = cur_h
(8)                 for p in box.children:
(12)                    if isinstance(p, Char):
(16)                        p.render(output, cur_h + off_h, cur_v + off_v)
(16)                        cur_h += p.width
(12)                    elif isinstance(p, Kern):
(16)                        cur_h += p.width
(12)                    elif isinstance(p, List):
(16)                        if len(p.children) == 0:
(20)                            cur_h += p.width
(16)                        else:
```

```
(20)                                edge = cur_h
(20)                                cur_v = base_line + p.shift_amount
(20)                                if isinstance(p, Hlist):
(24)                                    hlist_out(p)
(20)                                elif isinstance(p, Vlist):
(24)                                    vlist_out(p)
(20)                                else:
(24)                                    assert False, "unreachable code"
(20)                                cur_h = edge + p.width
(20)                                cur_v = base_line
(12)                        elif isinstance(p, Box):
(16)                            rule_height = p.height
(16)                            rule_depth = p.depth
(16)                            rule_width = p.width
(16)                            if np.isinf(rule_height):
(20)                                rule_height = box.height
(16)                            if np.isinf(rule_depth):
(20)                                rule_depth = box.depth
(16)                            if rule_height > 0 and rule_width > 0:
(20)                                cur_v = base_line + rule_depth
(20)                                p.render(output,
(29)                                         cur_h + off_h, cur_v + off_v,
(29)                                         rule_width, rule_height)
(20)                                cur_v = base_line
(16)                            cur_h += rule_width
(12)                        elif isinstance(p, Glue):
(16)                            glue_spec = p.glue_spec
(16)                            rule_width = glue_spec.width - cur_g
(16)                            if glue_sign != 0:  # normal
(20)                                if glue_sign == 1:  # stretching
(24)                                    if glue_spec.stretch_order == glue_order:
(28)                                        cur_glue += glue_spec.stretch
(28)                                        cur_g = round(clamp(box.glue_set * cur_glue))
(20)                                elif glue_spec.shrink_order == glue_order:
(24)                                    cur_glue += glue_spec.shrink
(24)                                    cur_g = round(clamp(box.glue_set * cur_glue))
(16)                            rule_width += cur_g
(16)                            cur_h += rule_width
(4)                 def vlist_out(box: Vlist) -> None:
(8)                     nonlocal cur_v, cur_h, off_h, off_v
(8)                     cur_g = 0
(8)                     cur_glue = 0.
(8)                     glue_order = box.glue_order
(8)                     glue_sign = box.glue_sign
(8)                     left_edge = cur_h
(8)                     cur_v -= box.height
(8)                     top_edge = cur_v
(8)                     for p in box.children:
(12)                        if isinstance(p, Kern):
(16)                            cur_v += p.width
(12)                        elif isinstance(p, List):
(16)                            if len(p.children) == 0:
(20)                                cur_v += p.height + p.depth
(16)                            else:
(20)                                cur_v += p.height
(20)                                cur_h = left_edge + p.shift_amount
(20)                                save_v = cur_v
(20)                                p.width = box.width
(20)                                if isinstance(p, Hlist):
(24)                                    hlist_out(p)
(20)                                elif isinstance(p, Vlist):
(24)                                    vlist_out(p)
(20)                                else:
(24)                                    assert False, "unreachable code"
(20)                                cur_v = save_v + p.depth
(20)                                cur_h = left_edge
(12)                        elif isinstance(p, Box):
(16)                            rule_height = p.height
(16)                            rule_depth = p.depth
```

```
1346: (16)                          rule_width = p.width
1347: (16)                          if np.isinf(rule_width):
1348: (20)                              rule_width = box.width
1349: (16)                          rule_height += rule_depth
1350: (16)                          if rule_height > 0 and rule_depth > 0:
1351: (20)                              cur_v += rule_height
1352: (20)                              p.render(output,
1353: (29)                                       cur_h + off_h, cur_v + off_v,
1354: (29)                                       rule_width, rule_height)
1355: (12)                      elif isinstance(p, Glue):
1356: (16)                          glue_spec = p.glue_spec
1357: (16)                          rule_height = glue_spec.width - cur_g
1358: (16)                          if glue_sign != 0:  # normal
1359: (20)                              if glue_sign == 1:  # stretching
1360: (24)                                  if glue_spec.stretch_order == glue_order:
1361: (28)                                      cur_glue += glue_spec.stretch
1362: (28)                                      cur_g = round(clamp(box.glue_set * cur_glue))
1363: (20)                              elif glue_spec.shrink_order == glue_order:  # shrinking
1364: (24)                                  cur_glue += glue_spec.shrink
1365: (24)                                  cur_g = round(clamp(box.glue_set * cur_glue))
1366: (16)                          rule_height += cur_g
1367: (16)                          cur_v += rule_height
1368: (12)                      elif isinstance(p, Char):
1369: (16)                          raise RuntimeError(
1370: (20)                              "Internal mathtext error: Char node found in vlist")
1371: (4)               assert isinstance(box, Hlist)
1372: (4)               hlist_out(box)
1373: (4)               return output
1374: (0)           def Error(msg: str) -> ParserElement:
1375: (4)               """Helper class to raise parser errors."""
1376: (4)               def raise_error(s: str, loc: int, toks: ParseResults) -> T.Any:
1377: (8)                   raise ParseFatalException(s, loc, msg)
1378: (4)               return Empty().setParseAction(raise_error)
1379: (0)           class ParserState:
1380: (4)               """
1381: (4)               Parser state.
1382: (4)               States are pushed and popped from a stack as necessary, and the "current"
1383: (4)               state is always at the top of the stack.
1384: (4)               Upon entering and leaving a group { } or math/non-math, the stack is
pushed
1385: (4)               and popped accordingly.
1386: (4)               """
1387: (4)               def __init__(self, fontset: Fonts, font: str, font_class: str, fontsize:
float,
1388: (17)                           dpi: float):
1389: (8)                   self.fontset = fontset
1390: (8)                   self._font = font
1391: (8)                   self.font_class = font_class
1392: (8)                   self.fontsize = fontsize
1393: (8)                   self.dpi = dpi
1394: (4)               def copy(self) -> ParserState:
1395: (8)                   return copy.copy(self)
1396: (4)               @property
1397: (4)               def font(self) -> str:
1398: (8)                   return self._font
1399: (4)               @font.setter
1400: (4)               def font(self, name: str) -> None:
1401: (8)                   if name in ('rm', 'it', 'bf', 'bfit'):
1402: (12)                      self.font_class = name
1403: (8)                   self._font = name
1404: (4)               def get_current_underline_thickness(self) -> float:
1405: (8)                   """Return the underline thickness for this state."""
1406: (8)                   return self.fontset.get_underline_thickness(
1407: (12)                      self.font, self.fontsize, self.dpi)
1408: (0)           def cmd(expr: str, args: ParserElement) -> ParserElement:
1409: (4)               r"""
1410: (4)               Helper to define TeX commands.
1411: (4)               ``cmd("\cmd", args)`` is equivalent to
1412: (4)               ``"\cmd" - (args | Error("Expected \cmd{arg}{...}"))`` where the names in
```

```
(4)             the error message are taken from element names in *args*.  If *expr*
(4)             already includes arguments (e.g. "\cmd{arg}{...}"), then they are stripped
(4)             when constructing the parse element, but kept (and *expr* is used as is)
in
(4)             the error message.
(4)             """
(4)             def names(elt: ParserElement) -> T.Generator[str, None, None]:
(8)                 if isinstance(elt, ParseExpression):
(12)                    for expr in elt.exprs:
(16)                        yield from names(expr)
(8)                 elif elt.resultsName:
(12)                    yield elt.resultsName
(4)             csname = expr.split("{", 1)[0]
(4)             err = (csname + "".join("{%s}" % name for name in names(args))
(11)                   if expr == csname else expr)
(4)             return csname - (args | Error(f"Expected {err}"))
(0)         class Parser:
(4)             """
(4)             A pyparsing-based parser for strings containing math expressions.
(4)             Raw text may also appear outside of pairs of ``$``.
(4)             The grammar is based directly on that in TeX, though it cuts a few
corners.
(4)             """
(4)             class _MathStyle(enum.Enum):
(8)                 DISPLAYSTYLE = 0
(8)                 TEXTSTYLE = 1
(8)                 SCRIPTSTYLE = 2
(8)                 SCRIPTSCRIPTSTYLE = 3
(4)             _binary_operators = set(
(6)               '+ * - \N{MINUS SIGN}'
(6)               r'''
(6)               \pm             \sqcap                   \rhd
(6)               \mp             \sqcup                   \unlhd
(6)               \times          \vee                     \unrhd
(6)               \div            \wedge                   \oplus
(6)               \ast            \setminus                \ominus
(6)               \star           \wr                      \otimes
(6)               \circ           \diamond                 \oslash
(6)               \bullet         \bigtriangleup           \odot
(6)               \cdot           \bigtriangledown         \bigcirc
(6)               \cap            \triangleleft            \dagger
(6)               \cup            \triangleright           \ddagger
(6)               \uplus          \lhd                     \amalg
(6)               \dotplus        \dotminus                \Cap
(6)               \Cup            \barwedge                \boxdot
(6)               \boxminus       \boxplus                 \boxtimes
(6)               \curlyvee       \curlywedge              \divideontimes
(6)               \doublebarwedge \leftthreetimes          \rightthreetimes
(6)               \slash          \veebar                  \barvee
(6)               \cupdot         \intercal                \amalg
(6)               \circledcirc    \circleddash             \circledast
(6)               \boxbar         \obar                    \merge
(6)               \minuscolon     \dotsminusdots
(6)               '''.split())
(4)             _relation_symbols = set(r'''
(6)               = < > :
(6)               \leq          \geq          \equiv       \models
(6)               \prec         \succ         \sim         \perp
(6)               \preceq       \succeq       \simeq       \mid
(6)               \ll           \gg           \asymp       \parallel
(6)               \subset       \supset       \approx      \bowtie
(6)               \subseteq     \supseteq     \cong        \Join
(6)               \sqsubset     \sqsupset     \neq         \smile
(6)               \sqsubseteq   \sqsupseteq   \doteq       \frown
(6)               \in           \ni           \propto      \vdash
(6)               \dashv        \dots         \doteqdot    \leqslant
(6)               \geqslant     \lneqq        \gneqq       \lessgtr
(6)               \leqslant     \geqslant     \eqgtr       \eqless
(6)               \eqslantless  \eqslantgtr   \lesseqgtr   \backsim
```

```
(6)                 \backsimeq    \lesssim      \gtrsim      \precsim
(6)                 \precnsim     \gnsim        \lnsim       \succsim
(6)                 \succnsim     \nsim         \lesseqgtr   \gtreqless
(6)                 \gtreqless    \subseteqq    \supseteqq   \subsetneqq
(6)                 \supsetneqq   \lesssapprox   \approxeq    \gtrapprox
(6)                 \precapprox   \succapprox   \precnapprox \succnapprox
(6)                 \npreccurlyeq \nsucccurlyeq \nsqsubseteq \nsqsupseteq
(6)                 \sqsubsetneq  \sqsupsetneq  \nlesssim    \ngtrsim
(6)                 \nlessgtr     \ngtrless     \lnapprox    \gnapprox
(6)                 \napprox      \approxeq     \approxident \lll
(6)                 \ggg          \nparallel    \Vdash       \Vvdash
(6)                 \nVdash       \nvdash       \vDash       \nvDash
(6)                 \nVDash       \oequal       \simneqq     \triangle
(6)                 \triangleq         \triangleeq         \triangleleft
(6)                 \triangleright     \ntriangleleft      \ntriangleright
(6)                 \trianglelefteq    \ntrianglelefteq    \trianglerighteq
(6)                 \ntrianglerighteq  \blacktriangleleft  \blacktriangleright
(6)                 \equalparallel     \measuredrightangle \varlrtriangle
(6)                 \Doteq        \Bumpeq       \Subset      \Supset
(6)                 \backepsilon  \because      \therefore   \bot
(6)                 \top          \bumpeq       \circeq      \coloneq
(6)                 \curlyeqprec  \curlyeqsucc  \eqcirc      \eqcolon
(6)                 \eqsim        \fallingdotseq \gtrdot     \gtrless
(6)                 \ltimes       \rtimes       \lessdot     \ne
(6)                 \ncong        \nequiv       \ngeq        \ngtr
(6)                 \nleq         \nless        \nmid        \notin
(6)                 \nprec        \nsubset      \nsubseteq   \nsucc
(6)                 \nsupset      \nsupseteq    \pitchfork   \preccurlyeq
(6)                 \risingdotseq \subsetneq    \succcurlyeq \supsetneq
(6)                 \varpropto    \vartriangleleft \scurel
(6)                 \vartriangleright \rightangle \equal     \backcong
(6)                 \eqdef        \wedgeq       \questeq     \between
(6)                 \veeeq        \disin        \varisins    \isins
(6)                 \isindot      \varisinobar  \isinobar    \isinvb
(6)                 \isinE        \nisd         \varnis      \nis
(6)                 \varniobar    \niobar       \bagmember   \ratio
(6)                 \Equiv        \stareq       \measeq      \arceq
(6)                 \rightassert  \rightModels  \smallin     \smallowns
(6)                 \notsmallowns \nsimeq'''.split())
(4)               _arrow_symbols = set(r"""
(5)                \leftarrow \longleftarrow \uparrow \Leftarrow \Longleftarrow
(5)                \Uparrow \rightarrow \longrightarrow \downarrow \Rightarrow
(5)                \Longrightarrow \Downarrow \leftrightarrow \updownarrow
(5)                \longleftrightarrow \updownarrow \Leftrightarrow
(5)                \Longleftrightarrow \Updownarrow \mapsto \longmapsto \nearrow
(5)                \hookleftarrow \hookrightarrow \searrow \leftharpoonup
(5)                \rightharpoonup \swarrow \leftharpoondown \rightharpoondown
(5)                \nwarrow \rightleftharpoons \leadsto \dashrightarrow
(5)                \dashleftarrow \leftleftarrows \leftrightarrows \Lleftarrow
(5)                \Rrightarrow \twoheadleftarrow \leftarrowtail \looparrowleft
(5)                \leftrightharpoons \curvearrowleft \circlearrowleft \Lsh
(5)                \upuparrows \upharpoonleft \downharpoonleft \multimap
(5)                \leftrightsquigarrow \rightrightarrows \rightleftarrows
(5)                \rightrightarrows \rightleftarrows \twoheadrightarrow
(5)                \rightarrowtail \looparrowright \rightleftharpoons
(5)                \curvearrowright \circlearrowright \Rsh \downdownarrows
(5)                \upharpoonright \downharpoonright \rightsquigarrow \nleftarrow
(5)                \nrightarrow \nLeftarrow \nRightarrow \nleftrightarrow
(5)                \nLeftrightarrow \to \Swarrow \Searrow \Nwarrow \Nearrow
(5)                \leftsquigarrow \overleftarrow \overleftrightarrow \cwopencirclearrow
(5)                \downzigzagarrow \cupleftarrow \rightzigzagarrow \twoheaddownarrow
(5)                \updownarrowbar \twoheaduparrow \rightarrowbar \updownarrows
(5)                \barleftarrow \mapsfrom \mapsdown \mapsup \Ldsh \Rdsh
(5)                """.split())
(4)               _spaced_symbols = _binary_operators | _relation_symbols | _arrow_symbols
(4)               _punctuation_symbols = set(r', ; . ! \ldotp \cdotp'.split())
(4)               _overunder_symbols = set(r'''
(7)                  \sum \prod \coprod \bigcap \bigcup \bigsqcup \bigvee
(7)                  \bigwedge \bigodot \bigotimes \bigoplus \biguplus
```

```
(7)                 '''.split())
(4)             _overunder_functions = set("lim liminf limsup sup max min".split())
(4)             _dropsub_symbols = set(r'\int \oint \iint \oiint \iiint \oiiint
\iiint'.split())
(4)             _fontnames = set("rm cal it tt sf bf bfit "
(21)                             "default bb frak scr regular".split())
(4)             _function_names = set("""
(6)               arccos csc ker min arcsin deg lg Pr arctan det lim sec arg dim
(6)               liminf sin cos exp limsup sinh cosh gcd ln sup cot hom log tan
(6)               coth inf max tanh""".split())
(4)             _ambi_delims = set(r"""
(6)               | \| / \backslash \uparrow \downarrow \updownarrow \Uparrow
(6)               \Downarrow \Updownarrow . \vert \Vert""".split())
(4)             _left_delims = set(r"""
(6)               ( [ \{ < \lfloor \langle \lceil \lbrace \leftbrace \lbrack \leftparen
\lgroup
(6)               """.split())
(4)             _right_delims = set(r"""
(6)               ) ] \} > \rfloor \rangle \rceil \rbrace \rightbrace \rbrack \rightparen
\rgroup
(6)               """.split())
(4)             _delims = _left_delims | _right_delims | _ambi_delims
(4)             _small_greek = set([unicodedata.name(chr(i)).split()[-1].lower() for i in
(23)                                range(ord('\N{GREEK SMALL LETTER ALPHA}'),
(29)                                      ord('\N{GREEK SMALL LETTER OMEGA}') + 1)])
(4)             _latin_alphabets = set(string.ascii_letters)
(4)             def __init__(self) -> None:
(8)                 p = types.SimpleNamespace()
(8)                 def set_names_and_parse_actions() -> None:
(12)                    for key, val in vars(p).items():
(16)                        if not key.startswith('_'):
(20)                            if key not in ("token", "placeable", "auto_delim"):
(24)                                val.setName(key)
(20)                            if hasattr(self, key):
(24)                                val.setParseAction(getattr(self, key))
(8)                 def csnames(group: str, names: Iterable[str]) -> Regex:
(12)                    ends_with_alpha = []
(12)                    ends_with_nonalpha = []
(12)                    for name in names:
(16)                        if name[-1].isalpha():
(20)                            ends_with_alpha.append(name)
(16)                        else:
(20)                            ends_with_nonalpha.append(name)
(12)                    return Regex(
(16)                        r"\\(?P<{group}>(?:{alpha})(?![A-Za-z]){additional}
{nonalpha})".format(
(20)                            group=group,
(20)                            alpha="|".join(map(re.escape, ends_with_alpha)),
(20)                            additional="|" if ends_with_nonalpha else "",
(20)                            nonalpha="|".join(map(re.escape, ends_with_nonalpha)),
(16)                        )
(12)                    )
(8)                 p.float_literal  = Regex(r"[-+]?([0-9]+\.?[0-9]*|\.[0-9]+)")
(8)                 p.space          = oneOf(self._space_widths)("space")
(8)                 p.style_literal  = oneOf(
(12)                    [str(e.value) for e in self._MathStyle])("style_literal")
(8)                 p.symbol         = Regex(
(12)                    r"[a-zA-Z0-9 +\-*/<>=:,.;!\?&'@()\[\]|\U00000080-\U0001ffff]"
(12)                    r"|\\[%${}\[\]_|]"
(12)                    + r"|\\(?:{})(?![A-Za-z])".format(
(16)                        "|".join(map(re.escape, tex2uni)))
(8)                 )("sym").leaveWhitespace()
(8)                 p.unknown_symbol = Regex(r"\\[A-Za-z]+")("name")
(8)                 p.font           = csnames("font", self._fontnames)
(8)                 p.start_group    = Optional(r"\math" + oneOf(self._fontnames)("font"))
+ "{"
(8)                 p.end_group      = Literal("}")
(8)                 p.delim          = oneOf(self._delims)
(8)                 p.auto_delim       = Forward()
```

```
(8)                     p.placeable        = Forward()
(8)                     p.named_placeable  = Forward()
(8)                     p.required_group   = Forward()
(8)                     p.optional_group   = Forward()
(8)                     p.token            = Forward()
(8)                     p.named_placeable <<= p.placeable
(8)                     set_names_and_parse_actions()  # for mutually recursive definitions.
(8)                     p.optional_group <<= "{" + ZeroOrMore(p.token)("group") + "}"
(8)                     p.required_group <<= "{" + OneOrMore(p.token)("group") + "}"
(8)                     p.customspace = cmd(r"\hspace", "{" + p.float_literal("space") + "}")
(8)                     p.accent = (
(12)                        csnames("accent", [*self._accent_map, *self._wide_accents])
(12)                        - p.named_placeable("sym"))
(8)                     p.function = csnames("name", self._function_names)
(8)                     p.group = p.start_group + ZeroOrMore(p.token)("group") + p.end_group
(8)                     p.unclosed_group = (p.start_group + ZeroOrMore(p.token)("group") +
StringEnd())
(8)                     p.frac  = cmd(r"\frac", p.required_group("num") +
p.required_group("den"))
(8)                     p.dfrac = cmd(r"\dfrac", p.required_group("num") +
p.required_group("den"))
(8)                     p.binom = cmd(r"\binom", p.required_group("num") +
p.required_group("den"))
(8)                     p.genfrac = cmd(
(12)                        r"\genfrac",
(12)                        "{" + Optional(p.delim)("ldelim") + "}"
(12)                        + "{" + Optional(p.delim)("rdelim") + "}"
(12)                        + "{" + p.float_literal("rulesize") + "}"
(12)                        + "{" + Optional(p.style_literal)("style") + "}"
(12)                        + p.required_group("num")
(12)                        + p.required_group("den"))
(8)                     p.sqrt = cmd(
(12)                        r"\sqrt{value}",
(12)                        Optional("[" + OneOrMore(NotAny("]") + p.token)("root") + "]")
(12)                        + p.required_group("value"))
(8)                     p.overline = cmd(r"\overline", p.required_group("body"))
(8)                     p.overset  = cmd(
(12)                        r"\overset",
(12)                        p.optional_group("annotation") + p.optional_group("body"))
(8)                     p.underset = cmd(
(12)                        r"\underset",
(12)                        p.optional_group("annotation") + p.optional_group("body"))
(8)                     p.text = cmd(r"\text", QuotedString('{', '\\', endQuoteChar="}"))
(8)                     p.substack = cmd(r"\substack",
(27)                                     nested_expr(opener="{", closer="}",
(39)                                                 content=Group(OneOrMore(p.token)) +
(39)                                                 ZeroOrMore(Literal("\\\\").suppress()))
("parts"))
(8)                     p.subsuper = (
(12)                        (Optional(p.placeable)("nucleus")
(13)                         + OneOrMore(oneOf(["_", "^"]) - p.placeable)("subsuper")
(13)                         + Regex("'*")("apostrophes"))
(12)                        | Regex("'+")("apostrophes")
(12)                        | (p.named_placeable("nucleus") + Regex("'*")("apostrophes"))
(8)                     )
(8)                     p.simple = p.space | p.customspace | p.font | p.subsuper
(8)                     p.token <<= (
(12)                        p.simple
(12)                        | p.auto_delim
(12)                        | p.unclosed_group
(12)                        | p.unknown_symbol  # Must be last
(8)                     )
(8)                     p.operatorname = cmd(r"\operatorname", "{" + ZeroOrMore(p.simple)
("name") + "}")
(8)                     p.boldsymbol = cmd(
(12)                        r"\boldsymbol", "{" + ZeroOrMore(p.simple)("value") + "}")
(8)                     p.placeable     <<= (
(12)                        p.accent     # Must be before symbol as all accents are symbols
(12)                        | p.symbol   # Must be second to catch all named symbols and
```

```
single
1676: (12)                      | p.function
1677: (12)                      | p.operatorname
1678: (12)                      | p.group
1679: (12)                      | p.frac
1680: (12)                      | p.dfrac
1681: (12)                      | p.binom
1682: (12)                      | p.genfrac
1683: (12)                      | p.overset
1684: (12)                      | p.underset
1685: (12)                      | p.sqrt
1686: (12)                      | p.overline
1687: (12)                      | p.text
1688: (12)                      | p.boldsymbol
1689: (12)                      | p.substack
1690: (8)                   )
1691: (8)                   mdelim = r"\middle" - (p.delim("mdelim") | Error("Expected a
delimiter"))
1692: (8)                   p.auto_delim    <<= (
1693: (12)                      r"\left" - (p.delim("left") | Error("Expected a delimiter"))
1694: (12)                      + ZeroOrMore(p.simple | p.auto_delim | mdelim)("mid")
1695: (12)                      + r"\right" - (p.delim("right") | Error("Expected a delimiter"))
1696: (8)                   )
1697: (8)                   p.math          = OneOrMore(p.token)
1698: (8)                   p.math_string   = QuotedString('$', '\\', unquoteResults=False)
1699: (8)                   p.non_math      = Regex(r"(?:(?:\\[$])|[^$])*").leaveWhitespace()
1700: (8)                   p.main          = (
1701: (12)                      p.non_math + ZeroOrMore(p.math_string + p.non_math) + StringEnd()
1702: (8)                   )
1703: (8)                   set_names_and_parse_actions()  # for leaf definitions.
1704: (8)                   self._expression = p.main
1705: (8)                   self._math_expression = p.math
1706: (8)                   self._in_subscript_or_superscript = False
1707: (4)               def parse(self, s: str, fonts_object: Fonts, fontsize: float, dpi: float)
-> Hlist:
1708: (8)                   """
1709: (8)                   Parse expression *s* using the given *fonts_object* for
1710: (8)                   output, at the given *fontsize* and *dpi*.
1711: (8)                   Returns the parse tree of `Node` instances.
1712: (8)                   """
1713: (8)                   self._state_stack = [
1714: (12)                      ParserState(fonts_object, 'default', 'rm', fontsize, dpi)]
1715: (8)                   self._em_width_cache: dict[tuple[str, float, float], float] = {}
1716: (8)                   try:
1717: (12)                      result = self._expression.parseString(s)
1718: (8)                   except ParseBaseException as err:
1719: (12)                      raise ValueError("\n" + ParseException.explain(err, 0)) from None
1720: (8)                   self._state_stack = []
1721: (8)                   self._in_subscript_or_superscript = False
1722: (8)                   self._em_width_cache = {}
1723: (8)                   ParserElement.resetCache()
1724: (8)                   return T.cast(Hlist, result[0])  # Known return type from main.
1725: (4)               def get_state(self) -> ParserState:
1726: (8)                   """Get the current `State` of the parser."""
1727: (8)                   return self._state_stack[-1]
1728: (4)               def pop_state(self) -> None:
1729: (8)                   """Pop a `State` off of the stack."""
1730: (8)                   self._state_stack.pop()
1731: (4)               def push_state(self) -> None:
1732: (8)                   """Push a new `State` onto the stack, copying the current state."""
1733: (8)                   self._state_stack.append(self.get_state().copy())
1734: (4)               def main(self, toks: ParseResults) -> list[Hlist]:
1735: (8)                   return [Hlist(toks.asList())]
1736: (4)               def math_string(self, toks: ParseResults) -> ParseResults:
1737: (8)                   return self._math_expression.parseString(toks[0][1:-1], parseAll=True)
1738: (4)               def math(self, toks: ParseResults) -> T.Any:
1739: (8)                   hlist = Hlist(toks.asList())
1740: (8)                   self.pop_state()
1741: (8)                   return [hlist]
```

```
(4)     def non_math(self, toks: ParseResults) -> T.Any:
(8)         s = toks[0].replace(r'\$', '$')
(8)         symbols = [Char(c, self.get_state()) for c in s]
(8)         hlist = Hlist(symbols)
(8)         self.push_state()
(8)         self.get_state().font = mpl.rcParams['mathtext.default']
(8)         return [hlist]
(4)     float_literal = staticmethod(pyparsing_common.convertToFloat)
(4)     def text(self, toks: ParseResults) -> T.Any:
(8)         self.push_state()
(8)         state = self.get_state()
(8)         state.font = 'rm'
(8)         hlist = Hlist([Char(c, state) for c in toks[1]])
(8)         self.pop_state()
(8)         return [hlist]
(4)     def _make_space(self, percentage: float) -> Kern:
(8)         state = self.get_state()
(8)         key = (state.font, state.fontsize, state.dpi)
(8)         width = self._em_width_cache.get(key)
(8)         if width is None:
(12)            metrics = state.fontset.get_metrics(
(16)                'it', mpl.rcParams['mathtext.default'], 'm',
(16)                state.fontsize, state.dpi)
(12)            width = metrics.advance
(12)            self._em_width_cache[key] = width
(8)         return Kern(width * percentage)
(4)     _space_widths = {
(8)         r'\,':         0.16667,   # 3/18 em = 3 mu
(8)         r'\thinspace': 0.16667,   # 3/18 em = 3 mu
(8)         r'\/':         0.16667,   # 3/18 em = 3 mu
(8)         r'\>':         0.22222,   # 4/18 em = 4 mu
(8)         r'\:':         0.22222,   # 4/18 em = 4 mu
(8)         r'\;':         0.27778,   # 5/18 em = 5 mu
(8)         r'\ ':         0.33333,   # 6/18 em = 6 mu
(8)         r'~':          0.33333,   # 6/18 em = 6 mu, nonbreakable
(8)         r'\enspace':   0.5,       # 9/18 em = 9 mu
(8)         r'\quad':      1,         # 1 em = 18 mu
(8)         r'\qquad':     2,         # 2 em = 36 mu
(8)         r'\!':         -0.16667,  # -3/18 em = -3 mu
(4)     }
(4)     def space(self, toks: ParseResults) -> T.Any:
(8)         num = self._space_widths[toks["space"]]
(8)         box = self._make_space(num)
(8)         return [box]
(4)     def customspace(self, toks: ParseResults) -> T.Any:
(8)         return [self._make_space(toks["space"])]
(4)     def symbol(self, s: str, loc: int,
(15)                   toks: ParseResults | dict[str, str]) -> T.Any:
(8)         c = toks["sym"]
(8)         if c == "-":
(12)            c = "\N{MINUS SIGN}"
(8)         try:
(12)            char = Char(c, self.get_state())
(8)         except ValueError as err:
(12)            raise ParseFatalException(s, loc,
(38)                                      "Unknown symbol: %s" % c) from err
(8)         if c in self._spaced_symbols:
(12)            prev_char = next((c for c in s[:loc][::-1] if c != ' '), '')
(12)            if (self._in_subscript_or_superscript or (
(20)                    c in self._binary_operators and (
(20)                    len(s[:loc].split()) == 0 or prev_char == '{' or
(20)                    prev_char in self._left_delims))):
(16)                return [char]
(12)            else:
(16)                return [Hlist([self._make_space(0.2),
(31)                               char,
(31)                               self._make_space(0.2)],
(30)                              do_kern=True)]
(8)         elif c in self._punctuation_symbols:
```

```
(12)                        prev_char = next((c for c in s[:loc][::-1] if c != ' '), '')
(12)                        next_char = next((c for c in s[loc + 1:] if c != ' '), '')
(12)                        if c == ',':
(16)                            if prev_char == '{' and next_char == '}':
(20)                                return [char]
(12)                        if c == '.' and prev_char.isdigit() and next_char.isdigit():
(16)                            return [char]
(12)                        else:
(16)                            return [Hlist([char, self._make_space(0.2)], do_kern=True)]
(8)                     return [char]
(4)                 def unknown_symbol(self, s: str, loc: int, toks: ParseResults) -> T.Any:
(8)                     raise ParseFatalException(s, loc, f"Unknown symbol: {toks['name']}")
(4)                 _accent_map = {
(8)                     r'hat':            r'\circumflexaccent',
(8)                     r'breve':          r'\combiningbreve',
(8)                     r'bar':            r'\combiningoverline',
(8)                     r'grave':          r'\combininggraveaccent',
(8)                     r'acute':          r'\combiningacuteaccent',
(8)                     r'tilde':          r'\combiningtilde',
(8)                     r'dot':            r'\combiningdotabove',
(8)                     r'ddot':           r'\combiningdiaeresis',
(8)                     r'dddot':          r'\combiningthreedotsabove',
(8)                     r'ddddot':         r'\combiningfourdotsabove',
(8)                     r'vec':            r'\combiningrightarrowabove',
(8)                     r'"':              r'\combiningdiaeresis',
(8)                     r"`":              r'\combininggraveaccent',
(8)                     r"'":              r'\combiningacuteaccent',
(8)                     r'~':              r'\combiningtilde',
(8)                     r'.':              r'\combiningdotabove',
(8)                     r'^':              r'\circumflexaccent',
(8)                     r'overrightarrow': r'\rightarrow',
(8)                     r'overleftarrow':  r'\leftarrow',
(8)                     r'mathring':       r'\circ',
(4)                 }
(4)                 _wide_accents = set(r"widehat widetilde widebar".split())
(4)                 def accent(self, toks: ParseResults) -> T.Any:
(8)                     state = self.get_state()
(8)                     thickness = state.get_current_underline_thickness()
(8)                     accent = toks["accent"]
(8)                     sym = toks["sym"]
(8)                     accent_box: Node
(8)                     if accent in self._wide_accents:
(12)                        accent_box = AutoWidthChar(
(16)                            '\\' + accent, sym.width, state, char_class=Accent)
(8)                     else:
(12)                        accent_box = Accent(self._accent_map[accent], state)
(8)                     if accent == 'mathring':
(12)                        accent_box.shrink()
(12)                        accent_box.shrink()
(8)                     centered = HCentered([Hbox(sym.width / 4.0), accent_box])
(8)                     centered.hpack(sym.width, 'exactly')
(8)                     return Vlist([
(16)                            centered,
(16)                            Vbox(0., thickness * 2.0),
(16)                            Hlist([sym])
(16)                            ])
(4)                 def function(self, s: str, loc: int, toks: ParseResults) -> T.Any:
(8)                     hlist = self.operatorname(s, loc, toks)
(8)                     hlist.function_name = toks["name"]
(8)                     return hlist
(4)                 def operatorname(self, s: str, loc: int, toks: ParseResults) -> T.Any:
(8)                     self.push_state()
(8)                     state = self.get_state()
(8)                     state.font = 'rm'
(8)                     hlist_list: list[Node] = []
(8)                     name = toks["name"]
(8)                     for c in name:
(12)                        if isinstance(c, Char):
(16)                            c.font = 'rm'
```

```
(16)                        c._update_metrics()
(16)                        hlist_list.append(c)
(12)                    elif isinstance(c, str):
(16)                        hlist_list.append(Char(c, state))
(12)                    else:
(16)                        hlist_list.append(c)
(8)                 next_char_loc = loc + len(name) + 1
(8)                 if isinstance(name, ParseResults):
(12)                    next_char_loc += len('operatorname{}')
(8)                 next_char = next((c for c in s[next_char_loc:] if c != ' '), '')
(8)                 delimiters = self._delims | {'^', '_'}
(8)                 if (next_char not in delimiters and
(16)                        name not in self._overunder_functions):
(12)                    hlist_list += [self._make_space(self._space_widths[r'\,'])]
(8)                 self.pop_state()
(8)                 if next_char in {'^', '_'}:
(12)                    self._in_subscript_or_superscript = True
(8)                 else:
(12)                    self._in_subscript_or_superscript = False
(8)                 return Hlist(hlist_list)
(4)             def start_group(self, toks: ParseResults) -> T.Any:
(8)                 self.push_state()
(8)                 if toks.get("font"):
(12)                    self.get_state().font = toks.get("font")
(8)                 return []
(4)             def group(self, toks: ParseResults) -> T.Any:
(8)                 grp = Hlist(toks.get("group", []))
(8)                 return [grp]
(4)             def required_group(self, toks: ParseResults) -> T.Any:
(8)                 return Hlist(toks.get("group", []))
(4)             optional_group = required_group
(4)             def end_group(self) -> T.Any:
(8)                 self.pop_state()
(8)                 return []
(4)             def unclosed_group(self, s: str, loc: int, toks: ParseResults) -> T.Any:
(8)                 raise ParseFatalException(s, len(s), "Expected '}'")
(4)             def font(self, toks: ParseResults) -> T.Any:
(8)                 self.get_state().font = toks["font"]
(8)                 return []
(4)             def is_overunder(self, nucleus: Node) -> bool:
(8)                 if isinstance(nucleus, Char):
(12)                    return nucleus.c in self._overunder_symbols
(8)                 elif isinstance(nucleus, Hlist) and hasattr(nucleus, 'function_name'):
(12)                    return nucleus.function_name in self._overunder_functions
(8)                 return False
(4)             def is_dropsub(self, nucleus: Node) -> bool:
(8)                 if isinstance(nucleus, Char):
(12)                    return nucleus.c in self._dropsub_symbols
(8)                 return False
(4)             def is_slanted(self, nucleus: Node) -> bool:
(8)                 if isinstance(nucleus, Char):
(12)                    return nucleus.is_slanted()
(8)                 return False
(4)             def subsuper(self, s: str, loc: int, toks: ParseResults) -> T.Any:
(8)                 nucleus = toks.get("nucleus", Hbox(0))
(8)                 subsuper = toks.get("subsuper", [])
(8)                 napostrophes = len(toks.get("apostrophes", []))
(8)                 if not subsuper and not napostrophes:
(12)                    return nucleus
(8)                 sub = super = None
(8)                 while subsuper:
(12)                    op, arg, *subsuper = subsuper
(12)                    if op == '_':
(16)                        if sub is not None:
(20)                            raise ParseFatalException("Double subscript")
(16)                        sub = arg
(12)                    else:
(16)                        if super is not None:
(20)                            raise ParseFatalException("Double superscript")
```

```
(16)                            super = arg
(8)                     state = self.get_state()
(8)                     rule_thickness = state.fontset.get_underline_thickness(
(12)                        state.font, state.fontsize, state.dpi)
(8)                     xHeight = state.fontset.get_xheight(
(12)                        state.font, state.fontsize, state.dpi)
(8)                     if napostrophes:
(12)                        if super is None:
(16)                            super = Hlist([])
(12)                        for i in range(napostrophes):
(16)                            super.children.extend(self.symbol(s, loc, {"sym": "\\prime"}))
(12)                        super.kern()
(12)                        super.hpack()
(8)                     if self.is_overunder(nucleus):
(12)                        vlist = []
(12)                        shift = 0.
(12)                        width = nucleus.width
(12)                        if super is not None:
(16)                            super.shrink()
(16)                            width = max(width, super.width)
(12)                        if sub is not None:
(16)                            sub.shrink()
(16)                            width = max(width, sub.width)
(12)                        vgap = rule_thickness * 3.0
(12)                        if super is not None:
(16)                            hlist = HCentered([super])
(16)                            hlist.hpack(width, 'exactly')
(16)                            vlist.extend([hlist, Vbox(0, vgap)])
(12)                        hlist = HCentered([nucleus])
(12)                        hlist.hpack(width, 'exactly')
(12)                        vlist.append(hlist)
(12)                        if sub is not None:
(16)                            hlist = HCentered([sub])
(16)                            hlist.hpack(width, 'exactly')
(16)                            vlist.extend([Vbox(0, vgap), hlist])
(16)                            shift = hlist.height + vgap + nucleus.depth
(12)                        vlt = Vlist(vlist)
(12)                        vlt.shift_amount = shift
(12)                        result = Hlist([vlt])
(12)                        return [result]
(8)                     last_char = nucleus
(8)                     if isinstance(nucleus, Hlist):
(12)                        new_children = nucleus.children
(12)                        if len(new_children):
(16)                            if (isinstance(new_children[-1], Kern) and
(24)                                    hasattr(new_children[-2], '_metrics')):
(20)                                new_children = new_children[:-1]
(16)                            last_char = new_children[-1]
(16)                            if hasattr(last_char, '_metrics'):
(20)                                last_char.width = last_char._metrics.advance
(12)                        nucleus = Hlist(new_children, do_kern=False)
(8)                     else:
(12)                        if isinstance(nucleus, Char):
(16)                            last_char.width = last_char._metrics.advance
(12)                        nucleus = Hlist([nucleus])
(8)                     constants = _get_font_constant_set(state)
(8)                     lc_height   = last_char.height
(8)                     lc_baseline = 0
(8)                     if self.is_dropsub(last_char):
(12)                        lc_baseline = last_char.depth
(8)                     superkern = constants.delta * xHeight
(8)                     subkern = constants.delta * xHeight
(8)                     if self.is_slanted(last_char):
(12)                        superkern += constants.delta * xHeight
(12)                        superkern += (constants.delta_slanted *
(26)                                      (lc_height - xHeight * 2. / 3.))
(12)                        if self.is_dropsub(last_char):
(16)                            subkern = (3 * constants.delta -
(27)                                        constants.delta_integral) * lc_height
```

```
2018: (16)                          superkern = (3 * constants.delta +
2019: (29)                                       constants.delta_integral) * lc_height
2020: (12)                      else:
2021: (16)                          subkern = 0
2022: (8)                   x: List
2023: (8)                   if super is None:
2024: (12)                      x = Hlist([Kern(subkern), T.cast(Node, sub)])
2025: (12)                      x.shrink()
2026: (12)                      if self.is_dropsub(last_char):
2027: (16)                          shift_down = lc_baseline + constants.subdrop * xHeight
2028: (12)                      else:
2029: (16)                          shift_down = constants.sub1 * xHeight
2030: (12)                      x.shift_amount = shift_down
2031: (8)                   else:
2032: (12)                      x = Hlist([Kern(superkern), super])
2033: (12)                      x.shrink()
2034: (12)                      if self.is_dropsub(last_char):
2035: (16)                          shift_up = lc_height - constants.subdrop * xHeight
2036: (12)                      else:
2037: (16)                          shift_up = constants.sup1 * xHeight
2038: (12)                      if sub is None:
2039: (16)                          x.shift_amount = -shift_up
2040: (12)                      else:  # Both sub and superscript
2041: (16)                          y = Hlist([Kern(subkern), sub])
2042: (16)                          y.shrink()
2043: (16)                          if self.is_dropsub(last_char):
2044: (20)                              shift_down = lc_baseline + constants.subdrop * xHeight
2045: (16)                          else:
2046: (20)                              shift_down = constants.sub2 * xHeight
2047: (16)                          clr = (2.0 * rule_thickness -
2048: (23)                                 ((shift_up - x.depth) - (y.height - shift_down)))
2049: (16)                          if clr > 0.:
2050: (20)                              shift_up += clr
2051: (16)                          x = Vlist([
2052: (20)                              x,
2053: (20)                              Kern((shift_up - x.depth) - (y.height - shift_down)),
2054: (20)                              y])
2055: (16)                          x.shift_amount = shift_down
2056: (8)                   if not self.is_dropsub(last_char):
2057: (12)                      x.width += constants.script_space * xHeight
2058: (8)                   spaced_nucleus = [nucleus, x]
2059: (8)                   if self._in_subscript_or_superscript:
2060: (12)                      spaced_nucleus += [self._make_space(self._space_widths[r'\,'])]
2061: (12)                      self._in_subscript_or_superscript = False
2062: (8)                   result = Hlist(spaced_nucleus)
2063: (8)                   return [result]
2064: (4)               def _genfrac(self, ldelim: str, rdelim: str, rule: float | None, style:
_MathStyle,
2065: (17)                          num: Hlist, den: Hlist) -> T.Any:
2066: (8)                   state = self.get_state()
2067: (8)                   thickness = state.get_current_underline_thickness()
2068: (8)                   for _ in range(style.value):
2069: (12)                      num.shrink()
2070: (12)                      den.shrink()
2071: (8)                   cnum = HCentered([num])
2072: (8)                   cden = HCentered([den])
2073: (8)                   width = max(num.width, den.width)
2074: (8)                   cnum.hpack(width, 'exactly')
2075: (8)                   cden.hpack(width, 'exactly')
2076: (8)                   vlist = Vlist([cnum,                      # numerator
2077: (23)                                 Vbox(0, thickness * 2.0),  # space
2078: (23)                                 Hrule(state, rule),        # rule
2079: (23)                                 Vbox(0, thickness * 2.0),  # space
2080: (23)                                 cden                       # denominator
2081: (23)                                 ])
2082: (8)                   metrics = state.fontset.get_metrics(
2083: (12)                      state.font, mpl.rcParams['mathtext.default'],
2084: (12)                      '=', state.fontsize, state.dpi)
2085: (8)                   shift = (cden.height -
```

```
2086: (17)                          ((metrics.ymax + metrics.ymin) / 2 -
2087: (18)                           thickness * 3.0))
2088: (8)                   vlist.shift_amount = shift
2089: (8)                   result = [Hlist([vlist, Hbox(thickness * 2.)])]
2090: (8)                   if ldelim or rdelim:
2091: (12)                      if ldelim == '':
2092: (16)                          ldelim = '.'
2093: (12)                      if rdelim == '':
2094: (16)                          rdelim = '.'
2095: (12)                      return self._auto_sized_delimiter(ldelim,
2096: (46)                                                        T.cast(list[T.Union[Box, Char,
str]],
2097: (53)                                                               result),
2098: (46)                                                        rdelim)
2099: (8)                   return result
2100: (4)               def style_literal(self, toks: ParseResults) -> T.Any:
2101: (8)                   return self._MathStyle(int(toks["style_literal"]))
2102: (4)               def genfrac(self, toks: ParseResults) -> T.Any:
2103: (8)                   return self._genfrac(
2104: (12)                      toks.get("ldelim", ""), toks.get("rdelim", ""),
2105: (12)                      toks["rulesize"], toks.get("style", self._MathStyle.TEXTSTYLE),
2106: (12)                      toks["num"], toks["den"])
2107: (4)               def frac(self, toks: ParseResults) -> T.Any:
2108: (8)                   return self._genfrac(
2109: (12)                      "", "", self.get_state().get_current_underline_thickness(),
2110: (12)                      self._MathStyle.TEXTSTYLE, toks["num"], toks["den"])
2111: (4)               def dfrac(self, toks: ParseResults) -> T.Any:
2112: (8)                   return self._genfrac(
2113: (12)                      "", "", self.get_state().get_current_underline_thickness(),
2114: (12)                      self._MathStyle.DISPLAYSTYLE, toks["num"], toks["den"])
2115: (4)               def binom(self, toks: ParseResults) -> T.Any:
2116: (8)                   return self._genfrac(
2117: (12)                      "(", ")", 0,
2118: (12)                      self._MathStyle.TEXTSTYLE, toks["num"], toks["den"])
2119: (4)               def _genset(self, s: str, loc: int, toks: ParseResults) -> T.Any:
2120: (8)                   annotation = toks["annotation"]
2121: (8)                   body = toks["body"]
2122: (8)                   thickness = self.get_state().get_current_underline_thickness()
2123: (8)                   annotation.shrink()
2124: (8)                   centered_annotation = HCentered([annotation])
2125: (8)                   centered_body = HCentered([body])
2126: (8)                   width = max(centered_annotation.width, centered_body.width)
2127: (8)                   centered_annotation.hpack(width, 'exactly')
2128: (8)                   centered_body.hpack(width, 'exactly')
2129: (8)                   vgap = thickness * 3
2130: (8)                   if s[loc + 1] == "u":  # \underset
2131: (12)                      vlist = Vlist([
2132: (16)                          centered_body,               # body
2133: (16)                          Vbox(0, vgap),               # space
2134: (16)                          centered_annotation          # annotation
2135: (12)                      ])
2136: (12)                      vlist.shift_amount = centered_body.depth +
centered_annotation.height + vgap
2137: (8)                   else:  # \overset
2138: (12)                      vlist = Vlist([
2139: (16)                          centered_annotation,         # annotation
2140: (16)                          Vbox(0, vgap),               # space
2141: (16)                          centered_body                # body
2142: (12)                      ])
2143: (8)                   return vlist
2144: (4)               overset = underset = _genset
2145: (4)               def sqrt(self, toks: ParseResults) -> T.Any:
2146: (8)                   root = toks.get("root")
2147: (8)                   body = toks["value"]
2148: (8)                   state = self.get_state()
2149: (8)                   thickness = state.get_current_underline_thickness()
2150: (8)                   height = body.height - body.shift_amount + thickness * 5.0
2151: (8)                   depth = body.depth + body.shift_amount
2152: (8)                   check = AutoHeightChar(r'\__sqrt__', height, depth, state,
```

```
always=True)
                    height = check.height - check.shift_amount
                    depth = check.depth + check.shift_amount
                    padded_body = Hlist([Hbox(2 * thickness), body, Hbox(2 * thickness)])
                    rightside = Vlist([Hrule(state), Glue('fill'), padded_body])
                    rightside.vpack(height + (state.fontsize * state.dpi) / (100.0 *
12.0),
                                    'exactly', depth)
                    if not root:
                        root = Box(check.width * 0.5, 0., 0.)
                    else:
                        root = Hlist(root)
                        root.shrink()
                        root.shrink()
                    root_vlist = Vlist([Hlist([root])])
                    root_vlist.shift_amount = -height * 0.6
                    hlist = Hlist([root_vlist,               # Root
                                   Kern(-check.width * 0.5),
                                   check,                    # Check
                                   rightside])               # Body
                    return [hlist]
                def overline(self, toks: ParseResults) -> T.Any:
                    body = toks["body"]
                    state = self.get_state()
                    thickness = state.get_current_underline_thickness()
                    height = body.height - body.shift_amount + thickness * 3.0
                    depth = body.depth + body.shift_amount
                    rightside = Vlist([Hrule(state), Glue('fill'), Hlist([body])])
                    rightside.vpack(height + (state.fontsize * state.dpi) / (100.0 *
12.0),
                                    'exactly', depth)
                    hlist = Hlist([rightside])
                    return [hlist]
                def _auto_sized_delimiter(self, front: str,
                                          middle: list[Box | Char | str],
                                          back: str) -> T.Any:
                    state = self.get_state()
                    if len(middle):
                        height = max([x.height for x in middle if not isinstance(x, str)])
                        depth = max([x.depth for x in middle if not isinstance(x, str)])
                        factor = None
                        for idx, el in enumerate(middle):
                            if el == r'\middle':
                                c = T.cast(str, middle[idx + 1])  # Should be one of
p.delims.
                                if c != '.':
                                    middle[idx + 1] = AutoHeightChar(
                                            c, height, depth, state, factor=factor)
                                else:
                                    middle.remove(c)
                                del middle[idx]
                        middle_part = T.cast(list[T.Union[Box, Char]], middle)
                    else:
                        height = 0
                        depth = 0
                        factor = 1.0
                        middle_part = []
                    parts: list[Node] = []
                    if front != '.':
                        parts.append(
                            AutoHeightChar(front, height, depth, state, factor=factor))
                    parts.extend(middle_part)
                    if back != '.':
                        parts.append(
                            AutoHeightChar(back, height, depth, state, factor=factor))
                    hlist = Hlist(parts)
                    return hlist
                def auto_delim(self, toks: ParseResults) -> T.Any:
                    return self._auto_sized_delimiter(
```

```
2218: (12)                      toks["left"],
2219: (12)                      toks["mid"].asList() if "mid" in toks else [],
2220: (12)                      toks["right"])
2221: (4)               def boldsymbol(self, toks: ParseResults) -> T.Any:
2222: (8)                   self.push_state()
2223: (8)                   state = self.get_state()
2224: (8)                   hlist: list[Node] = []
2225: (8)                   name = toks["value"]
2226: (8)                   for c in name:
2227: (12)                      if isinstance(c, Hlist):
2228: (16)                          k = c.children[1]
2229: (16)                          if isinstance(k, Char):
2230: (20)                              k.font = "bf"
2231: (20)                              k._update_metrics()
2232: (16)                          hlist.append(c)
2233: (12)                      elif isinstance(c, Char):
2234: (16)                          c.font = "bf"
2235: (16)                          if (c.c in self._latin_alphabets or
2236: (19)                             c.c[1:] in self._small_greek):
2237: (20)                              c.font = "bfit"
2238: (20)                              c._update_metrics()
2239: (16)                          c._update_metrics()
2240: (16)                          hlist.append(c)
2241: (12)                      else:
2242: (16)                          hlist.append(c)
2243: (8)                   self.pop_state()
2244: (8)                   return Hlist(hlist)
2245: (4)               def substack(self, toks: ParseResults) -> T.Any:
2246: (8)                   parts = toks["parts"]
2247: (8)                   state = self.get_state()
2248: (8)                   thickness = state.get_current_underline_thickness()
2249: (8)                   hlist = [Hlist(k) for k in parts[0]]
2250: (8)                   max_width = max(map(lambda c: c.width, hlist))
2251: (8)                   vlist = []
2252: (8)                   for sub in hlist:
2253: (12)                      cp = HCentered([sub])
2254: (12)                      cp.hpack(max_width, 'exactly')
2255: (12)                      vlist.append(cp)
2256: (8)                   stack = [val
2257: (17)                           for pair in zip(vlist, [Vbox(0, thickness * 2)] * len(vlist))
2258: (17)                           for val in pair]
2259: (8)                   del stack[-1]
2260: (8)                   vlt = Vlist(stack)
2261: (8)                   result = [Hlist([vlt])]
2262: (8)                   return result
```

File 15 - _mathtext_data.py:

```
1: (0)              """
2: (0)              font data tables for truetype and afm computer modern fonts
3: (0)              """
4: (0)              from __future__ import annotations
5: (0)              from typing import overload, Union
6: (0)              latex_to_bakoma = {
7: (4)                  '\\__sqrt__'                : ('cmex10', 0x70),
8: (4)                  '\\bigcap'                  : ('cmex10', 0x5c),
9: (4)                  '\\bigcup'                  : ('cmex10', 0x5b),
10: (4)                 '\\bigodot'                 : ('cmex10', 0x4b),
11: (4)                 '\\bigoplus'                : ('cmex10', 0x4d),
12: (4)                 '\\bigotimes'               : ('cmex10', 0x4f),
13: (4)                 '\\biguplus'                : ('cmex10', 0x5d),
14: (4)                 '\\bigvee'                  : ('cmex10', 0x5f),
15: (4)                 '\\bigwedge'                : ('cmex10', 0x5e),
16: (4)                 '\\coprod'                  : ('cmex10', 0x61),
17: (4)                 '\\int'                     : ('cmex10', 0x5a),
18: (4)                 '\\langle'                  : ('cmex10', 0xad),
19: (4)                 '\\leftangle'               : ('cmex10', 0xad),
```

```
 20: (4)                  '\\leftbrace'                : ('cmex10', 0xa9),
 21: (4)                  '\\oint'                     : ('cmex10', 0x49),
 22: (4)                  '\\prod'                     : ('cmex10', 0x59),
 23: (4)                  '\\rangle'                   : ('cmex10', 0xae),
 24: (4)                  '\\rightangle'               : ('cmex10', 0xae),
 25: (4)                  '\\rightbrace'               : ('cmex10', 0xaa),
 26: (4)                  '\\sum'                      : ('cmex10', 0x58),
 27: (4)                  '\\widehat'                  : ('cmex10', 0x62),
 28: (4)                  '\\widetilde'                : ('cmex10', 0x65),
 29: (4)                  '\\{'                        : ('cmex10', 0xa9),
 30: (4)                  '\\}'                        : ('cmex10', 0xaa),
 31: (4)                  '{'                          : ('cmex10', 0xa9),
 32: (4)                  '}'                          : ('cmex10', 0xaa),
 33: (4)                  ','                          : ('cmmi10', 0x3b),
 34: (4)                  '.'                          : ('cmmi10', 0x3a),
 35: (4)                  '/'                          : ('cmmi10', 0x3d),
 36: (4)                  '<'                          : ('cmmi10', 0x3c),
 37: (4)                  '>'                          : ('cmmi10', 0x3e),
 38: (4)                  '\\alpha'                    : ('cmmi10', 0xae),
 39: (4)                  '\\beta'                     : ('cmmi10', 0xaf),
 40: (4)                  '\\chi'                      : ('cmmi10', 0xc2),
 41: (4)                  '\\combiningrightarrowabove' : ('cmmi10', 0x7e),
 42: (4)                  '\\delta'                    : ('cmmi10', 0xb1),
 43: (4)                  '\\ell'                      : ('cmmi10', 0x60),
 44: (4)                  '\\epsilon'                  : ('cmmi10', 0xb2),
 45: (4)                  '\\eta'                      : ('cmmi10', 0xb4),
 46: (4)                  '\\flat'                     : ('cmmi10', 0x5b),
 47: (4)                  '\\frown'                    : ('cmmi10', 0x5f),
 48: (4)                  '\\gamma'                    : ('cmmi10', 0xb0),
 49: (4)                  '\\imath'                    : ('cmmi10', 0x7b),
 50: (4)                  '\\iota'                     : ('cmmi10', 0xb6),
 51: (4)                  '\\jmath'                    : ('cmmi10', 0x7c),
 52: (4)                  '\\kappa'                    : ('cmmi10', 0x2219),
 53: (4)                  '\\lambda'                   : ('cmmi10', 0xb8),
 54: (4)                  '\\leftharpoondown'          : ('cmmi10', 0x29),
 55: (4)                  '\\leftharpoonup'            : ('cmmi10', 0x28),
 56: (4)                  '\\mu'                       : ('cmmi10', 0xb9),
 57: (4)                  '\\natural'                  : ('cmmi10', 0x5c),
 58: (4)                  '\\nu'                       : ('cmmi10', 0xba),
 59: (4)                  '\\omega'                    : ('cmmi10', 0x21),
 60: (4)                  '\\phi'                      : ('cmmi10', 0xc1),
 61: (4)                  '\\pi'                       : ('cmmi10', 0xbc),
 62: (4)                  '\\psi'                      : ('cmmi10', 0xc3),
 63: (4)                  '\\rho'                      : ('cmmi10', 0xbd),
 64: (4)                  '\\rightharpoondown'         : ('cmmi10', 0x2b),
 65: (4)                  '\\rightharpoonup'           : ('cmmi10', 0x2a),
 66: (4)                  '\\sharp'                    : ('cmmi10', 0x5d),
 67: (4)                  '\\sigma'                    : ('cmmi10', 0xbe),
 68: (4)                  '\\smile'                    : ('cmmi10', 0x5e),
 69: (4)                  '\\tau'                      : ('cmmi10', 0xbf),
 70: (4)                  '\\theta'                    : ('cmmi10', 0xb5),
 71: (4)                  '\\triangleleft'             : ('cmmi10', 0x2f),
 72: (4)                  '\\triangleright'            : ('cmmi10', 0x2e),
 73: (4)                  '\\upsilon'                  : ('cmmi10', 0xc0),
 74: (4)                  '\\varepsilon'               : ('cmmi10', 0x22),
 75: (4)                  '\\varphi'                   : ('cmmi10', 0x27),
 76: (4)                  '\\varrho'                   : ('cmmi10', 0x25),
 77: (4)                  '\\varsigma'                 : ('cmmi10', 0x26),
 78: (4)                  '\\vartheta'                 : ('cmmi10', 0x23),
 79: (4)                  '\\wp'                       : ('cmmi10', 0x7d),
 80: (4)                  '\\xi'                       : ('cmmi10', 0xbb),
 81: (4)                  '\\zeta'                     : ('cmmi10', 0xb3),
 82: (4)                  '!'                          : ('cmr10', 0x21),
 83: (4)                  '%'                          : ('cmr10', 0x25),
 84: (4)                  '&'                          : ('cmr10', 0x26),
 85: (4)                  '('                          : ('cmr10', 0x28),
 86: (4)                  ')'                          : ('cmr10', 0x29),
 87: (4)                  '+'                          : ('cmr10', 0x2b),
 88: (4)                  '0'                          : ('cmr10', 0x30),
```

```
 89: (4)                  '1'                          : ('cmr10', 0x31),
 90: (4)                  '2'                          : ('cmr10', 0x32),
 91: (4)                  '3'                          : ('cmr10', 0x33),
 92: (4)                  '4'                          : ('cmr10', 0x34),
 93: (4)                  '5'                          : ('cmr10', 0x35),
 94: (4)                  '6'                          : ('cmr10', 0x36),
 95: (4)                  '7'                          : ('cmr10', 0x37),
 96: (4)                  '8'                          : ('cmr10', 0x38),
 97: (4)                  '9'                          : ('cmr10', 0x39),
 98: (4)                  ':'                          : ('cmr10', 0x3a),
 99: (4)                  ';'                          : ('cmr10', 0x3b),
 100: (4)                 '='                          : ('cmr10', 0x3d),
 101: (4)                 '?'                          : ('cmr10', 0x3f),
 102: (4)                 '@'                          : ('cmr10', 0x40),
 103: (4)                 '['                          : ('cmr10', 0x5b),
 104: (4)                 '\\#'                        : ('cmr10', 0x23),
 105: (4)                 '\\$'                        : ('cmr10', 0x24),
 106: (4)                 '\\%'                        : ('cmr10', 0x25),
 107: (4)                 '\\Delta'                    : ('cmr10', 0xa2),
 108: (4)                 '\\Gamma'                    : ('cmr10', 0xa1),
 109: (4)                 '\\Lambda'                   : ('cmr10', 0xa4),
 110: (4)                 '\\Omega'                    : ('cmr10', 0xad),
 111: (4)                 '\\Phi'                      : ('cmr10', 0xa9),
 112: (4)                 '\\Pi'                       : ('cmr10', 0xa6),
 113: (4)                 '\\Psi'                      : ('cmr10', 0xaa),
 114: (4)                 '\\Sigma'                    : ('cmr10', 0xa7),
 115: (4)                 '\\Theta'                    : ('cmr10', 0xa3),
 116: (4)                 '\\Upsilon'                  : ('cmr10', 0xa8),
 117: (4)                 '\\Xi'                       : ('cmr10', 0xa5),
 118: (4)                 '\\circumflexaccent'         : ('cmr10', 0x5e),
 119: (4)                 '\\combiningacuteaccent'     : ('cmr10', 0xb6),
 120: (4)                 '\\combiningbreve'           : ('cmr10', 0xb8),
 121: (4)                 '\\combiningdiaeresis'       : ('cmr10', 0xc4),
 122: (4)                 '\\combiningdotabove'        : ('cmr10', 0x5f),
 123: (4)                 '\\combininggraveaccent'     : ('cmr10', 0xb5),
 124: (4)                 '\\combiningoverline'        : ('cmr10', 0xb9),
 125: (4)                 '\\combiningtilde'           : ('cmr10', 0x7e),
 126: (4)                 '\\leftbracket'              : ('cmr10', 0x5b),
 127: (4)                 '\\leftparen'                : ('cmr10', 0x28),
 128: (4)                 '\\rightbracket'             : ('cmr10', 0x5d),
 129: (4)                 '\\rightparen'               : ('cmr10', 0x29),
 130: (4)                 '\\widebar'                  : ('cmr10', 0xb9),
 131: (4)                 ']'                          : ('cmr10', 0x5d),
 132: (4)                 '*'                          : ('cmsy10', 0xa4),
 133: (4)                 '\N{MINUS SIGN}'             : ('cmsy10', 0xa1),
 134: (4)                 '\\Downarrow'                : ('cmsy10', 0x2b),
 135: (4)                 '\\Im'                       : ('cmsy10', 0x3d),
 136: (4)                 '\\Leftarrow'                : ('cmsy10', 0x28),
 137: (4)                 '\\Leftrightarrow'           : ('cmsy10', 0x2c),
 138: (4)                 '\\P'                        : ('cmsy10', 0x7b),
 139: (4)                 '\\Re'                       : ('cmsy10', 0x3c),
 140: (4)                 '\\Rightarrow'               : ('cmsy10', 0x29),
 141: (4)                 '\\S'                        : ('cmsy10', 0x78),
 142: (4)                 '\\Uparrow'                  : ('cmsy10', 0x2a),
 143: (4)                 '\\Updownarrow'              : ('cmsy10', 0x6d),
 144: (4)                 '\\Vert'                     : ('cmsy10', 0x6b),
 145: (4)                 '\\aleph'                    : ('cmsy10', 0x40),
 146: (4)                 '\\approx'                   : ('cmsy10', 0xbc),
 147: (4)                 '\\ast'                      : ('cmsy10', 0xa4),
 148: (4)                 '\\asymp'                    : ('cmsy10', 0xb3),
 149: (4)                 '\\backslash'                : ('cmsy10', 0x6e),
 150: (4)                 '\\bigcirc'                  : ('cmsy10', 0xb0),
 151: (4)                 '\\bigtriangledown'          : ('cmsy10', 0x35),
 152: (4)                 '\\bigtriangleup'            : ('cmsy10', 0x34),
 153: (4)                 '\\bot'                      : ('cmsy10', 0x3f),
 154: (4)                 '\\bullet'                   : ('cmsy10', 0xb2),
 155: (4)                 '\\cap'                      : ('cmsy10', 0x5c),
 156: (4)                 '\\cdot'                     : ('cmsy10', 0xa2),
 157: (4)                 '\\circ'                     : ('cmsy10', 0xb1),
```

```
(4)                 '\\clubsuit'                : ('cmsy10', 0x7c),
(4)                 '\\cup'                     : ('cmsy10', 0x5b),
(4)                 '\\dag'                     : ('cmsy10', 0x79),
(4)                 '\\dashv'                   : ('cmsy10', 0x61),
(4)                 '\\ddag'                    : ('cmsy10', 0x7a),
(4)                 '\\diamond'                 : ('cmsy10', 0xa6),
(4)                 '\\diamondsuit'             : ('cmsy10', 0x7d),
(4)                 '\\div'                     : ('cmsy10', 0xa5),
(4)                 '\\downarrow'               : ('cmsy10', 0x23),
(4)                 '\\emptyset'                : ('cmsy10', 0x3b),
(4)                 '\\equiv'                   : ('cmsy10', 0xb4),
(4)                 '\\exists'                  : ('cmsy10', 0x39),
(4)                 '\\forall'                  : ('cmsy10', 0x38),
(4)                 '\\geq'                     : ('cmsy10', 0xb8),
(4)                 '\\gg'                      : ('cmsy10', 0xc0),
(4)                 '\\heartsuit'               : ('cmsy10', 0x7e),
(4)                 '\\in'                      : ('cmsy10', 0x32),
(4)                 '\\infty'                   : ('cmsy10', 0x31),
(4)                 '\\lbrace'                  : ('cmsy10', 0x66),
(4)                 '\\lceil'                   : ('cmsy10', 0x64),
(4)                 '\\leftarrow'               : ('cmsy10', 0xc3),
(4)                 '\\leftrightarrow'          : ('cmsy10', 0x24),
(4)                 '\\leq'                     : ('cmsy10', 0x2219),
(4)                 '\\lfloor'                  : ('cmsy10', 0x62),
(4)                 '\\ll'                      : ('cmsy10', 0xbf),
(4)                 '\\mid'                     : ('cmsy10', 0x6a),
(4)                 '\\mp'                      : ('cmsy10', 0xa8),
(4)                 '\\nabla'                   : ('cmsy10', 0x72),
(4)                 '\\nearrow'                 : ('cmsy10', 0x25),
(4)                 '\\neg'                     : ('cmsy10', 0x3a),
(4)                 '\\ni'                      : ('cmsy10', 0x33),
(4)                 '\\nwarrow'                 : ('cmsy10', 0x2d),
(4)                 '\\odot'                    : ('cmsy10', 0xaf),
(4)                 '\\ominus'                  : ('cmsy10', 0xaa),
(4)                 '\\oplus'                   : ('cmsy10', 0xa9),
(4)                 '\\oslash'                  : ('cmsy10', 0xae),
(4)                 '\\otimes'                  : ('cmsy10', 0xad),
(4)                 '\\pm'                      : ('cmsy10', 0xa7),
(4)                 '\\prec'                    : ('cmsy10', 0xc1),
(4)                 '\\preceq'                  : ('cmsy10', 0xb9),
(4)                 '\\prime'                   : ('cmsy10', 0x30),
(4)                 '\\propto'                  : ('cmsy10', 0x2f),
(4)                 '\\rbrace'                  : ('cmsy10', 0x67),
(4)                 '\\rceil'                   : ('cmsy10', 0x65),
(4)                 '\\rfloor'                  : ('cmsy10', 0x63),
(4)                 '\\rightarrow'              : ('cmsy10', 0x21),
(4)                 '\\searrow'                 : ('cmsy10', 0x26),
(4)                 '\\sim'                     : ('cmsy10', 0xbb),
(4)                 '\\simeq'                   : ('cmsy10', 0x27),
(4)                 '\\slash'                   : ('cmsy10', 0x36),
(4)                 '\\spadesuit'               : ('cmsy10', 0xc4),
(4)                 '\\sqcap'                   : ('cmsy10', 0x75),
(4)                 '\\sqcup'                   : ('cmsy10', 0x74),
(4)                 '\\sqsubseteq'              : ('cmsy10', 0x76),
(4)                 '\\sqsupseteq'              : ('cmsy10', 0x77),
(4)                 '\\subset'                  : ('cmsy10', 0xbd),
(4)                 '\\subseteq'                : ('cmsy10', 0xb5),
(4)                 '\\succ'                    : ('cmsy10', 0xc2),
(4)                 '\\succeq'                  : ('cmsy10', 0xba),
(4)                 '\\supset'                  : ('cmsy10', 0xbe),
(4)                 '\\supseteq'                : ('cmsy10', 0xb6),
(4)                 '\\swarrow'                 : ('cmsy10', 0x2e),
(4)                 '\\times'                   : ('cmsy10', 0xa3),
(4)                 '\\to'                      : ('cmsy10', 0x21),
(4)                 '\\top'                     : ('cmsy10', 0x3e),
(4)                 '\\uparrow'                 : ('cmsy10', 0x22),
(4)                 '\\updownarrow'             : ('cmsy10', 0x6c),
(4)                 '\\uplus'                   : ('cmsy10', 0x5d),
(4)                 '\\vdash'                   : ('cmsy10', 0x60),
```

```
(4)                 '\\vee'                     : ('cmsy10', 0x5f),
(4)                 '\\vert'                    : ('cmsy10', 0x6a),
(4)                 '\\wedge'                   : ('cmsy10', 0x5e),
(4)                 '\\wr'                      : ('cmsy10', 0x6f),
(4)                 '\\|'                       : ('cmsy10', 0x6b),
(4)                 '|'                         : ('cmsy10', 0x6a),
(4)                 '\\_'                       : ('cmtt10', 0x5f)
(0)             }
(0)             type12uni = {
(4)                 'aring'          : 229,
(4)                 'quotedblright'  : 8221,
(4)                 'V'              : 86,
(4)                 'dollar'         : 36,
(4)                 'four'           : 52,
(4)                 'Yacute'         : 221,
(4)                 'P'              : 80,
(4)                 'underscore'     : 95,
(4)                 'p'              : 112,
(4)                 'Otilde'         : 213,
(4)                 'perthousand'    : 8240,
(4)                 'zero'           : 48,
(4)                 'dotlessi'       : 305,
(4)                 'Scaron'         : 352,
(4)                 'zcaron'         : 382,
(4)                 'egrave'         : 232,
(4)                 'section'        : 167,
(4)                 'Icircumflex'    : 206,
(4)                 'ntilde'         : 241,
(4)                 'ampersand'      : 38,
(4)                 'dotaccent'      : 729,
(4)                 'degree'         : 176,
(4)                 'K'              : 75,
(4)                 'acircumflex'    : 226,
(4)                 'Aring'          : 197,
(4)                 'k'              : 107,
(4)                 'smalltilde'     : 732,
(4)                 'Agrave'         : 192,
(4)                 'divide'         : 247,
(4)                 'ocircumflex'    : 244,
(4)                 'asciitilde'     : 126,
(4)                 'two'            : 50,
(4)                 'E'              : 69,
(4)                 'scaron'         : 353,
(4)                 'F'              : 70,
(4)                 'bracketleft'    : 91,
(4)                 'asciicircum'    : 94,
(4)                 'f'              : 102,
(4)                 'ordmasculine'   : 186,
(4)                 'mu'             : 181,
(4)                 'paragraph'      : 182,
(4)                 'nine'           : 57,
(4)                 'v'              : 118,
(4)                 'guilsinglleft'  : 8249,
(4)                 'backslash'      : 92,
(4)                 'six'            : 54,
(4)                 'A'              : 65,
(4)                 'icircumflex'    : 238,
(4)                 'a'              : 97,
(4)                 'ogonek'         : 731,
(4)                 'q'              : 113,
(4)                 'oacute'         : 243,
(4)                 'ograve'         : 242,
(4)                 'edieresis'      : 235,
(4)                 'comma'          : 44,
(4)                 'otilde'         : 245,
(4)                 'guillemotright' : 187,
(4)                 'ecircumflex'    : 234,
(4)                 'greater'        : 62,
(4)                 'uacute'         : 250,
```

```
(4)                 'L'              : 76,
(4)                 'bullet'         : 8226,
(4)                 'cedilla'        : 184,
(4)                 'ydieresis'      : 255,
(4)                 'l'              : 108,
(4)                 'logicalnot'     : 172,
(4)                 'exclamdown'     : 161,
(4)                 'endash'         : 8211,
(4)                 'agrave'         : 224,
(4)                 'Adieresis'      : 196,
(4)                 'germandbls'     : 223,
(4)                 'Odieresis'      : 214,
(4)                 'space'          : 32,
(4)                 'quoteright'     : 8217,
(4)                 'ucircumflex'    : 251,
(4)                 'G'              : 71,
(4)                 'quoteleft'      : 8216,
(4)                 'W'              : 87,
(4)                 'Q'              : 81,
(4)                 'g'              : 103,
(4)                 'w'              : 119,
(4)                 'question'       : 63,
(4)                 'one'            : 49,
(4)                 'ring'           : 730,
(4)                 'figuredash'     : 8210,
(4)                 'B'              : 66,
(4)                 'iacute'         : 237,
(4)                 'Ydieresis'      : 376,
(4)                 'R'              : 82,
(4)                 'b'              : 98,
(4)                 'r'              : 114,
(4)                 'Ccedilla'       : 199,
(4)                 'minus'          : 8722,
(4)                 'Lslash'         : 321,
(4)                 'Uacute'         : 218,
(4)                 'yacute'         : 253,
(4)                 'Ucircumflex'    : 219,
(4)                 'quotedbl'       : 34,
(4)                 'onehalf'        : 189,
(4)                 'Thorn'          : 222,
(4)                 'M'              : 77,
(4)                 'eight'          : 56,
(4)                 'multiply'       : 215,
(4)                 'grave'          : 96,
(4)                 'Ocircumflex'    : 212,
(4)                 'm'              : 109,
(4)                 'Ugrave'         : 217,
(4)                 'guilsinglright' : 8250,
(4)                 'Ntilde'         : 209,
(4)                 'questiondown'   : 191,
(4)                 'Atilde'         : 195,
(4)                 'ccedilla'       : 231,
(4)                 'Z'              : 90,
(4)                 'copyright'      : 169,
(4)                 'yen'            : 165,
(4)                 'Eacute'         : 201,
(4)                 'H'              : 72,
(4)                 'X'              : 88,
(4)                 'Idieresis'      : 207,
(4)                 'bar'            : 124,
(4)                 'h'              : 104,
(4)                 'x'              : 120,
(4)                 'udieresis'      : 252,
(4)                 'ordfeminine'    : 170,
(4)                 'braceleft'      : 123,
(4)                 'macron'         : 175,
(4)                 'atilde'         : 227,
(4)                 'Acircumflex'    : 194,
(4)                 'Oslash'         : 216,
```

```
(4)                 'C'              : 67,
(4)                 'quotedblleft'   : 8220,
(4)                 'S'              : 83,
(4)                 'exclam'         : 33,
(4)                 'Zcaron'         : 381,
(4)                 'equal'          : 61,
(4)                 's'              : 115,
(4)                 'eth'            : 240,
(4)                 'Egrave'         : 200,
(4)                 'hyphen'         : 45,
(4)                 'period'         : 46,
(4)                 'igrave'         : 236,
(4)                 'colon'          : 58,
(4)                 'Ecircumflex'    : 202,
(4)                 'trademark'      : 8482,
(4)                 'Aacute'         : 193,
(4)                 'cent'           : 162,
(4)                 'lslash'         : 322,
(4)                 'c'              : 99,
(4)                 'N'              : 78,
(4)                 'breve'          : 728,
(4)                 'Oacute'         : 211,
(4)                 'guillemotleft'  : 171,
(4)                 'n'              : 110,
(4)                 'idieresis'      : 239,
(4)                 'braceright'     : 125,
(4)                 'seven'          : 55,
(4)                 'brokenbar'      : 166,
(4)                 'ugrave'         : 249,
(4)                 'periodcentered' : 183,
(4)                 'sterling'       : 163,
(4)                 'I'              : 73,
(4)                 'Y'              : 89,
(4)                 'Eth'            : 208,
(4)                 'emdash'         : 8212,
(4)                 'i'              : 105,
(4)                 'daggerdbl'      : 8225,
(4)                 'y'              : 121,
(4)                 'plusminus'      : 177,
(4)                 'less'           : 60,
(4)                 'Udieresis'      : 220,
(4)                 'D'              : 68,
(4)                 'five'           : 53,
(4)                 'T'              : 84,
(4)                 'oslash'         : 248,
(4)                 'acute'          : 180,
(4)                 'd'              : 100,
(4)                 'OE'             : 338,
(4)                 'Igrave'         : 204,
(4)                 't'              : 116,
(4)                 'parenright'     : 41,
(4)                 'adieresis'      : 228,
(4)                 'quotesingle'    : 39,
(4)                 'twodotenleader' : 8229,
(4)                 'slash'          : 47,
(4)                 'ellipsis'       : 8230,
(4)                 'numbersign'     : 35,
(4)                 'odieresis'      : 246,
(4)                 'O'              : 79,
(4)                 'oe'             : 339,
(4)                 'o'              : 111,
(4)                 'Edieresis'      : 203,
(4)                 'plus'           : 43,
(4)                 'dagger'         : 8224,
(4)                 'three'          : 51,
(4)                 'hungarumlaut'   : 733,
(4)                 'parenleft'      : 40,
(4)                 'fraction'       : 8260,
(4)                 'registered'     : 174,
```

```
(4)                 'J'              : 74,
(4)                 'dieresis'       : 168,
(4)                 'Ograve'         : 210,
(4)                 'j'              : 106,
(4)                 'z'              : 122,
(4)                 'ae'             : 230,
(4)                 'semicolon'      : 59,
(4)                 'at'             : 64,
(4)                 'Iacute'         : 205,
(4)                 'percent'        : 37,
(4)                 'bracketright'   : 93,
(4)                 'AE'             : 198,
(4)                 'asterisk'       : 42,
(4)                 'aacute'         : 225,
(4)                 'U'              : 85,
(4)                 'eacute'         : 233,
(4)                 'e'              : 101,
(4)                 'thorn'          : 254,
(4)                 'u'              : 117,
(0)             }
(0)             uni2type1 = {v: k for k, v in type12uni.items()}
(0)             tex2uni = {
(4)                 '#'                       : 0x23,
(4)                 '$'                       : 0x24,
(4)                 '%'                       : 0x25,
(4)                 'AA'                      : 0xc5,
(4)                 'AE'                      : 0xc6,
(4)                 'BbbC'                    : 0x2102,
(4)                 'BbbN'                    : 0x2115,
(4)                 'BbbP'                    : 0x2119,
(4)                 'BbbQ'                    : 0x211a,
(4)                 'BbbR'                    : 0x211d,
(4)                 'BbbZ'                    : 0x2124,
(4)                 'Bumpeq'                  : 0x224e,
(4)                 'Cap'                     : 0x22d2,
(4)                 'Colon'                   : 0x2237,
(4)                 'Cup'                     : 0x22d3,
(4)                 'DH'                      : 0xd0,
(4)                 'Delta'                   : 0x394,
(4)                 'Doteq'                   : 0x2251,
(4)                 'Downarrow'               : 0x21d3,
(4)                 'Equiv'                   : 0x2263,
(4)                 'Finv'                    : 0x2132,
(4)                 'Game'                    : 0x2141,
(4)                 'Gamma'                   : 0x393,
(4)                 'H'                       : 0x30b,
(4)                 'Im'                      : 0x2111,
(4)                 'Join'                    : 0x2a1d,
(4)                 'L'                       : 0x141,
(4)                 'Lambda'                  : 0x39b,
(4)                 'Ldsh'                    : 0x21b2,
(4)                 'Leftarrow'               : 0x21d0,
(4)                 'Leftrightarrow'          : 0x21d4,
(4)                 'Lleftarrow'              : 0x21da,
(4)                 'Longleftarrow'           : 0x27f8,
(4)                 'Longleftrightarrow'      : 0x27fa,
(4)                 'Longrightarrow'          : 0x27f9,
(4)                 'Lsh'                     : 0x21b0,
(4)                 'Nearrow'                 : 0x21d7,
(4)                 'Nwarrow'                 : 0x21d6,
(4)                 'O'                       : 0xd8,
(4)                 'OE'                      : 0x152,
(4)                 'Omega'                   : 0x3a9,
(4)                 'P'                       : 0xb6,
(4)                 'Phi'                     : 0x3a6,
(4)                 'Pi'                      : 0x3a0,
(4)                 'Psi'                     : 0x3a8,
(4)                 'QED'                     : 0x220e,
(4)                 'Rdsh'                    : 0x21b3,
```

```
(4)                 'Re'                    : 0x211c,
(4)                 'Rightarrow'            : 0x21d2,
(4)                 'Rrightarrow'           : 0x21db,
(4)                 'Rsh'                   : 0x21b1,
(4)                 'S'                     : 0xa7,
(4)                 'Searrow'               : 0x21d8,
(4)                 'Sigma'                 : 0x3a3,
(4)                 'Subset'                : 0x22d0,
(4)                 'Supset'                : 0x22d1,
(4)                 'Swarrow'               : 0x21d9,
(4)                 'Theta'                 : 0x398,
(4)                 'Thorn'                 : 0xde,
(4)                 'Uparrow'               : 0x21d1,
(4)                 'Updownarrow'           : 0x21d5,
(4)                 'Upsilon'               : 0x3a5,
(4)                 'Vdash'                 : 0x22a9,
(4)                 'Vert'                  : 0x2016,
(4)                 'Vvdash'                : 0x22aa,
(4)                 'Xi'                    : 0x39e,
(4)                 '_'                     : 0x5f,
(4)                 '__sqrt__'              : 0x221a,
(4)                 'aa'                    : 0xe5,
(4)                 'ac'                    : 0x223e,
(4)                 'acute'                 : 0x301,
(4)                 'acwopencirclearrow'    : 0x21ba,
(4)                 'adots'                 : 0x22f0,
(4)                 'ae'                    : 0xe6,
(4)                 'aleph'                 : 0x2135,
(4)                 'alpha'                 : 0x3b1,
(4)                 'amalg'                 : 0x2a3f,
(4)                 'angle'                 : 0x2220,
(4)                 'approx'                : 0x2248,
(4)                 'approxeq'              : 0x224a,
(4)                 'approxident'           : 0x224b,
(4)                 'arceq'                 : 0x2258,
(4)                 'ast'                   : 0x2217,
(4)                 'asterisk'              : 0x2a,
(4)                 'asymp'                 : 0x224d,
(4)                 'backcong'              : 0x224c,
(4)                 'backepsilon'           : 0x3f6,
(4)                 'backprime'             : 0x2035,
(4)                 'backsim'               : 0x223d,
(4)                 'backsimeq'             : 0x22cd,
(4)                 'backslash'             : 0x5c,
(4)                 'bagmember'             : 0x22ff,
(4)                 'bar'                   : 0x304,
(4)                 'barleftarrow'          : 0x21e4,
(4)                 'barvee'                : 0x22bd,
(4)                 'barwedge'              : 0x22bc,
(4)                 'because'               : 0x2235,
(4)                 'beta'                  : 0x3b2,
(4)                 'beth'                  : 0x2136,
(4)                 'between'               : 0x226c,
(4)                 'bigcap'                : 0x22c2,
(4)                 'bigcirc'               : 0x25cb,
(4)                 'bigcup'                : 0x22c3,
(4)                 'bigodot'               : 0x2a00,
(4)                 'bigoplus'              : 0x2a01,
(4)                 'bigotimes'             : 0x2a02,
(4)                 'bigsqcup'              : 0x2a06,
(4)                 'bigstar'               : 0x2605,
(4)                 'bigtriangledown'       : 0x25bd,
(4)                 'bigtriangleup'         : 0x25b3,
(4)                 'biguplus'              : 0x2a04,
(4)                 'bigvee'                : 0x22c1,
(4)                 'bigwedge'              : 0x22c0,
(4)                 'blacksquare'           : 0x25a0,
(4)                 'blacktriangle'         : 0x25b4,
(4)                 'blacktriangledown'     : 0x25be,
```

```
572: (4)                 'blacktriangleleft'      : 0x25c0,
573: (4)                 'blacktriangleright'     : 0x25b6,
574: (4)                 'bot'                    : 0x22a5,
575: (4)                 'bowtie'                 : 0x22c8,
576: (4)                 'boxbar'                 : 0x25eb,
577: (4)                 'boxdot'                 : 0x22a1,
578: (4)                 'boxminus'               : 0x229f,
579: (4)                 'boxplus'                : 0x229e,
580: (4)                 'boxtimes'               : 0x22a0,
581: (4)                 'breve'                  : 0x306,
582: (4)                 'bullet'                 : 0x2219,
583: (4)                 'bumpeq'                 : 0x224f,
584: (4)                 'c'                      : 0x327,
585: (4)                 'candra'                 : 0x310,
586: (4)                 'cap'                    : 0x2229,
587: (4)                 'carriagereturn'         : 0x21b5,
588: (4)                 'cdot'                   : 0x22c5,
589: (4)                 'cdotp'                  : 0xb7,
590: (4)                 'cdots'                  : 0x22ef,
591: (4)                 'cent'                   : 0xa2,
592: (4)                 'check'                  : 0x30c,
593: (4)                 'checkmark'              : 0x2713,
594: (4)                 'chi'                    : 0x3c7,
595: (4)                 'circ'                   : 0x2218,
596: (4)                 'circeq'                 : 0x2257,
597: (4)                 'circlearrowleft'        : 0x21ba,
598: (4)                 'circlearrowright'       : 0x21bb,
599: (4)                 'circledR'               : 0xae,
600: (4)                 'circledS'               : 0x24c8,
601: (4)                 'circledast'             : 0x229b,
602: (4)                 'circledcirc'            : 0x229a,
603: (4)                 'circleddash'            : 0x229d,
604: (4)                 'circumflexaccent'       : 0x302,
605: (4)                 'clubsuit'               : 0x2663,
606: (4)                 'clubsuitopen'           : 0x2667,
607: (4)                 'colon'                  : 0x3a,
608: (4)                 'coloneq'                : 0x2254,
609: (4)                 'combiningacuteaccent'   : 0x301,
610: (4)                 'combiningbreve'         : 0x306,
611: (4)                 'combiningdiaeresis'     : 0x308,
612: (4)                 'combiningdotabove'      : 0x307,
613: (4)                 'combiningfourdotsabove' : 0x20dc,
614: (4)                 'combininggraveaccent'   : 0x300,
615: (4)                 'combiningoverline'      : 0x304,
616: (4)                 'combiningrightarrowabove' : 0x20d7,
617: (4)                 'combiningthreedotsabove' : 0x20db,
618: (4)                 'combiningtilde'         : 0x303,
619: (4)                 'complement'             : 0x2201,
620: (4)                 'cong'                   : 0x2245,
621: (4)                 'coprod'                 : 0x2210,
622: (4)                 'copyright'              : 0xa9,
623: (4)                 'cup'                    : 0x222a,
624: (4)                 'cupdot'                 : 0x228d,
625: (4)                 'cupleftarrow'           : 0x228c,
626: (4)                 'curlyeqprec'            : 0x22de,
627: (4)                 'curlyeqsucc'            : 0x22df,
628: (4)                 'curlyvee'               : 0x22ce,
629: (4)                 'curlywedge'             : 0x22cf,
630: (4)                 'curvearrowleft'         : 0x21b6,
631: (4)                 'curvearrowright'        : 0x21b7,
632: (4)                 'cwopencirclearrow'      : 0x21bb,
633: (4)                 'd'                      : 0x323,
634: (4)                 'dag'                    : 0x2020,
635: (4)                 'dagger'                 : 0x2020,
636: (4)                 'daleth'                 : 0x2138,
637: (4)                 'danger'                 : 0x2621,
638: (4)                 'dashleftarrow'          : 0x290e,
639: (4)                 'dashrightarrow'         : 0x290f,
640: (4)                 'dashv'                  : 0x22a3,
```

```
(4)                 'ddag'                  : 0x2021,
(4)                 'ddagger'               : 0x2021,
(4)                 'ddddot'                : 0x20dc,
(4)                 'dddot'                 : 0x20db,
(4)                 'ddot'                  : 0x308,
(4)                 'ddots'                 : 0x22f1,
(4)                 'degree'                : 0xb0,
(4)                 'delta'                 : 0x3b4,
(4)                 'dh'                    : 0xf0,
(4)                 'diamond'               : 0x22c4,
(4)                 'diamondsuit'           : 0x2662,
(4)                 'digamma'               : 0x3dd,
(4)                 'disin'                 : 0x22f2,
(4)                 'div'                   : 0xf7,
(4)                 'divideontimes'         : 0x22c7,
(4)                 'dot'                   : 0x307,
(4)                 'doteq'                 : 0x2250,
(4)                 'doteqdot'              : 0x2251,
(4)                 'dotminus'              : 0x2238,
(4)                 'dotplus'               : 0x2214,
(4)                 'dots'                  : 0x2026,
(4)                 'dotsminusdots'         : 0x223a,
(4)                 'doublebarwedge'        : 0x2306,
(4)                 'downarrow'             : 0x2193,
(4)                 'downdownarrows'        : 0x21ca,
(4)                 'downharpoonleft'       : 0x21c3,
(4)                 'downharpoonright'      : 0x21c2,
(4)                 'downzigzagarrow'       : 0x21af,
(4)                 'ell'                   : 0x2113,
(4)                 'emdash'                : 0x2014,
(4)                 'emptyset'              : 0x2205,
(4)                 'endash'                : 0x2013,
(4)                 'epsilon'               : 0x3b5,
(4)                 'eqcirc'                : 0x2256,
(4)                 'eqcolon'               : 0x2255,
(4)                 'eqdef'                 : 0x225d,
(4)                 'eqgtr'                 : 0x22dd,
(4)                 'eqless'                : 0x22dc,
(4)                 'eqsim'                 : 0x2242,
(4)                 'eqslantgtr'            : 0x2a96,
(4)                 'eqslantless'           : 0x2a95,
(4)                 'equal'                 : 0x3d,
(4)                 'equalparallel'         : 0x22d5,
(4)                 'equiv'                 : 0x2261,
(4)                 'eta'                   : 0x3b7,
(4)                 'eth'                   : 0xf0,
(4)                 'exists'                : 0x2203,
(4)                 'fallingdotseq'         : 0x2252,
(4)                 'flat'                  : 0x266d,
(4)                 'forall'                : 0x2200,
(4)                 'frakC'                 : 0x212d,
(4)                 'frakZ'                 : 0x2128,
(4)                 'frown'                 : 0x2322,
(4)                 'gamma'                 : 0x3b3,
(4)                 'geq'                   : 0x2265,
(4)                 'geqq'                  : 0x2267,
(4)                 'geqslant'              : 0x2a7e,
(4)                 'gg'                    : 0x226b,
(4)                 'ggg'                   : 0x22d9,
(4)                 'gimel'                 : 0x2137,
(4)                 'gnapprox'              : 0x2a8a,
(4)                 'gneqq'                 : 0x2269,
(4)                 'gnsim'                 : 0x22e7,
(4)                 'grave'                 : 0x300,
(4)                 'greater'               : 0x3e,
(4)                 'gtrapprox'             : 0x2a86,
(4)                 'gtrdot'                : 0x22d7,
(4)                 'gtreqless'             : 0x22db,
(4)                 'gtreqless'             : 0x2a8c,
```

```
710: (4)                'gtrless'               : 0x2277,
711: (4)                'gtrsim'                : 0x2273,
712: (4)                'guillemotleft'         : 0xab,
713: (4)                'guillemotright'        : 0xbb,
714: (4)                'guilsinglleft'         : 0x2039,
715: (4)                'guilsinglright'        : 0x203a,
716: (4)                'hat'                   : 0x302,
717: (4)                'hbar'                  : 0x127,
718: (4)                'heartsuit'             : 0x2661,
719: (4)                'hermitmatrix'          : 0x22b9,
720: (4)                'hookleftarrow'         : 0x21a9,
721: (4)                'hookrightarrow'        : 0x21aa,
722: (4)                'hslash'                : 0x210f,
723: (4)                'i'                     : 0x131,
724: (4)                'iiiint'                : 0x2a0c,
725: (4)                'iiint'                 : 0x222d,
726: (4)                'iint'                  : 0x222c,
727: (4)                'imageof'               : 0x22b7,
728: (4)                'imath'                 : 0x131,
729: (4)                'in'                    : 0x2208,
730: (4)                'increment'             : 0x2206,
731: (4)                'infty'                 : 0x221e,
732: (4)                'int'                   : 0x222b,
733: (4)                'intercal'              : 0x22ba,
734: (4)                'invnot'                : 0x2310,
735: (4)                'iota'                  : 0x3b9,
736: (4)                'isinE'                 : 0x22f9,
737: (4)                'isindot'               : 0x22f5,
738: (4)                'isinobar'              : 0x22f7,
739: (4)                'isins'                 : 0x22f4,
740: (4)                'isinvb'                : 0x22f8,
741: (4)                'jmath'                 : 0x237,
742: (4)                'k'                     : 0x328,
743: (4)                'kappa'                 : 0x3ba,
744: (4)                'kernelcontraction'     : 0x223b,
745: (4)                'l'                     : 0x142,
746: (4)                'lambda'                : 0x3bb,
747: (4)                'lambdabar'             : 0x19b,
748: (4)                'langle'                : 0x27e8,
749: (4)                'lasp'                  : 0x2bd,
750: (4)                'lbrace'                : 0x7b,
751: (4)                'lbrack'                : 0x5b,
752: (4)                'lceil'                 : 0x2308,
753: (4)                'ldots'                 : 0x2026,
754: (4)                'leadsto'               : 0x21dd,
755: (4)                'leftarrow'             : 0x2190,
756: (4)                'leftarrowtail'         : 0x21a2,
757: (4)                'leftbrace'             : 0x7b,
758: (4)                'leftharpoonaccent'     : 0x20d0,
759: (4)                'leftharpoondown'       : 0x21bd,
760: (4)                'leftharpoonup'         : 0x21bc,
761: (4)                'leftleftarrows'        : 0x21c7,
762: (4)                'leftparen'             : 0x28,
763: (4)                'leftrightarrow'        : 0x2194,
764: (4)                'leftrightarrows'       : 0x21c6,
765: (4)                'leftrightharpoons'     : 0x21cb,
766: (4)                'leftrightsquigarrow'   : 0x21ad,
767: (4)                'leftsquigarrow'        : 0x219c,
768: (4)                'leftthreetimes'        : 0x22cb,
769: (4)                'leq'                   : 0x2264,
770: (4)                'leqq'                  : 0x2266,
771: (4)                'leqslant'              : 0x2a7d,
772: (4)                'less'                  : 0x3c,
773: (4)                'lessapprox'            : 0x2a85,
774: (4)                'lessdot'               : 0x22d6,
775: (4)                'lesseqgtr'             : 0x22da,
776: (4)                'lesseqqgtr'            : 0x2a8b,
777: (4)                'lessgtr'               : 0x2276,
778: (4)                'lesssim'               : 0x2272,
```

```
(4)                 'lfloor'                 : 0x230a,
(4)                 'lgroup'                 : 0x27ee,
(4)                 'lhd'                    : 0x25c1,
(4)                 'll'                     : 0x226a,
(4)                 'llcorner'               : 0x231e,
(4)                 'lll'                    : 0x22d8,
(4)                 'lnapprox'               : 0x2a89,
(4)                 'lneqq'                  : 0x2268,
(4)                 'lnsim'                  : 0x22e6,
(4)                 'longleftarrow'          : 0x27f5,
(4)                 'longleftrightarrow'     : 0x27f7,
(4)                 'longmapsto'             : 0x27fc,
(4)                 'longrightarrow'         : 0x27f6,
(4)                 'looparrowleft'          : 0x21ab,
(4)                 'looparrowright'         : 0x21ac,
(4)                 'lq'                     : 0x2018,
(4)                 'lrcorner'               : 0x231f,
(4)                 'ltimes'                 : 0x22c9,
(4)                 'macron'                 : 0xaf,
(4)                 'maltese'                : 0x2720,
(4)                 'mapsdown'               : 0x21a7,
(4)                 'mapsfrom'               : 0x21a4,
(4)                 'mapsto'                 : 0x21a6,
(4)                 'mapsup'                 : 0x21a5,
(4)                 'measeq'                 : 0x225e,
(4)                 'measuredangle'          : 0x2221,
(4)                 'measuredrightangle'     : 0x22be,
(4)                 'merge'                  : 0x2a55,
(4)                 'mho'                    : 0x2127,
(4)                 'mid'                    : 0x2223,
(4)                 'minus'                  : 0x2212,
(4)                 'minuscolon'             : 0x2239,
(4)                 'models'                 : 0x22a7,
(4)                 'mp'                     : 0x2213,
(4)                 'mu'                     : 0x3bc,
(4)                 'multimap'               : 0x22b8,
(4)                 'nLeftarrow'             : 0x21cd,
(4)                 'nLeftrightarrow'        : 0x21ce,
(4)                 'nRightarrow'            : 0x21cf,
(4)                 'nVDash'                 : 0x22af,
(4)                 'nVdash'                 : 0x22ae,
(4)                 'nabla'                  : 0x2207,
(4)                 'napprox'                : 0x2249,
(4)                 'natural'                : 0x266e,
(4)                 'ncong'                  : 0x2247,
(4)                 'ne'                     : 0x2260,
(4)                 'nearrow'                : 0x2197,
(4)                 'neg'                    : 0xac,
(4)                 'neq'                    : 0x2260,
(4)                 'nequiv'                 : 0x2262,
(4)                 'nexists'                : 0x2204,
(4)                 'ngeq'                   : 0x2271,
(4)                 'ngtr'                   : 0x226f,
(4)                 'ngtrless'               : 0x2279,
(4)                 'ngtrsim'                : 0x2275,
(4)                 'ni'                     : 0x220b,
(4)                 'niobar'                 : 0x22fe,
(4)                 'nis'                    : 0x22fc,
(4)                 'nisd'                   : 0x22fa,
(4)                 'nleftarrow'             : 0x219a,
(4)                 'nleftrightarrow'        : 0x21ae,
(4)                 'nleq'                   : 0x2270,
(4)                 'nless'                  : 0x226e,
(4)                 'nlessgtr'               : 0x2278,
(4)                 'nlesssim'               : 0x2274,
(4)                 'nmid'                   : 0x2224,
(4)                 'not'                    : 0x338,
(4)                 'notin'                  : 0x2209,
(4)                 'notsmallowns'           : 0x220c,
```

```
(4)                 'nparallel'              : 0x2226,
(4)                 'nprec'                  : 0x2280,
(4)                 'npreccurlyeq'           : 0x22e0,
(4)                 'nrightarrow'            : 0x219b,
(4)                 'nsim'                   : 0x2241,
(4)                 'nsimeq'                 : 0x2244,
(4)                 'nsqsubseteq'            : 0x22e2,
(4)                 'nsqsupseteq'            : 0x22e3,
(4)                 'nsubset'                : 0x2284,
(4)                 'nsubseteq'              : 0x2288,
(4)                 'nsucc'                  : 0x2281,
(4)                 'nsucccurlyeq'           : 0x22e1,
(4)                 'nsupset'                : 0x2285,
(4)                 'nsupseteq'              : 0x2289,
(4)                 'ntriangleleft'          : 0x22ea,
(4)                 'ntrianglelefteq'        : 0x22ec,
(4)                 'ntriangleright'         : 0x22eb,
(4)                 'ntrianglerighteq'       : 0x22ed,
(4)                 'nu'                     : 0x3bd,
(4)                 'nvDash'                 : 0x22ad,
(4)                 'nvdash'                 : 0x22ac,
(4)                 'nwarrow'                : 0x2196,
(4)                 'o'                      : 0xf8,
(4)                 'obar'                   : 0x233d,
(4)                 'ocirc'                  : 0x30a,
(4)                 'odot'                   : 0x2299,
(4)                 'oe'                     : 0x153,
(4)                 'oequal'                 : 0x229c,
(4)                 'oiint'                  : 0x2230,
(4)                 'oint'                   : 0x222f,
(4)                 'oint'                   : 0x222e,
(4)                 'omega'                  : 0x3c9,
(4)                 'ominus'                 : 0x2296,
(4)                 'oplus'                  : 0x2295,
(4)                 'origof'                 : 0x22b6,
(4)                 'oslash'                 : 0x2298,
(4)                 'otimes'                 : 0x2297,
(4)                 'overarc'                : 0x311,
(4)                 'overleftarrow'          : 0x20d6,
(4)                 'overleftrightarrow'     : 0x20e1,
(4)                 'parallel'               : 0x2225,
(4)                 'partial'                : 0x2202,
(4)                 'perp'                   : 0x27c2,
(4)                 'perthousand'            : 0x2030,
(4)                 'phi'                    : 0x3d5,
(4)                 'pi'                     : 0x3c0,
(4)                 'pitchfork'              : 0x22d4,
(4)                 'plus'                   : 0x2b,
(4)                 'pm'                     : 0xb1,
(4)                 'prec'                   : 0x227a,
(4)                 'precapprox'             : 0x2ab7,
(4)                 'preccurlyeq'            : 0x227c,
(4)                 'preceq'                 : 0x227c,
(4)                 'precnapprox'            : 0x2ab9,
(4)                 'precnsim'               : 0x22e8,
(4)                 'precsim'                : 0x227e,
(4)                 'prime'                  : 0x2032,
(4)                 'prod'                   : 0x220f,
(4)                 'propto'                 : 0x221d,
(4)                 'prurel'                 : 0x22b0,
(4)                 'psi'                    : 0x3c8,
(4)                 'quad'                   : 0x2003,
(4)                 'questeq'                : 0x225f,
(4)                 'rangle'                 : 0x27e9,
(4)                 'rasp'                   : 0x2bc,
(4)                 'ratio'                  : 0x2236,
(4)                 'rbrace'                 : 0x7d,
(4)                 'rbrack'                 : 0x5d,
(4)                 'rceil'                  : 0x2309,
```

```
(4)                 'rfloor'                 : 0x230b,
(4)                 'rgroup'                 : 0x27ef,
(4)                 'rhd'                    : 0x25b7,
(4)                 'rho'                    : 0x3c1,
(4)                 'rightModels'            : 0x22ab,
(4)                 'rightangle'             : 0x221f,
(4)                 'rightarrow'             : 0x2192,
(4)                 'rightarrowbar'          : 0x21e5,
(4)                 'rightarrowtail'         : 0x21a3,
(4)                 'rightassert'            : 0x22a6,
(4)                 'rightbrace'             : 0x7d,
(4)                 'rightharpoonaccent'     : 0x20d1,
(4)                 'rightharpoondown'       : 0x21c1,
(4)                 'rightharpoonup'         : 0x21c0,
(4)                 'rightleftarrows'        : 0x21c4,
(4)                 'rightleftharpoons'      : 0x21cc,
(4)                 'rightparen'             : 0x29,
(4)                 'rightrightarrows'       : 0x21c9,
(4)                 'rightsquigarrow'        : 0x219d,
(4)                 'rightthreetimes'        : 0x22cc,
(4)                 'rightzigzagarrow'       : 0x21dd,
(4)                 'ring'                   : 0x2da,
(4)                 'risingdotseq'           : 0x2253,
(4)                 'rq'                     : 0x2019,
(4)                 'rtimes'                 : 0x22ca,
(4)                 'scrB'                   : 0x212c,
(4)                 'scrE'                   : 0x2130,
(4)                 'scrF'                   : 0x2131,
(4)                 'scrH'                   : 0x210b,
(4)                 'scrI'                   : 0x2110,
(4)                 'scrL'                   : 0x2112,
(4)                 'scrM'                   : 0x2133,
(4)                 'scrR'                   : 0x211b,
(4)                 'scre'                   : 0x212f,
(4)                 'scrg'                   : 0x210a,
(4)                 'scro'                   : 0x2134,
(4)                 'scurel'                 : 0x22b1,
(4)                 'searrow'                : 0x2198,
(4)                 'setminus'               : 0x2216,
(4)                 'sharp'                  : 0x266f,
(4)                 'sigma'                  : 0x3c3,
(4)                 'sim'                    : 0x223c,
(4)                 'simeq'                  : 0x2243,
(4)                 'simneqq'                : 0x2246,
(4)                 'sinewave'               : 0x223f,
(4)                 'slash'                  : 0x2215,
(4)                 'smallin'                : 0x220a,
(4)                 'smallintclockwise'      : 0x2231,
(4)                 'smallointctrcclockwise' : 0x2233,
(4)                 'smallowns'              : 0x220d,
(4)                 'smallsetminus'          : 0x2216,
(4)                 'smallvarointclockwise'  : 0x2232,
(4)                 'smile'                  : 0x2323,
(4)                 'solbar'                 : 0x233f,
(4)                 'spadesuit'              : 0x2660,
(4)                 'spadesuitopen'          : 0x2664,
(4)                 'sphericalangle'         : 0x2222,
(4)                 'sqcap'                  : 0x2293,
(4)                 'sqcup'                  : 0x2294,
(4)                 'sqsubset'               : 0x228f,
(4)                 'sqsubseteq'             : 0x2291,
(4)                 'sqsubsetneq'            : 0x22e4,
(4)                 'sqsupset'               : 0x2290,
(4)                 'sqsupseteq'             : 0x2292,
(4)                 'sqsupsetneq'            : 0x22e5,
(4)                 'ss'                     : 0xdf,
(4)                 'star'                   : 0x22c6,
(4)                 'stareq'                 : 0x225b,
(4)                 'sterling'               : 0xa3,
```

```
986: (4)                'subset'                : 0x2282,
987: (4)                'subseteq'              : 0x2286,
988: (4)                'subseteqq'             : 0x2ac5,
989: (4)                'subsetneq'             : 0x228a,
990: (4)                'subsetneqq'            : 0x2acb,
991: (4)                'succ'                  : 0x227b,
992: (4)                'succapprox'            : 0x2ab8,
993: (4)                'succcurlyeq'           : 0x227d,
994: (4)                'succeq'                : 0x227d,
995: (4)                'succnapprox'           : 0x2aba,
996: (4)                'succnsim'              : 0x22e9,
997: (4)                'succsim'               : 0x227f,
998: (4)                'sum'                   : 0x2211,
999: (4)                'supset'                : 0x2283,
1000: (4)               'supseteq'              : 0x2287,
1001: (4)               'supseteqq'             : 0x2ac6,
1002: (4)               'supsetneq'             : 0x228b,
1003: (4)               'supsetneqq'            : 0x2acc,
1004: (4)               'swarrow'               : 0x2199,
1005: (4)               't'                     : 0x361,
1006: (4)               'tau'                   : 0x3c4,
1007: (4)               'textasciiacute'        : 0xb4,
1008: (4)               'textasciicircum'       : 0x5e,
1009: (4)               'textasciigrave'        : 0x60,
1010: (4)               'textasciitilde'        : 0x7e,
1011: (4)               'textexclamdown'        : 0xa1,
1012: (4)               'textquestiondown'      : 0xbf,
1013: (4)               'textquotedblleft'      : 0x201c,
1014: (4)               'textquotedblright'     : 0x201d,
1015: (4)               'therefore'             : 0x2234,
1016: (4)               'theta'                 : 0x3b8,
1017: (4)               'thickspace'            : 0x2005,
1018: (4)               'thorn'                 : 0xfe,
1019: (4)               'tilde'                 : 0x303,
1020: (4)               'times'                 : 0xd7,
1021: (4)               'to'                    : 0x2192,
1022: (4)               'top'                   : 0x22a4,
1023: (4)               'triangle'              : 0x25b3,
1024: (4)               'triangledown'          : 0x25bf,
1025: (4)               'triangleeq'            : 0x225c,
1026: (4)               'triangleleft'          : 0x25c1,
1027: (4)               'triangleleftеq'        : 0x22b4,
1028: (4)               'triangleq'             : 0x225c,
1029: (4)               'triangleright'         : 0x25b7,
1030: (4)               'triangleright eq'      : 0x22b5,
1031: (4)               'turnednot'             : 0x2319,
1032: (4)               'twoheaddownarrow'      : 0x21a1,
1033: (4)               'twoheadleftarrow'      : 0x219e,
1034: (4)               'twoheadrightarrow'     : 0x21a0,
1035: (4)               'twoheaduparrow'        : 0x219f,
1036: (4)               'ulcorner'              : 0x231c,
1037: (4)               'underbar'              : 0x331,
1038: (4)               'unlhd'                 : 0x22b4,
1039: (4)               'unrhd'                 : 0x22b5,
1040: (4)               'uparrow'               : 0x2191,
1041: (4)               'updownarrow'           : 0x2195,
1042: (4)               'updownarrowbar'        : 0x21a8,
1043: (4)               'updownarrows'          : 0x21c5,
1044: (4)               'upharpoonleft'         : 0x21bf,
1045: (4)               'upharpoonright'        : 0x21be,
1046: (4)               'uplus'                 : 0x228e,
1047: (4)               'upsilon'               : 0x3c5,
1048: (4)               'upuparrows'            : 0x21c8,
1049: (4)               'urcorner'              : 0x231d,
1050: (4)               'vDash'                 : 0x22a8,
1051: (4)               'varepsilon'            : 0x3b5,
1052: (4)               'varisinobar'           : 0x22f6,
1053: (4)               'varisins'              : 0x22f3,
1054: (4)               'varkappa'              : 0x3f0,
```

```
(4)                 'varlrtriangle'          : 0x22bf,
(4)                 'varniobar'              : 0x22fd,
(4)                 'varnis'                 : 0x22fb,
(4)                 'varnothing'             : 0x2205,
(4)                 'varphi'                 : 0x3c6,
(4)                 'varpi'                  : 0x3d6,
(4)                 'varpropto'              : 0x221d,
(4)                 'varrho'                 : 0x3f1,
(4)                 'varsigma'               : 0x3c2,
(4)                 'vartheta'               : 0x3d1,
(4)                 'vartriangle'            : 0x25b5,
(4)                 'vartriangleleft'        : 0x22b2,
(4)                 'vartrianglerright'      : 0x22b3,
(4)                 'vdash'                  : 0x22a2,
(4)                 'vdots'                  : 0x22ee,
(4)                 'vec'                    : 0x20d7,
(4)                 'vee'                    : 0x2228,
(4)                 'veebar'                 : 0x22bb,
(4)                 'veeeq'                  : 0x225a,
(4)                 'vert'                   : 0x7c,
(4)                 'wedge'                  : 0x2227,
(4)                 'wedgeq'                 : 0x2259,
(4)                 'widebar'                : 0x305,
(4)                 'widehat'                : 0x302,
(4)                 'widetilde'              : 0x303,
(4)                 'wp'                     : 0x2118,
(4)                 'wr'                     : 0x2240,
(4)                 'xi'                     : 0x3be,
(4)                 'yen'                    : 0xa5,
(4)                 'zeta'                   : 0x3b6,
(4)                 '{'                      : 0x7b,
(4)                 '|'                      : 0x2016,
(4)                 '}'                      : 0x7d,
(0)             }
(0)             _EntryTypeIn = tuple[str, str, str, Union[str, int]]
(0)             _EntryTypeOut = tuple[int, int, str, int]
(0)             _stix_virtual_fonts: dict[str, Union[dict[
(4)                 str, list[_EntryTypeIn]], list[_EntryTypeIn]]] = {
(4)                 'bb': {
(8)                     "rm": [
(12)                        ("\N{DIGIT ZERO}",
(13)                         "\N{DIGIT NINE}",
(13)                         "rm",
(13)                         "\N{MATHEMATICAL DOUBLE-STRUCK DIGIT ZERO}"),
(12)                        ("\N{LATIN CAPITAL LETTER A}",
(13)                         "\N{LATIN CAPITAL LETTER B}",
(13)                         "rm",
(13)                         "\N{MATHEMATICAL DOUBLE-STRUCK CAPITAL A}"),
(12)                        ("\N{LATIN CAPITAL LETTER C}",
(13)                         "\N{LATIN CAPITAL LETTER C}",
(13)                         "rm",
(13)                         "\N{DOUBLE-STRUCK CAPITAL C}"),
(12)                        ("\N{LATIN CAPITAL LETTER D}",
(13)                         "\N{LATIN CAPITAL LETTER G}",
(13)                         "rm",
(13)                         "\N{MATHEMATICAL DOUBLE-STRUCK CAPITAL D}"),
(12)                        ("\N{LATIN CAPITAL LETTER H}",
(13)                         "\N{LATIN CAPITAL LETTER H}",
(13)                         "rm",
(13)                         "\N{DOUBLE-STRUCK CAPITAL H}"),
(12)                        ("\N{LATIN CAPITAL LETTER I}",
(13)                         "\N{LATIN CAPITAL LETTER M}",
(13)                         "rm",
(13)                         "\N{MATHEMATICAL DOUBLE-STRUCK CAPITAL I}"),
(12)                        ("\N{LATIN CAPITAL LETTER N}",
(13)                         "\N{LATIN CAPITAL LETTER N}",
(13)                         "rm",
(13)                         "\N{DOUBLE-STRUCK CAPITAL N}"),
(12)                        ("\N{LATIN CAPITAL LETTER O}",
```

```
(13)                        "\N{LATIN CAPITAL LETTER O}",
(13)                        "rm",
(13)                        "\N{MATHEMATICAL DOUBLE-STRUCK CAPITAL O}"),
(12)                       ("\N{LATIN CAPITAL LETTER P}",
(13)                        "\N{LATIN CAPITAL LETTER Q}",
(13)                        "rm",
(13)                        "\N{DOUBLE-STRUCK CAPITAL P}"),
(12)                       ("\N{LATIN CAPITAL LETTER R}",
(13)                        "\N{LATIN CAPITAL LETTER R}",
(13)                        "rm",
(13)                        "\N{DOUBLE-STRUCK CAPITAL R}"),
(12)                       ("\N{LATIN CAPITAL LETTER S}",
(13)                        "\N{LATIN CAPITAL LETTER Y}",
(13)                        "rm",
(13)                        "\N{MATHEMATICAL DOUBLE-STRUCK CAPITAL S}"),
(12)                       ("\N{LATIN CAPITAL LETTER Z}",
(13)                        "\N{LATIN CAPITAL LETTER Z}",
(13)                        "rm",
(13)                        "\N{DOUBLE-STRUCK CAPITAL Z}"),
(12)                       ("\N{LATIN SMALL LETTER A}",
(13)                        "\N{LATIN SMALL LETTER Z}",
(13)                        "rm",
(13)                        "\N{MATHEMATICAL DOUBLE-STRUCK SMALL A}"),
(12)                       ("\N{GREEK CAPITAL LETTER GAMMA}",
(13)                        "\N{GREEK CAPITAL LETTER GAMMA}",
(13)                        "rm",
(13)                        "\N{DOUBLE-STRUCK CAPITAL GAMMA}"),
(12)                       ("\N{GREEK CAPITAL LETTER PI}",
(13)                        "\N{GREEK CAPITAL LETTER PI}",
(13)                        "rm",
(13)                        "\N{DOUBLE-STRUCK CAPITAL PI}"),
(12)                       ("\N{GREEK CAPITAL LETTER SIGMA}",
(13)                        "\N{GREEK CAPITAL LETTER SIGMA}",
(13)                        "rm",
(13)                        "\N{DOUBLE-STRUCK N-ARY SUMMATION}"),
(12)                       ("\N{GREEK SMALL LETTER GAMMA}",
(13)                        "\N{GREEK SMALL LETTER GAMMA}",
(13)                        "rm",
(13)                        "\N{DOUBLE-STRUCK SMALL GAMMA}"),
(12)                       ("\N{GREEK SMALL LETTER PI}",
(13)                        "\N{GREEK SMALL LETTER PI}",
(13)                        "rm",
(13)                        "\N{DOUBLE-STRUCK SMALL PI}"),
(8)                    ],
(8)                    "it": [
(12)                       ("\N{DIGIT ZERO}",
(13)                        "\N{DIGIT NINE}",
(13)                        "rm",
(13)                        "\N{MATHEMATICAL DOUBLE-STRUCK DIGIT ZERO}"),
(12)                       ("\N{LATIN CAPITAL LETTER A}",
(13)                        "\N{LATIN CAPITAL LETTER B}",
(13)                        "it",
(13)                        0xe154),
(12)                       ("\N{LATIN CAPITAL LETTER C}",
(13)                        "\N{LATIN CAPITAL LETTER C}",
(13)                        "it",
(13)                        "\N{DOUBLE-STRUCK CAPITAL C}"),
(12)                       ("\N{LATIN CAPITAL LETTER D}",
(13)                        "\N{LATIN CAPITAL LETTER D}",
(13)                        "it",
(13)                        "\N{DOUBLE-STRUCK ITALIC CAPITAL D}"),
(12)                       ("\N{LATIN CAPITAL LETTER E}",
(13)                        "\N{LATIN CAPITAL LETTER G}",
(13)                        "it",
(13)                        0xe156),
(12)                       ("\N{LATIN CAPITAL LETTER H}",
(13)                        "\N{LATIN CAPITAL LETTER H}",
(13)                        "it",
(13)                        "\N{DOUBLE-STRUCK CAPITAL H}"),
```

```
(12)                        ("\N{LATIN CAPITAL LETTER I}",
(13)                         "\N{LATIN CAPITAL LETTER M}",
(13)                         "it",
(13)                         0xe159),
(12)                        ("\N{LATIN CAPITAL LETTER N}",
(13)                         "\N{LATIN CAPITAL LETTER N}",
(13)                         "it",
(13)                         "\N{DOUBLE-STRUCK CAPITAL N}"),
(12)                        ("\N{LATIN CAPITAL LETTER O}",
(13)                         "\N{LATIN CAPITAL LETTER O}",
(13)                         "it",
(13)                         0xe15e),
(12)                        ("\N{LATIN CAPITAL LETTER P}",
(13)                         "\N{LATIN CAPITAL LETTER Q}",
(13)                         "it",
(13)                         "\N{DOUBLE-STRUCK CAPITAL P}"),
(12)                        ("\N{LATIN CAPITAL LETTER R}",
(13)                         "\N{LATIN CAPITAL LETTER R}",
(13)                         "it",
(13)                         "\N{DOUBLE-STRUCK CAPITAL R}"),
(12)                        ("\N{LATIN CAPITAL LETTER S}",
(13)                         "\N{LATIN CAPITAL LETTER Y}",
(13)                         "it",
(13)                         0xe15f),
(12)                        ("\N{LATIN CAPITAL LETTER Z}",
(13)                         "\N{LATIN CAPITAL LETTER Z}",
(13)                         "it",
(13)                         "\N{DOUBLE-STRUCK CAPITAL Z}"),
(12)                        ("\N{LATIN SMALL LETTER A}",
(13)                         "\N{LATIN SMALL LETTER C}",
(13)                         "it",
(13)                         0xe166),
(12)                        ("\N{LATIN SMALL LETTER D}",
(13)                         "\N{LATIN SMALL LETTER E}",
(13)                         "it",
(13)                         "\N{DOUBLE-STRUCK ITALIC SMALL D}"),
(12)                        ("\N{LATIN SMALL LETTER F}",
(13)                         "\N{LATIN SMALL LETTER H}",
(13)                         "it",
(13)                         0xe169),
(12)                        ("\N{LATIN SMALL LETTER I}",
(13)                         "\N{LATIN SMALL LETTER J}",
(13)                         "it",
(13)                         "\N{DOUBLE-STRUCK ITALIC SMALL I}"),
(12)                        ("\N{LATIN SMALL LETTER K}",
(13)                         "\N{LATIN SMALL LETTER Z}",
(13)                         "it",
(13)                         0xe16c),
(12)                        ("\N{GREEK CAPITAL LETTER GAMMA}",
(13)                         "\N{GREEK CAPITAL LETTER GAMMA}",
(13)                         "it",
(13)                         "\N{DOUBLE-STRUCK CAPITAL GAMMA}"),  # \Gamma (not in beta STIX
fonts)
(12)                        ("\N{GREEK CAPITAL LETTER PI}",
(13)                         "\N{GREEK CAPITAL LETTER PI}",
(13)                         "it",
(13)                         "\N{DOUBLE-STRUCK CAPITAL PI}"),
(12)                        ("\N{GREEK CAPITAL LETTER SIGMA}",
(13)                         "\N{GREEK CAPITAL LETTER SIGMA}",
(13)                         "it",
(13)                         "\N{DOUBLE-STRUCK N-ARY SUMMATION}"),  # \Sigma (not in beta STIX
fonts)
(12)                        ("\N{GREEK SMALL LETTER GAMMA}",
(13)                         "\N{GREEK SMALL LETTER GAMMA}",
(13)                         "it",
(13)                         "\N{DOUBLE-STRUCK SMALL GAMMA}"),  # \gamma (not in beta STIX
fonts)
(12)                        ("\N{GREEK SMALL LETTER PI}",
(13)                         "\N{GREEK SMALL LETTER PI}",
```

```
(13)                        "it",
(13)                        "\N{DOUBLE-STRUCK SMALL PI}"),
(8)                     ],
(8)                     "bf": [
(12)                        ("\N{DIGIT ZERO}",
(13)                         "\N{DIGIT NINE}",
(13)                         "rm",
(13)                         "\N{MATHEMATICAL DOUBLE-STRUCK DIGIT ZERO}"),
(12)                        ("\N{LATIN CAPITAL LETTER A}",
(13)                         "\N{LATIN CAPITAL LETTER B}",
(13)                         "bf",
(13)                         0xe38a),
(12)                        ("\N{LATIN CAPITAL LETTER C}",
(13)                         "\N{LATIN CAPITAL LETTER C}",
(13)                         "bf",
(13)                         "\N{DOUBLE-STRUCK CAPITAL C}"),
(12)                        ("\N{LATIN CAPITAL LETTER D}",
(13)                         "\N{LATIN CAPITAL LETTER D}",
(13)                         "bf",
(13)                         "\N{DOUBLE-STRUCK ITALIC CAPITAL D}"),
(12)                        ("\N{LATIN CAPITAL LETTER E}",
(13)                         "\N{LATIN CAPITAL LETTER G}",
(13)                         "bf",
(13)                         0xe38d),
(12)                        ("\N{LATIN CAPITAL LETTER H}",
(13)                         "\N{LATIN CAPITAL LETTER H}",
(13)                         "bf",
(13)                         "\N{DOUBLE-STRUCK CAPITAL H}"),
(12)                        ("\N{LATIN CAPITAL LETTER I}",
(13)                         "\N{LATIN CAPITAL LETTER M}",
(13)                         "bf",
(13)                         0xe390),
(12)                        ("\N{LATIN CAPITAL LETTER N}",
(13)                         "\N{LATIN CAPITAL LETTER N}",
(13)                         "bf",
(13)                         "\N{DOUBLE-STRUCK CAPITAL N}"),
(12)                        ("\N{LATIN CAPITAL LETTER O}",
(13)                         "\N{LATIN CAPITAL LETTER O}",
(13)                         "bf",
(13)                         0xe395),
(12)                        ("\N{LATIN CAPITAL LETTER P}",
(13)                         "\N{LATIN CAPITAL LETTER Q}",
(13)                         "bf",
(13)                         "\N{DOUBLE-STRUCK CAPITAL P}"),
(12)                        ("\N{LATIN CAPITAL LETTER R}",
(13)                         "\N{LATIN CAPITAL LETTER R}",
(13)                         "bf",
(13)                         "\N{DOUBLE-STRUCK CAPITAL R}"),
(12)                        ("\N{LATIN CAPITAL LETTER S}",
(13)                         "\N{LATIN CAPITAL LETTER Y}",
(13)                         "bf",
(13)                         0xe396),
(12)                        ("\N{LATIN CAPITAL LETTER Z}",
(13)                         "\N{LATIN CAPITAL LETTER Z}",
(13)                         "bf",
(13)                         "\N{DOUBLE-STRUCK CAPITAL Z}"),
(12)                        ("\N{LATIN SMALL LETTER A}",
(13)                         "\N{LATIN SMALL LETTER C}",
(13)                         "bf",
(13)                         0xe39d),
(12)                        ("\N{LATIN SMALL LETTER D}",
(13)                         "\N{LATIN SMALL LETTER E}",
(13)                         "bf",
(13)                         "\N{DOUBLE-STRUCK ITALIC SMALL D}"),
(12)                        ("\N{LATIN SMALL LETTER F}",
(13)                         "\N{LATIN SMALL LETTER H}",
(13)                         "bf",
(13)                         0xe3a2),
(12)                        ("\N{LATIN SMALL LETTER I}",
```

```
(13)                        "\N{LATIN SMALL LETTER J}",
(13)                        "bf",
(13)                        "\N{DOUBLE-STRUCK ITALIC SMALL I}"),
(12)                       ("\N{LATIN SMALL LETTER K}",
(13)                        "\N{LATIN SMALL LETTER Z}",
(13)                        "bf",
(13)                        0xe3a7),
(12)                       ("\N{GREEK CAPITAL LETTER GAMMA}",
(13)                        "\N{GREEK CAPITAL LETTER GAMMA}",
(13)                        "bf",
(13)                        "\N{DOUBLE-STRUCK CAPITAL GAMMA}"),
(12)                       ("\N{GREEK CAPITAL LETTER PI}",
(13)                        "\N{GREEK CAPITAL LETTER PI}",
(13)                        "bf",
(13)                        "\N{DOUBLE-STRUCK CAPITAL PI}"),
(12)                       ("\N{GREEK CAPITAL LETTER SIGMA}",
(13)                        "\N{GREEK CAPITAL LETTER SIGMA}",
(13)                        "bf",
(13)                        "\N{DOUBLE-STRUCK N-ARY SUMMATION}"),
(12)                       ("\N{GREEK SMALL LETTER GAMMA}",
(13)                        "\N{GREEK SMALL LETTER GAMMA}",
(13)                        "bf",
(13)                        "\N{DOUBLE-STRUCK SMALL GAMMA}"),
(12)                       ("\N{GREEK SMALL LETTER PI}",
(13)                        "\N{GREEK SMALL LETTER PI}",
(13)                        "bf",
(13)                        "\N{DOUBLE-STRUCK SMALL PI}"),
(8)                     ],
(4)                 },
(4)                 'cal': [
(8)                     ("\N{LATIN CAPITAL LETTER A}",
(9)                      "\N{LATIN CAPITAL LETTER Z}",
(9)                      "it",
(9)                      0xe22d),
(4)                 ],
(4)                 'frak': {
(8)                     "rm": [
(12)                       ("\N{LATIN CAPITAL LETTER A}",
(13)                        "\N{LATIN CAPITAL LETTER B}",
(13)                        "rm",
(13)                        "\N{MATHEMATICAL FRAKTUR CAPITAL A}"),
(12)                       ("\N{LATIN CAPITAL LETTER C}",
(13)                        "\N{LATIN CAPITAL LETTER C}",
(13)                        "rm",
(13)                        "\N{BLACK-LETTER CAPITAL C}"),
(12)                       ("\N{LATIN CAPITAL LETTER D}",
(13)                        "\N{LATIN CAPITAL LETTER G}",
(13)                        "rm",
(13)                        "\N{MATHEMATICAL FRAKTUR CAPITAL D}"),
(12)                       ("\N{LATIN CAPITAL LETTER H}",
(13)                        "\N{LATIN CAPITAL LETTER H}",
(13)                        "rm",
(13)                        "\N{BLACK-LETTER CAPITAL H}"),
(12)                       ("\N{LATIN CAPITAL LETTER I}",
(13)                        "\N{LATIN CAPITAL LETTER I}",
(13)                        "rm",
(13)                        "\N{BLACK-LETTER CAPITAL I}"),
(12)                       ("\N{LATIN CAPITAL LETTER J}",
(13)                        "\N{LATIN CAPITAL LETTER Q}",
(13)                        "rm",
(13)                        "\N{MATHEMATICAL FRAKTUR CAPITAL J}"),
(12)                       ("\N{LATIN CAPITAL LETTER R}",
(13)                        "\N{LATIN CAPITAL LETTER R}",
(13)                        "rm",
(13)                        "\N{BLACK-LETTER CAPITAL R}"),
(12)                       ("\N{LATIN CAPITAL LETTER S}",
(13)                        "\N{LATIN CAPITAL LETTER Y}",
(13)                        "rm",
(13)                        "\N{MATHEMATICAL FRAKTUR CAPITAL S}"),
```

```
(12)                        ("\N{LATIN CAPITAL LETTER Z}",
(13)                         "\N{LATIN CAPITAL LETTER Z}",
(13)                         "rm",
(13)                         "\N{BLACK-LETTER CAPITAL Z}"),
(12)                        ("\N{LATIN SMALL LETTER A}",
(13)                         "\N{LATIN SMALL LETTER Z}",
(13)                         "rm",
(13)                         "\N{MATHEMATICAL FRAKTUR SMALL A}"),
(12)                        ],
(8)                     "bf": [
(12)                        ("\N{LATIN CAPITAL LETTER A}",
(13)                         "\N{LATIN CAPITAL LETTER Z}",
(13)                         "bf",
(13)                         "\N{MATHEMATICAL BOLD FRAKTUR CAPITAL A}"),
(12)                        ("\N{LATIN SMALL LETTER A}",
(13)                         "\N{LATIN SMALL LETTER Z}",
(13)                         "bf",
(13)                         "\N{MATHEMATICAL BOLD FRAKTUR SMALL A}"),
(8)                     ],
(4)                 },
(4)                 'scr': [
(8)                     ("\N{LATIN CAPITAL LETTER A}",
(9)                      "\N{LATIN CAPITAL LETTER A}",
(9)                      "it",
(9)                      "\N{MATHEMATICAL SCRIPT CAPITAL A}"),
(8)                     ("\N{LATIN CAPITAL LETTER B}",
(9)                      "\N{LATIN CAPITAL LETTER B}",
(9)                      "it",
(9)                      "\N{SCRIPT CAPITAL B}"),
(8)                     ("\N{LATIN CAPITAL LETTER C}",
(9)                      "\N{LATIN CAPITAL LETTER D}",
(9)                      "it",
(9)                      "\N{MATHEMATICAL SCRIPT CAPITAL C}"),
(8)                     ("\N{LATIN CAPITAL LETTER E}",
(9)                      "\N{LATIN CAPITAL LETTER F}",
(9)                      "it",
(9)                      "\N{SCRIPT CAPITAL E}"),
(8)                     ("\N{LATIN CAPITAL LETTER G}",
(9)                      "\N{LATIN CAPITAL LETTER G}",
(9)                      "it",
(9)                      "\N{MATHEMATICAL SCRIPT CAPITAL G}"),
(8)                     ("\N{LATIN CAPITAL LETTER H}",
(9)                      "\N{LATIN CAPITAL LETTER H}",
(9)                      "it",
(9)                      "\N{SCRIPT CAPITAL H}"),
(8)                     ("\N{LATIN CAPITAL LETTER I}",
(9)                      "\N{LATIN CAPITAL LETTER I}",
(9)                      "it",
(9)                      "\N{SCRIPT CAPITAL I}"),
(8)                     ("\N{LATIN CAPITAL LETTER J}",
(9)                      "\N{LATIN CAPITAL LETTER K}",
(9)                      "it",
(9)                      "\N{MATHEMATICAL SCRIPT CAPITAL J}"),
(8)                     ("\N{LATIN CAPITAL LETTER L}",
(9)                      "\N{LATIN CAPITAL LETTER L}",
(9)                      "it",
(9)                      "\N{SCRIPT CAPITAL L}"),
(8)                     ("\N{LATIN CAPITAL LETTER M}",
(9)                      "\N{LATIN CAPITAL LETTER M}",
(9)                      "it",
(9)                      "\N{SCRIPT CAPITAL M}"),
(8)                     ("\N{LATIN CAPITAL LETTER N}",
(9)                      "\N{LATIN CAPITAL LETTER Q}",
(9)                      "it",
(9)                      "\N{MATHEMATICAL SCRIPT CAPITAL N}"),
(8)                     ("\N{LATIN CAPITAL LETTER R}",
(9)                      "\N{LATIN CAPITAL LETTER R}",
(9)                      "it",
(9)                      "\N{SCRIPT CAPITAL R}"),
```

```
(8)                 ("\N{LATIN CAPITAL LETTER S}",
(9)                  "\N{LATIN CAPITAL LETTER Z}",
(9)                  "it",
(9)                  "\N{MATHEMATICAL SCRIPT CAPITAL S}"),
(8)                 ("\N{LATIN SMALL LETTER A}",
(9)                  "\N{LATIN SMALL LETTER D}",
(9)                  "it",
(9)                  "\N{MATHEMATICAL SCRIPT SMALL A}"),
(8)                 ("\N{LATIN SMALL LETTER E}",
(9)                  "\N{LATIN SMALL LETTER E}",
(9)                  "it",
(9)                  "\N{SCRIPT SMALL E}"),
(8)                 ("\N{LATIN SMALL LETTER F}",
(9)                  "\N{LATIN SMALL LETTER F}",
(9)                  "it",
(9)                  "\N{MATHEMATICAL SCRIPT SMALL F}"),
(8)                 ("\N{LATIN SMALL LETTER G}",
(9)                  "\N{LATIN SMALL LETTER G}",
(9)                  "it",
(9)                  "\N{SCRIPT SMALL G}"),
(8)                 ("\N{LATIN SMALL LETTER H}",
(9)                  "\N{LATIN SMALL LETTER N}",
(9)                  "it",
(9)                  "\N{MATHEMATICAL SCRIPT SMALL H}"),
(8)                 ("\N{LATIN SMALL LETTER O}",
(9)                  "\N{LATIN SMALL LETTER O}",
(9)                  "it",
(9)                  "\N{SCRIPT SMALL O}"),
(8)                 ("\N{LATIN SMALL LETTER P}",
(9)                  "\N{LATIN SMALL LETTER Z}",
(9)                  "it",
(9)                  "\N{MATHEMATICAL SCRIPT SMALL P}"),
(4)             ],
(4)             'sf': {
(8)                 "rm": [
(12)                    ("\N{DIGIT ZERO}",
(13)                     "\N{DIGIT NINE}",
(13)                     "rm",
(13)                     "\N{MATHEMATICAL SANS-SERIF DIGIT ZERO}"),
(12)                    ("\N{LATIN CAPITAL LETTER A}",
(13)                     "\N{LATIN CAPITAL LETTER Z}",
(13)                     "rm",
(13)                     "\N{MATHEMATICAL SANS-SERIF CAPITAL A}"),
(12)                    ("\N{LATIN SMALL LETTER A}",
(13)                     "\N{LATIN SMALL LETTER Z}",
(13)                     "rm",
(13)                     "\N{MATHEMATICAL SANS-SERIF SMALL A}"),
(12)                    ("\N{GREEK CAPITAL LETTER ALPHA}",
(13)                     "\N{GREEK CAPITAL LETTER OMEGA}",
(13)                     "rm",
(13)                     0xe17d),
(12)                    ("\N{GREEK SMALL LETTER ALPHA}",
(13)                     "\N{GREEK SMALL LETTER OMEGA}",
(13)                     "rm",
(13)                     0xe196),
(12)                    ("\N{GREEK THETA SYMBOL}",
(13)                     "\N{GREEK THETA SYMBOL}",
(13)                     "rm",
(13)                     0xe1b0),
(12)                    ("\N{GREEK PHI SYMBOL}",
(13)                     "\N{GREEK PHI SYMBOL}",
(13)                     "rm",
(13)                     0xe1b1),
(12)                    ("\N{GREEK PI SYMBOL}",
(13)                     "\N{GREEK PI SYMBOL}",
(13)                     "rm",
(13)                     0xe1b3),
(12)                    ("\N{GREEK RHO SYMBOL}",
(13)                     "\N{GREEK RHO SYMBOL}",
```

```
(13)                        "rm",
(13)                        0xe1b2),
(12)                    ("\N{GREEK LUNATE EPSILON SYMBOL}",
(13)                        "\N{GREEK LUNATE EPSILON SYMBOL}",
(13)                        "rm",
(13)                        0xe1af),
(12)                    ("\N{PARTIAL DIFFERENTIAL}",
(13)                        "\N{PARTIAL DIFFERENTIAL}",
(13)                        "rm",
(13)                        0xe17c),
(8)                 ],
(8)                 "it": [
(12)                    ("\N{DIGIT ZERO}",
(13)                        "\N{DIGIT NINE}",
(13)                        "rm",
(13)                        "\N{MATHEMATICAL SANS-SERIF DIGIT ZERO}"),
(12)                    ("\N{LATIN CAPITAL LETTER A}",
(13)                        "\N{LATIN CAPITAL LETTER Z}",
(13)                        "it",
(13)                        "\N{MATHEMATICAL SANS-SERIF ITALIC CAPITAL A}"),
(12)                    ("\N{LATIN SMALL LETTER A}",
(13)                        "\N{LATIN SMALL LETTER Z}",
(13)                        "it",
(13)                        "\N{MATHEMATICAL SANS-SERIF ITALIC SMALL A}"),
(12)                    ("\N{GREEK CAPITAL LETTER ALPHA}",
(13)                        "\N{GREEK CAPITAL LETTER OMEGA}",
(13)                        "rm",
(13)                        0xe17d),
(12)                    ("\N{GREEK SMALL LETTER ALPHA}",
(13)                        "\N{GREEK SMALL LETTER OMEGA}",
(13)                        "it",
(13)                        0xe1d8),
(12)                    ("\N{GREEK THETA SYMBOL}",
(13)                        "\N{GREEK THETA SYMBOL}",
(13)                        "it",
(13)                        0xe1f2),
(12)                    ("\N{GREEK PHI SYMBOL}",
(13)                        "\N{GREEK PHI SYMBOL}",
(13)                        "it",
(13)                        0xe1f3),
(12)                    ("\N{GREEK PI SYMBOL}",
(13)                        "\N{GREEK PI SYMBOL}",
(13)                        "it",
(13)                        0xe1f5),
(12)                    ("\N{GREEK RHO SYMBOL}",
(13)                        "\N{GREEK RHO SYMBOL}",
(13)                        "it",
(13)                        0xe1f4),
(12)                    ("\N{GREEK LUNATE EPSILON SYMBOL}",
(13)                        "\N{GREEK LUNATE EPSILON SYMBOL}",
(13)                        "it",
(13)                        0xe1f1),
(8)                 ],
(8)                 "bf": [
(12)                    ("\N{DIGIT ZERO}",
(13)                        "\N{DIGIT NINE}",
(13)                        "bf",
(13)                        "\N{MATHEMATICAL SANS-SERIF BOLD DIGIT ZERO}"),
(12)                    ("\N{LATIN CAPITAL LETTER A}",
(13)                        "\N{LATIN CAPITAL LETTER Z}",
(13)                        "bf",
(13)                        "\N{MATHEMATICAL SANS-SERIF BOLD CAPITAL A}"),
(12)                    ("\N{LATIN SMALL LETTER A}",
(13)                        "\N{LATIN SMALL LETTER Z}",
(13)                        "bf",
(13)                        "\N{MATHEMATICAL SANS-SERIF BOLD SMALL A}"),
(12)                    ("\N{GREEK CAPITAL LETTER ALPHA}",
(13)                        "\N{GREEK CAPITAL LETTER OMEGA}",
(13)                        "bf",
```

```
(13)                        "\N{MATHEMATICAL SANS-SERIF BOLD CAPITAL ALPHA}"),
(12)                       ("\N{GREEK SMALL LETTER ALPHA}",
(13)                        "\N{GREEK SMALL LETTER OMEGA}",
(13)                        "bf",
(13)                        "\N{MATHEMATICAL SANS-SERIF BOLD SMALL ALPHA}"),
(12)                       ("\N{GREEK THETA SYMBOL}",
(13)                        "\N{GREEK THETA SYMBOL}",
(13)                        "bf",
(13)                        "\N{MATHEMATICAL SANS-SERIF BOLD THETA SYMBOL}"),
(12)                       ("\N{GREEK PHI SYMBOL}",
(13)                        "\N{GREEK PHI SYMBOL}",
(13)                        "bf",
(13)                        "\N{MATHEMATICAL SANS-SERIF BOLD PHI SYMBOL}"),
(12)                       ("\N{GREEK PI SYMBOL}",
(13)                        "\N{GREEK PI SYMBOL}",
(13)                        "bf",
(13)                        "\N{MATHEMATICAL SANS-SERIF BOLD PI SYMBOL}"),
(12)                       ("\N{GREEK KAPPA SYMBOL}",
(13)                        "\N{GREEK KAPPA SYMBOL}",
(13)                        "bf",
(13)                        "\N{MATHEMATICAL SANS-SERIF BOLD KAPPA SYMBOL}"),
(12)                       ("\N{GREEK RHO SYMBOL}",
(13)                        "\N{GREEK RHO SYMBOL}",
(13)                        "bf",
(13)                        "\N{MATHEMATICAL SANS-SERIF BOLD RHO SYMBOL}"),
(12)                       ("\N{GREEK LUNATE EPSILON SYMBOL}",
(13)                        "\N{GREEK LUNATE EPSILON SYMBOL}",
(13)                        "bf",
(13)                        "\N{MATHEMATICAL SANS-SERIF BOLD EPSILON SYMBOL}"),
(12)                       ("\N{PARTIAL DIFFERENTIAL}",
(13)                        "\N{PARTIAL DIFFERENTIAL}",
(13)                        "bf",
(13)                        "\N{MATHEMATICAL SANS-SERIF BOLD PARTIAL DIFFERENTIAL}"),
(12)                       ("\N{NABLA}",
(13)                        "\N{NABLA}",
(13)                        "bf",
(13)                        "\N{MATHEMATICAL SANS-SERIF BOLD NABLA}"),
(8)                     ],
(8)                     "bfit": [
(12)                       ("\N{LATIN CAPITAL LETTER A}",
(13)                        "\N{LATIN CAPITAL LETTER Z}",
(13)                        "bfit",
(13)                        "\N{MATHEMATICAL BOLD ITALIC CAPITAL A}"),
(12)                       ("\N{LATIN SMALL LETTER A}",
(13)                        "\N{LATIN SMALL LETTER Z}",
(13)                        "bfit",
(13)                        "\N{MATHEMATICAL BOLD ITALIC SMALL A}"),
(12)                       ("\N{GREEK CAPITAL LETTER GAMMA}",
(13)                        "\N{GREEK CAPITAL LETTER OMEGA}",
(13)                        "bfit",
(13)                        "\N{MATHEMATICAL BOLD ITALIC CAPITAL GAMMA}"),
(12)                       ("\N{GREEK SMALL LETTER ALPHA}",
(13)                        "\N{GREEK SMALL LETTER OMEGA}",
(13)                        "bfit",
(13)                        "\N{MATHEMATICAL BOLD ITALIC SMALL ALPHA}"),
(8)                     ],
(4)                 },
(4)                 'tt': [
(8)                     ("\N{DIGIT ZERO}",
(9)                      "\N{DIGIT NINE}",
(9)                      "rm",
(9)                      "\N{MATHEMATICAL MONOSPACE DIGIT ZERO}"),
(8)                     ("\N{LATIN CAPITAL LETTER A}",
(9)                      "\N{LATIN CAPITAL LETTER Z}",
(9)                      "rm",
(9)                      "\N{MATHEMATICAL MONOSPACE CAPITAL A}"),
(8)                     ("\N{LATIN SMALL LETTER A}",
(9)                      "\N{LATIN SMALL LETTER Z}",
(9)                      "rm",
```

```
1673: (9)                    "\N{MATHEMATICAL MONOSPACE SMALL A}")
1674: (4)                ],
1675: (0)            }
1676: (0)            @overload
1677: (0)            def _normalize_stix_fontcodes(d: _EntryTypeIn) -> _EntryTypeOut: ...
1678: (0)            @overload
1679: (0)            def _normalize_stix_fontcodes(d: list[_EntryTypeIn]) -> list[_EntryTypeOut]:
...
1680: (0)            @overload
1681: (0)            def _normalize_stix_fontcodes(d: dict[str, list[_EntryTypeIn] |
1682: (38)                                                 dict[str, list[_EntryTypeIn]]]
1683: (30)                                        ) -> dict[str, list[_EntryTypeOut] |
1684: (40)                                                  dict[str, list[_EntryTypeOut]]]: ...
1685: (0)            def _normalize_stix_fontcodes(d):
1686: (4)                if isinstance(d, tuple):
1687: (8)                    return tuple(ord(x) if isinstance(x, str) and len(x) == 1 else x for x
in d)
1688: (4)                elif isinstance(d, list):
1689: (8)                    return [_normalize_stix_fontcodes(x) for x in d]
1690: (4)                elif isinstance(d, dict):
1691: (8)                    return {k: _normalize_stix_fontcodes(v) for k, v in d.items()}
1692: (0)            stix_virtual_fonts: dict[str, Union[dict[str, list[_EntryTypeOut]],
1693: (36)                                                list[_EntryTypeOut]]]
1694: (0)            stix_virtual_fonts = _normalize_stix_fontcodes(_stix_virtual_fonts)
1695: (0)            del _stix_virtual_fonts
1696: (0)            stix_glyph_fixes = {
1697: (4)                0x22d2: 0x22d3,
1698: (4)                0x22d3: 0x22d2,
1699: (0)            }
```

File 16 - _pylab_helpers.py:

```
1: (0)               """
2: (0)               Manage figures for the pyplot interface.
3: (0)               """
4: (0)               import atexit
5: (0)               from collections import OrderedDict
6: (0)               class Gcf:
7: (4)                   """
8: (4)                   Singleton to maintain the relation between figures and their managers, and
9: (4)                   keep track of and "active" figure and manager.
10: (4)                  The canvas of a figure created through pyplot is associated with a figure
11: (4)                  manager, which handles the interaction between the figure and the backend.
12: (4)                  pyplot keeps track of figure managers using an identifier, the "figure
13: (4)                  number" or "manager number" (which can actually be any hashable value);
14: (4)                  this number is available as the :attr:`number` attribute of the manager.
15: (4)                  This class is never instantiated; it consists of an `OrderedDict` mapping
16: (4)                  figure/manager numbers to managers, and a set of class methods that
17: (4)                  manipulate this `OrderedDict`.
18: (4)                  Attributes
19: (4)                  ----------
20: (4)                  figs : OrderedDict
21: (8)                      `OrderedDict` mapping numbers to managers; the active manager is at
the
22: (8)                      end.
23: (4)                  """
24: (4)                  figs = OrderedDict()
25: (4)                  @classmethod
26: (4)                  def get_fig_manager(cls, num):
27: (8)                      """
28: (8)                      If manager number *num* exists, make it the active one and return it;
29: (8)                      otherwise return *None*.
30: (8)                      """
31: (8)                      manager = cls.figs.get(num, None)
32: (8)                      if manager is not None:
33: (12)                         cls.set_active(manager)
34: (8)                      return manager
```

```
(4)     @classmethod
(4)     def destroy(cls, num):
(8)         """
(8)         Destroy manager *num* -- either a manager instance or a manager
number.
(8)         In the interactive backends, this is bound to the window "destroy" and
(8)         "delete" events.
(8)         It is recommended to pass a manager instance, to avoid confusion when
(8)         two managers share the same number.
(8)         """
(8)         if all(hasattr(num, attr) for attr in ["num", "destroy"]):
(12)            manager = num
(12)            if cls.figs.get(manager.num) is manager:
(16)                cls.figs.pop(manager.num)
(8)         else:
(12)            try:
(16)                manager = cls.figs.pop(num)
(12)            except KeyError:
(16)                return
(8)         if hasattr(manager, "_cidgcf"):
(12)            manager.canvas.mpl_disconnect(manager._cidgcf)
(8)         manager.destroy()
(4)     @classmethod
(4)     def destroy_fig(cls, fig):
(8)         """Destroy figure *fig*."""
(8)         num = next((manager.num for manager in cls.figs.values()
(20)                    if manager.canvas.figure == fig), None)
(8)         if num is not None:
(12)            cls.destroy(num)
(4)     @classmethod
(4)     def destroy_all(cls):
(8)         """Destroy all figures."""
(8)         for manager in list(cls.figs.values()):
(12)            manager.canvas.mpl_disconnect(manager._cidgcf)
(12)            manager.destroy()
(8)         cls.figs.clear()
(4)     @classmethod
(4)     def has_fignum(cls, num):
(8)         """Return whether figure number *num* exists."""
(8)         return num in cls.figs
(4)     @classmethod
(4)     def get_all_fig_managers(cls):
(8)         """Return a list of figure managers."""
(8)         return list(cls.figs.values())
(4)     @classmethod
(4)     def get_num_fig_managers(cls):
(8)         """Return the number of figures being managed."""
(8)         return len(cls.figs)
(4)     @classmethod
(4)     def get_active(cls):
(8)         """Return the active manager, or *None* if there is no manager."""
(8)         return next(reversed(cls.figs.values())) if cls.figs else None
(4)     @classmethod
(4)     def _set_new_active_manager(cls, manager):
(8)         """Adopt *manager* into pyplot and make it the active manager."""
(8)         if not hasattr(manager, "_cidgcf"):
(12)            manager._cidgcf = manager.canvas.mpl_connect(
(16)                "button_press_event", lambda event: cls.set_active(manager))
(8)         fig = manager.canvas.figure
(8)         fig.number = manager.num
(8)         label = fig.get_label()
(8)         if label:
(12)            manager.set_window_title(label)
(8)         cls.set_active(manager)
(4)     @classmethod
(4)     def set_active(cls, manager):
(8)         """Make *manager* the active manager."""
(8)         cls.figs[manager.num] = manager
(8)         cls.figs.move_to_end(manager.num)
```

```
103: (4)                @classmethod
104: (4)                def draw_all(cls, force=False):
105: (8)                    """
106: (8)                    Redraw all stale managed figures, or, if *force* is True, all managed
107: (8)                    figures.
108: (8)                    """
109: (8)                    for manager in cls.get_all_fig_managers():
110: (12)                       if force or manager.canvas.figure.stale:
111: (16)                           manager.canvas.draw_idle()
112: (0)            atexit.register(Gcf.destroy_all)
```

File 17 - _text_helpers.py:

```
1: (0)              """
2: (0)              Low-level text helper utilities.
3: (0)              """
4: (0)              from __future__ import annotations
5: (0)              import dataclasses
6: (0)              from . import _api
7: (0)              from .ft2font import KERNING_DEFAULT, LOAD_NO_HINTING, FT2Font
8: (0)              @dataclasses.dataclass(frozen=True)
9: (0)              class LayoutItem:
10: (4)                 ft_object: FT2Font
11: (4)                 char: str
12: (4)                 glyph_idx: int
13: (4)                 x: float
14: (4)                 prev_kern: float
15: (0)             def warn_on_missing_glyph(codepoint, fontnames):
16: (4)                 _api.warn_external(
17: (8)                     f"Glyph {codepoint} "
18: (8)                     f"({chr(codepoint).encode('ascii', 'namereplace').decode('ascii')}) "
19: (8)                     f"missing from font(s) {fontnames}.")
20: (4)                 block = ("Hebrew" if 0x0590 <= codepoint <= 0x05ff else
21: (13)                         "Arabic" if 0x0600 <= codepoint <= 0x06ff else
22: (13)                         "Devanagari" if 0x0900 <= codepoint <= 0x097f else
23: (13)                         "Bengali" if 0x0980 <= codepoint <= 0x09ff else
24: (13)                         "Gurmukhi" if 0x0a00 <= codepoint <= 0x0a7f else
25: (13)                         "Gujarati" if 0x0a80 <= codepoint <= 0x0aff else
26: (13)                         "Oriya" if 0x0b00 <= codepoint <= 0x0b7f else
27: (13)                         "Tamil" if 0x0b80 <= codepoint <= 0x0bff else
28: (13)                         "Telugu" if 0x0c00 <= codepoint <= 0x0c7f else
29: (13)                         "Kannada" if 0x0c80 <= codepoint <= 0x0cff else
30: (13)                         "Malayalam" if 0x0d00 <= codepoint <= 0x0d7f else
31: (13)                         "Sinhala" if 0x0d80 <= codepoint <= 0x0dff else
32: (13)                         None)
33: (4)                 if block:
34: (8)                     _api.warn_external(
35: (12)                        f"Matplotlib currently does not support {block} natively.")
36: (0)             def layout(string, font, *, kern_mode=KERNING_DEFAULT):
37: (4)                 """
38: (4)                 Render *string* with *font*.
39: (4)                 For each character in *string*, yield a LayoutItem instance. When such an
instance
40: (4)                 is yielded, the font's glyph is set to the corresponding character.
41: (4)                 Parameters
42: (4)                 ----------
43: (4)                 string : str
44: (8)                     The string to be rendered.
45: (4)                 font : FT2Font
46: (8)                     The font.
47: (4)                 kern_mode : int
48: (8)                     A FreeType kerning mode.
49: (4)                 Yields
50: (4)                 ------
51: (4)                 LayoutItem
52: (4)                 """
53: (4)                 x = 0
```

```
54: (4)                 prev_glyph_idx = None
55: (4)                 char_to_font = font._get_fontmap(string)
56: (4)                 base_font = font
57: (4)                 for char in string:
58: (8)                     font = char_to_font.get(char, base_font)
59: (8)                     glyph_idx = font.get_char_index(ord(char))
60: (8)                     kern = (
61: (12)                        base_font.get_kerning(prev_glyph_idx, glyph_idx, kern_mode) / 64
62: (12)                        if prev_glyph_idx is not None else 0.
63: (8)                     )
64: (8)                     x += kern
65: (8)                     glyph = font.load_glyph(glyph_idx, flags=LOAD_NO_HINTING)
66: (8)                     yield LayoutItem(font, char, glyph_idx, x, kern)
67: (8)                     x += glyph.linearHoriAdvance / 65536
68: (8)                     prev_glyph_idx = glyph_idx
```

File 18 - _tight_bbox.py:

```
1: (0)              """
2: (0)              Helper module for the *bbox_inches* parameter in `.Figure.savefig`.
3: (0)              """
4: (0)              from matplotlib.transforms import Bbox, TransformedBbox, Affine2D
5: (0)              def adjust_bbox(fig, bbox_inches, fixed_dpi=None):
6: (4)                  """
7: (4)                  Temporarily adjust the figure so that only the specified area
8: (4)                  (bbox_inches) is saved.
9: (4)                  It modifies fig.bbox, fig.bbox_inches,
10: (4)                 fig.transFigure._boxout, and fig.patch.  While the figure size
11: (4)                 changes, the scale of the original figure is conserved.  A
12: (4)                 function which restores the original values are returned.
13: (4)                 """
14: (4)                 origBbox = fig.bbox
15: (4)                 origBboxInches = fig.bbox_inches
16: (4)                 _boxout = fig.transFigure._boxout
17: (4)                 old_aspect = []
18: (4)                 locator_list = []
19: (4)                 sentinel = object()
20: (4)                 for ax in fig.axes:
21: (8)                     locator = ax.get_axes_locator()
22: (8)                     if locator is not None:
23: (12)                        ax.apply_aspect(locator(ax, None))
24: (8)                     locator_list.append(locator)
25: (8)                     current_pos = ax.get_position(original=False).frozen()
26: (8)                     ax.set_axes_locator(lambda a, r, _pos=current_pos: _pos)
27: (8)                     if 'apply_aspect' in ax.__dict__:
28: (12)                        old_aspect.append(ax.apply_aspect)
29: (8)                     else:
30: (12)                        old_aspect.append(sentinel)
31: (8)                     ax.apply_aspect = lambda pos=None: None
32: (4)                 def restore_bbox():
33: (8)                     for ax, loc, aspect in zip(fig.axes, locator_list, old_aspect):
34: (12)                        ax.set_axes_locator(loc)
35: (12)                        if aspect is sentinel:
36: (16)                            del ax.apply_aspect
37: (12)                        else:
38: (16)                            ax.apply_aspect = aspect
39: (8)                     fig.bbox = origBbox
40: (8)                     fig.bbox_inches = origBboxInches
41: (8)                     fig.transFigure._boxout = _boxout
42: (8)                     fig.transFigure.invalidate()
43: (8)                     fig.patch.set_bounds(0, 0, 1, 1)
44: (4)                 if fixed_dpi is None:
45: (8)                     fixed_dpi = fig.dpi
46: (4)                 tr = Affine2D().scale(fixed_dpi)
47: (4)                 dpi_scale = fixed_dpi / fig.dpi
48: (4)                 fig.bbox_inches = Bbox.from_bounds(0, 0, *bbox_inches.size)
49: (4)                 x0, y0 = tr.transform(bbox_inches.p0)
```

```
50: (4)                 w1, h1 = fig.bbox.size * dpi_scale
51: (4)                 fig.transFigure._boxout = Bbox.from_bounds(-x0, -y0, w1, h1)
52: (4)                 fig.transFigure.invalidate()
53: (4)                 fig.bbox = TransformedBbox(fig.bbox_inches, tr)
54: (4)                 fig.patch.set_bounds(x0 / w1, y0 / h1,
55: (25)                                     fig.bbox.width / w1, fig.bbox.height / h1)
56: (4)                 return restore_bbox
57: (0)             def process_figure_for_rasterizing(fig, bbox_inches_restore, fixed_dpi=None):
58: (4)                 """
59: (4)                 A function that needs to be called when figure dpi changes during the
60: (4)                 drawing (e.g., rasterizing).  It recovers the bbox and re-adjust it with
61: (4)                 the new dpi.
62: (4)                 """
63: (4)                 bbox_inches, restore_bbox = bbox_inches_restore
64: (4)                 restore_bbox()
65: (4)                 r = adjust_bbox(fig, bbox_inches, fixed_dpi)
66: (4)                 return bbox_inches, r
```

File 19 - _tight_layout.py:

```
1: (0)              """
2: (0)              Routines to adjust subplot params so that subplots are
3: (0)              nicely fit in the figure. In doing so, only axis labels, tick labels, Axes
4: (0)              titles and offsetboxes that are anchored to Axes are currently considered.
5: (0)              Internally, this module assumes that the margins (left margin, etc.) which are
6: (0)              differences between ``Axes.get_tightbbox`` and ``Axes.bbox`` are independent
of
7: (0)              Axes position. This may fail if ``Axes.adjustable`` is ``datalim`` as well as
8: (0)              such cases as when left or right margin are affected by xlabel.
9: (0)              """
10: (0)             import numpy as np
11: (0)             import matplotlib as mpl
12: (0)             from matplotlib import _api, artist as martist
13: (0)             from matplotlib.font_manager import FontProperties
14: (0)             from matplotlib.transforms import Bbox
15: (0)             def _auto_adjust_subplotpars(
16: (8)                     fig, renderer, shape, span_pairs, subplot_list,
17: (8)                     ax_bbox_list=None, pad=1.08, h_pad=None, w_pad=None, rect=None):
18: (4)                 """
19: (4)                 Return a dict of subplot parameters to adjust spacing between subplots
20: (4)                 or ``None`` if resulting Axes would have zero height or width.
21: (4)                 Note that this function ignores geometry information of subplot itself,
but
22: (4)                 uses what is given by the *shape* and *subplot_list* parameters.  Also,
the
23: (4)                 results could be incorrect if some subplots have ``adjustable=datalim``.
24: (4)                 Parameters
25: (4)                 ----------
26: (4)                 shape : tuple[int, int]
27: (8)                     Number of rows and columns of the grid.
28: (4)                 span_pairs : list[tuple[slice, slice]]
29: (8)                     List of rowspans and colspans occupied by each subplot.
30: (4)                 subplot_list : list of subplots
31: (8)                     List of subplots that will be used to calculate optimal
subplot_params.
32: (4)                 pad : float
33: (8)                     Padding between the figure edge and the edges of subplots, as a
34: (8)                     fraction of the font size.
35: (4)                 h_pad, w_pad : float
36: (8)                     Padding (height/width) between edges of adjacent subplots, as a
37: (8)                     fraction of the font size.  Defaults to *pad*.
38: (4)                 rect : tuple
39: (8)                     (left, bottom, right, top), default: None.
40: (4)                 """
41: (4)                 rows, cols = shape
42: (4)                 font_size_inch = (FontProperties(
43: (8)                     size=mpl.rcParams["font.size"]).get_size_in_points() / 72)
```

```
 44: (4)                 pad_inch = pad * font_size_inch
 45: (4)                 vpad_inch = h_pad * font_size_inch if h_pad is not None else pad_inch
 46: (4)                 hpad_inch = w_pad * font_size_inch if w_pad is not None else pad_inch
 47: (4)                 if len(span_pairs) != len(subplot_list) or len(subplot_list) == 0:
 48: (8)                     raise ValueError
 49: (4)                 if rect is None:
 50: (8)                     margin_left = margin_bottom = margin_right = margin_top = None
 51: (4)                 else:
 52: (8)                     margin_left, margin_bottom, _right, _top = rect
 53: (8)                     margin_right = 1 - _right if _right else None
 54: (8)                     margin_top = 1 - _top if _top else None
 55: (4)                 vspaces = np.zeros((rows + 1, cols))
 56: (4)                 hspaces = np.zeros((rows, cols + 1))
 57: (4)                 if ax_bbox_list is None:
 58: (8)                     ax_bbox_list = [
 59: (12)                        Bbox.union([ax.get_position(original=True) for ax in subplots])
 60: (12)                        for subplots in subplot_list]
 61: (4)                 for subplots, ax_bbox, (rowspan, colspan) in zip(
 62: (12)                        subplot_list, ax_bbox_list, span_pairs):
 63: (8)                     if all(not ax.get_visible() for ax in subplots):
 64: (12)                        continue
 65: (8)                     bb = []
 66: (8)                     for ax in subplots:
 67: (12)                        if ax.get_visible():
 68: (16)                            bb += [martist._get_tightbbox_for_layout_only(ax, renderer)]
 69: (8)                     tight_bbox_raw = Bbox.union(bb)
 70: (8)                     tight_bbox = fig.transFigure.inverted().transform_bbox(tight_bbox_raw)
 71: (8)                     hspaces[rowspan, colspan.start] += ax_bbox.xmin - tight_bbox.xmin  # l
 72: (8)                     hspaces[rowspan, colspan.stop] += tight_bbox.xmax - ax_bbox.xmax  # r
 73: (8)                     vspaces[rowspan.start, colspan] += tight_bbox.ymax - ax_bbox.ymax  # t
 74: (8)                     vspaces[rowspan.stop, colspan] += ax_bbox.ymin - tight_bbox.ymin  # b
 75: (4)                 fig_width_inch, fig_height_inch = fig.get_size_inches()
 76: (4)                 if not margin_left:
 77: (8)                     margin_left = max(hspaces[:, 0].max(), 0) + pad_inch/fig_width_inch
 78: (8)                     suplabel = fig._supylabel
 79: (8)                     if suplabel and suplabel.get_in_layout():
 80: (12)                        rel_width = fig.transFigure.inverted().transform_bbox(
 81: (16)                            suplabel.get_window_extent(renderer)).width
 82: (12)                        margin_left += rel_width + pad_inch/fig_width_inch
 83: (4)                 if not margin_right:
 84: (8)                     margin_right = max(hspaces[:, -1].max(), 0) + pad_inch/fig_width_inch
 85: (4)                 if not margin_top:
 86: (8)                     margin_top = max(vspaces[0, :].max(), 0) + pad_inch/fig_height_inch
 87: (8)                     if fig._suptitle and fig._suptitle.get_in_layout():
 88: (12)                        rel_height = fig.transFigure.inverted().transform_bbox(
 89: (16)                            fig._suptitle.get_window_extent(renderer)).height
 90: (12)                        margin_top += rel_height + pad_inch/fig_height_inch
 91: (4)                 if not margin_bottom:
 92: (8)                     margin_bottom = max(vspaces[-1, :].max(), 0) +
pad_inch/fig_height_inch
 93: (8)                     suplabel = fig._supxlabel
 94: (8)                     if suplabel and suplabel.get_in_layout():
 95: (12)                        rel_height = fig.transFigure.inverted().transform_bbox(
 96: (16)                            suplabel.get_window_extent(renderer)).height
 97: (12)                        margin_bottom += rel_height + pad_inch/fig_height_inch
 98: (4)                 if margin_left + margin_right >= 1:
 99: (8)                     _api.warn_external('Tight layout not applied. The left and right '
100: (27)                                       'margins cannot be made large enough to '
101: (27)                                       'accommodate all Axes decorations.')
102: (8)                     return None
103: (4)                 if margin_bottom + margin_top >= 1:
104: (8)                     _api.warn_external('Tight layout not applied. The bottom and top '
105: (27)                                       'margins cannot be made large enough to '
106: (27)                                       'accommodate all Axes decorations.')
107: (8)                     return None
108: (4)                 kwargs = dict(left=margin_left,
109: (18)                              right=1 - margin_right,
110: (18)                              bottom=margin_bottom,
111: (18)                              top=1 - margin_top)
```

```
(4)                 if cols > 1:
(8)                     hspace = hspaces[:, 1:-1].max() + hpad_inch / fig_width_inch
(8)                     h_axes = (1 - margin_right - margin_left - hspace * (cols - 1)) / cols
(8)                     if h_axes < 0:
(12)                        _api.warn_external('Tight layout not applied. tight_layout '
(31)                                           'cannot make Axes width small enough to '
(31)                                           'accommodate all Axes decorations')
(12)                        return None
(8)                     else:
(12)                        kwargs["wspace"] = hspace / h_axes
(4)                 if rows > 1:
(8)                     vspace = vspaces[1:-1, :].max() + vpad_inch / fig_height_inch
(8)                     v_axes = (1 - margin_top - margin_bottom - vspace * (rows - 1)) / rows
(8)                     if v_axes < 0:
(12)                        _api.warn_external('Tight layout not applied. tight_layout '
(31)                                           'cannot make Axes height small enough to '
(31)                                           'accommodate all Axes decorations.')
(12)                        return None
(8)                     else:
(12)                        kwargs["hspace"] = vspace / v_axes
(4)                 return kwargs
(0)             def get_subplotspec_list(axes_list, grid_spec=None):
(4)                 """
(4)                 Return a list of subplotspec from the given list of Axes.
(4)                 For an instance of Axes that does not support subplotspec, None is
inserted
(4)                 in the list.
(4)                 If grid_spec is given, None is inserted for those not from the given
(4)                 grid_spec.
(4)                 """
(4)                 subplotspec_list = []
(4)                 for ax in axes_list:
(8)                     axes_or_locator = ax.get_axes_locator()
(8)                     if axes_or_locator is None:
(12)                        axes_or_locator = ax
(8)                     if hasattr(axes_or_locator, "get_subplotspec"):
(12)                        subplotspec = axes_or_locator.get_subplotspec()
(12)                        if subplotspec is not None:
(16)                            subplotspec = subplotspec.get_topmost_subplotspec()
(16)                            gs = subplotspec.get_gridspec()
(16)                            if grid_spec is not None:
(20)                                if gs != grid_spec:
(24)                                    subplotspec = None
(16)                            elif gs.locally_modified_subplot_params():
(20)                                subplotspec = None
(8)                     else:
(12)                        subplotspec = None
(8)                     subplotspec_list.append(subplotspec)
(4)                 return subplotspec_list
(0)             def get_tight_layout_figure(fig, axes_list, subplotspec_list, renderer,
(28)                                        pad=1.08, h_pad=None, w_pad=None, rect=None):
(4)                 """
(4)                 Return subplot parameters for tight-layouted-figure with specified
padding.
(4)                 Parameters
(4)                 ----------
(4)                 fig : Figure
(4)                 axes_list : list of Axes
(4)                 subplotspec_list : list of `.SubplotSpec`
(8)                     The subplotspecs of each Axes.
(4)                 renderer : renderer
(4)                 pad : float
(8)                     Padding between the figure edge and the edges of subplots, as a
(8)                     fraction of the font size.
(4)                 h_pad, w_pad : float
(8)                     Padding (height/width) between edges of adjacent subplots.  Defaults
to
(8)                     *pad*.
(4)                 rect : tuple (left, bottom, right, top), default: None.
```

```
(8)                 rectangle in normalized figure coordinates
(8)                 that the whole subplots area (including labels) will fit into.
(8)                 Defaults to using the entire figure.
(4)             Returns
(4)             -------
(4)             subplotspec or None
(8)                 subplotspec kwargs to be passed to `.Figure.subplots_adjust` or
(8)                 None if tight_layout could not be accomplished.
(4)             """
(4)             ss_to_subplots = {ss: [] for ss in subplotspec_list}
(4)             for ax, ss in zip(axes_list, subplotspec_list):
(8)                 ss_to_subplots[ss].append(ax)
(4)             if ss_to_subplots.pop(None, None):
(8)                 _api.warn_external(
(12)                    "This figure includes Axes that are not compatible with "
(12)                    "tight_layout, so results might be incorrect.")
(4)             if not ss_to_subplots:
(8)                 return {}
(4)             subplot_list = list(ss_to_subplots.values())
(4)             ax_bbox_list = [ss.get_position(fig) for ss in ss_to_subplots]
(4)             max_nrows = max(ss.get_gridspec().nrows for ss in ss_to_subplots)
(4)             max_ncols = max(ss.get_gridspec().ncols for ss in ss_to_subplots)
(4)             span_pairs = []
(4)             for ss in ss_to_subplots:
(8)                 rows, cols = ss.get_gridspec().get_geometry()
(8)                 div_row, mod_row = divmod(max_nrows, rows)
(8)                 div_col, mod_col = divmod(max_ncols, cols)
(8)                 if mod_row != 0:
(12)                    _api.warn_external('tight_layout not applied: number of rows '
(31)                                       'in subplot specifications must be '
(31)                                       'multiples of one another.')
(12)                    return {}
(8)                 if mod_col != 0:
(12)                    _api.warn_external('tight_layout not applied: number of '
(31)                                       'columns in subplot specifications must be '
(31)                                       'multiples of one another.')
(12)                    return {}
(8)                 span_pairs.append((
(12)                    slice(ss.rowspan.start * div_row, ss.rowspan.stop * div_row),
(12)                    slice(ss.colspan.start * div_col, ss.colspan.stop * div_col)))
(4)             kwargs = _auto_adjust_subplotpars(fig, renderer,
(38)                                              shape=(max_nrows, max_ncols),
(38)                                              span_pairs=span_pairs,
(38)                                              subplot_list=subplot_list,
(38)                                              ax_bbox_list=ax_bbox_list,
(38)                                              pad=pad, h_pad=h_pad, w_pad=w_pad)
(4)             if rect is not None and kwargs is not None:
(8)                 left, bottom, right, top = rect
(8)                 if left is not None:
(12)                    left += kwargs["left"]
(8)                 if bottom is not None:
(12)                    bottom += kwargs["bottom"]
(8)                 if right is not None:
(12)                    right -= (1 - kwargs["right"])
(8)                 if top is not None:
(12)                    top -= (1 - kwargs["top"])
(8)                 kwargs = _auto_adjust_subplotpars(fig, renderer,
(42)                                                  shape=(max_nrows, max_ncols),
(42)                                                  span_pairs=span_pairs,
(42)                                                  subplot_list=subplot_list,
(42)                                                  ax_bbox_list=ax_bbox_list,
(42)                                                  pad=pad, h_pad=h_pad, w_pad=w_pad,
(42)                                                  rect=(left, bottom, right, top))
(4)             return kwargs
```

File 20 - _type1font.py:

```
(0)  """
(0)  A class representing a Type 1 font.
(0)  This version reads pfa and pfb files and splits them for embedding in
(0)  pdf files. It also supports SlantFont and ExtendFont transformations,
(0)  similarly to pdfTeX and friends. There is no support yet for subsetting.
(0)  Usage::
(4)      font = Type1Font(filename)
(4)      clear_part, encrypted_part, finale = font.parts
(4)      slanted_font = font.transform({'slant': 0.167})
(4)      extended_font = font.transform({'extend': 1.2})
(0)  Sources:
(0)  * Adobe Technical Note #5040, Supporting Downloadable PostScript
(2)    Language Fonts.
(0)  * Adobe Type 1 Font Format, Adobe Systems Incorporated, third printing,
(2)    v1.1, 1993. ISBN 0-201-57044-0.
(0)  """
(0)  from __future__ import annotations
(0)  import binascii
(0)  import functools
(0)  import logging
(0)  import re
(0)  import string
(0)  import struct
(0)  import typing as T
(0)  import numpy as np
(0)  from matplotlib.cbook import _format_approx
(0)  from . import _api
(0)  _log = logging.getLogger(__name__)
(0)  class _Token:
(4)      """
(4)      A token in a PostScript stream.
(4)      Attributes
(4)      ----------
(4)      pos : int
(8)          Position, i.e. offset from the beginning of the data.
(4)      raw : str
(8)          Raw text of the token.
(4)      kind : str
(8)          Description of the token (for debugging or testing).
(4)      """
(4)      __slots__ = ('pos', 'raw')
(4)      kind = '?'
(4)      def __init__(self, pos, raw):
(8)          _log.debug('type1font._Token %s at %d: %r', self.kind, pos, raw)
(8)          self.pos = pos
(8)          self.raw = raw
(4)      def __str__(self):
(8)          return f"<{self.kind} {self.raw} @{self.pos}>"
(4)      def endpos(self):
(8)          """Position one past the end of the token"""
(8)          return self.pos + len(self.raw)
(4)      def is_keyword(self, *names):
(8)          """Is this a name token with one of the names?"""
(8)          return False
(4)      def is_slash_name(self):
(8)          """Is this a name token that starts with a slash?"""
(8)          return False
(4)      def is_delim(self):
(8)          """Is this a delimiter token?"""
(8)          return False
(4)      def is_number(self):
(8)          """Is this a number token?"""
(8)          return False
(4)      def value(self):
(8)          return self.raw
(0)  class _NameToken(_Token):
(4)      kind = 'name'
(4)      def is_slash_name(self):
(8)          return self.raw.startswith('/')
```

```
70: (4)                 def value(self):
71: (8)                     return self.raw[1:]
72: (0)             class _BooleanToken(_Token):
73: (4)                 kind = 'boolean'
74: (4)                 def value(self):
75: (8)                     return self.raw == 'true'
76: (0)             class _KeywordToken(_Token):
77: (4)                 kind = 'keyword'
78: (4)                 def is_keyword(self, *names):
79: (8)                     return self.raw in names
80: (0)             class _DelimiterToken(_Token):
81: (4)                 kind = 'delimiter'
82: (4)                 def is_delim(self):
83: (8)                     return True
84: (4)                 def opposite(self):
85: (8)                     return {'[': ']', ']': '[',
86: (16)                            '{': '}', '}': '{',
87: (16)                            '<<': '>>', '>>': '<<'
88: (16)                            }[self.raw]
89: (0)             class _WhitespaceToken(_Token):
90: (4)                 kind = 'whitespace'
91: (0)             class _StringToken(_Token):
92: (4)                 kind = 'string'
93: (4)                 _escapes_re = re.compile(r'\\([\\()nrtbf]|[0-7]{1,3})')
94: (4)                 _replacements = {'\\': '\\', '(': '(', ')': ')', 'n': '\n',
95: (21)                                 'r': '\r', 't': '\t', 'b': '\b', 'f': '\f'}
96: (4)                 _ws_re = re.compile('[\0\t\r\f\n ]')
97: (4)                 @classmethod
98: (4)                 def _escape(cls, match):
99: (8)                     group = match.group(1)
100: (8)                    try:
101: (12)                       return cls._replacements[group]
102: (8)                    except KeyError:
103: (12)                       return chr(int(group, 8))
104: (4)                @functools.lru_cache
105: (4)                def value(self):
106: (8)                    if self.raw[0] == '(':
107: (12)                       return self._escapes_re.sub(self._escape, self.raw[1:-1])
108: (8)                    else:
109: (12)                       data = self._ws_re.sub('', self.raw[1:-1])
110: (12)                       if len(data) % 2 == 1:
111: (16)                           data += '0'
112: (12)                       return binascii.unhexlify(data)
113: (0)            class _BinaryToken(_Token):
114: (4)                kind = 'binary'
115: (4)                def value(self):
116: (8)                    return self.raw[1:]
117: (0)            class _NumberToken(_Token):
118: (4)                kind = 'number'
119: (4)                def is_number(self):
120: (8)                    return True
121: (4)                def value(self):
122: (8)                    if '.' not in self.raw:
123: (12)                       return int(self.raw)
124: (8)                    else:
125: (12)                       return float(self.raw)
126: (0)            def _tokenize(data: bytes, skip_ws: bool) -> T.Generator[_Token, int, None]:
127: (4)                """
128: (4)                A generator that produces _Token instances from Type-1 font code.
129: (4)                The consumer of the generator may send an integer to the tokenizer to
130: (4)                indicate that the next token should be _BinaryToken of the given length.
131: (4)                Parameters
132: (4)                ----------
133: (4)                data : bytes
134: (8)                    The data of the font to tokenize.
135: (4)                skip_ws : bool
136: (8)                    If true, the generator will drop any _WhitespaceTokens from the
output.
137: (4)                """
```

```
(4)     text = data.decode('ascii', 'replace')
(4)     whitespace_or_comment_re = re.compile(r'[\0\t\r\f\n ]+|%[^\r\n]*')
(4)     token_re = re.compile(r'/{0,2}[^]\0\t\r\f\n ()<>{}/%[]+')
(4)     instring_re = re.compile(r'[()\\]')
(4)     hex_re = re.compile(r'^<[0-9a-fA-F\0\t\r\f\n ]*>$')
(4)     oct_re = re.compile(r'[0-7]{1,3}')
(4)     pos = 0
(4)     next_binary: int | None = None
(4)     while pos < len(text):
(8)         if next_binary is not None:
(12)            n = next_binary
(12)            next_binary = (yield _BinaryToken(pos, data[pos:pos+n]))
(12)            pos += n
(12)            continue
(8)         match = whitespace_or_comment_re.match(text, pos)
(8)         if match:
(12)            if not skip_ws:
(16)                next_binary = (yield _WhitespaceToken(pos, match.group()))
(12)            pos = match.end()
(8)         elif text[pos] == '(':
(12)            start = pos
(12)            pos += 1
(12)            depth = 1
(12)            while depth:
(16)                match = instring_re.search(text, pos)
(16)                if match is None:
(20)                    raise ValueError(
(24)                        f'Unterminated string starting at {start}')
(16)                pos = match.end()
(16)                if match.group() == '(':
(20)                    depth += 1
(16)                elif match.group() == ')':
(20)                    depth -= 1
(16)                else:  # a backslash
(20)                    char = text[pos]
(20)                    if char in r'\()nrtbf':
(24)                        pos += 1
(20)                    else:
(24)                        octal = oct_re.match(text, pos)
(24)                        if octal:
(28)                            pos = octal.end()
(24)                        else:
(28)                            pass  # non-escaping backslash
(12)            next_binary = (yield _StringToken(start, text[start:pos]))
(8)         elif text[pos:pos + 2] in ('<<', '>>'):
(12)            next_binary = (yield _DelimiterToken(pos, text[pos:pos + 2]))
(12)            pos += 2
(8)         elif text[pos] == '<':
(12)            start = pos
(12)            try:
(16)                pos = text.index('>', pos) + 1
(12)            except ValueError as e:
(16)                raise ValueError(f'Unterminated hex string starting at
{start}'
(33)                                 ) from e
(12)            if not hex_re.match(text[start:pos]):
(16)                raise ValueError(f'Malformed hex string starting at {start}')
(12)            next_binary = (yield _StringToken(pos, text[start:pos]))
(8)         else:
(12)            match = token_re.match(text, pos)
(12)            if match:
(16)                raw = match.group()
(16)                if raw.startswith('/'):
(20)                    next_binary = (yield _NameToken(pos, raw))
(16)                elif match.group() in ('true', 'false'):
(20)                    next_binary = (yield _BooleanToken(pos, raw))
(16)                else:
(20)                    try:
(24)                        float(raw)
```

```
(24)                                next_binary = (yield _NumberToken(pos, raw))
(20)                            except ValueError:
(24)                                next_binary = (yield _KeywordToken(pos, raw))
(16)                        pos = match.end()
(12)                    else:
(16)                        next_binary = (yield _DelimiterToken(pos, text[pos]))
(16)                        pos += 1
(0)         class _BalancedExpression(_Token):
(4)             pass
(0)         def _expression(initial, tokens, data):
(4)             """
(4)             Consume some number of tokens and return a balanced PostScript expression.
(4)             Parameters
(4)             ----------
(4)             initial : _Token
(8)                 The token that triggered parsing a balanced expression.
(4)             tokens : iterator of _Token
(8)                 Following tokens.
(4)             data : bytes
(8)                 Underlying data that the token positions point to.
(4)             Returns
(4)             -------
(4)             _BalancedExpression
(4)             """
(4)             delim_stack = []
(4)             token = initial
(4)             while True:
(8)                 if token.is_delim():
(12)                    if token.raw in ('[', '{'):
(16)                        delim_stack.append(token)
(12)                    elif token.raw in (']', '}'):
(16)                        if not delim_stack:
(20)                            raise RuntimeError(f"unmatched closing token {token}")
(16)                        match = delim_stack.pop()
(16)                        if match.raw != token.opposite():
(20)                            raise RuntimeError(
(24)                                f"opening token {match} closed by {token}"
(20)                            )
(16)                        if not delim_stack:
(20)                            break
(12)                    else:
(16)                        raise RuntimeError(f'unknown delimiter {token}')
(8)                 elif not delim_stack:
(12)                    break
(8)                 token = next(tokens)
(4)             return _BalancedExpression(
(8)                 initial.pos,
(8)                 data[initial.pos:token.endpos()].decode('ascii', 'replace')
(4)             )
(0)         class Type1Font:
(4)             """
(4)             A class representing a Type-1 font, for use by backends.
(4)             Attributes
(4)             ----------
(4)             parts : tuple
(8)                 A 3-tuple of the cleartext part, the encrypted part, and the finale of
(8)                 zeros.
(4)             decrypted : bytes
(8)                 The decrypted form of ``parts[1]``.
(4)             prop : dict[str, Any]
(8)                 A dictionary of font properties. Noteworthy keys include:
(8)                 - FontName: PostScript name of the font
(8)                 - Encoding: dict from numeric codes to glyph names
(8)                 - FontMatrix: bytes object encoding a matrix
(8)                 - UniqueID: optional font identifier, dropped when modifying the font
(8)                 - CharStrings: dict from glyph names to byte code
(8)                 - Subrs: array of byte code subroutines
(8)                 - OtherSubrs: bytes object encoding some PostScript code
(4)             """
```

```
(4)     __slots__ = ('parts', 'decrypted', 'prop', '_pos', '_abbr')
(4)     def __init__(self, input):
(8)         """
(8)         Initialize a Type-1 font.
(8)         Parameters
(8)         ----------
(8)         input : str or 3-tuple
(12)            Either a pfb file name, or a 3-tuple of already-decoded Type-1
(12)            font `~.Type1Font.parts`.
(8)         """
(8)         if isinstance(input, tuple) and len(input) == 3:
(12)            self.parts = input
(8)         else:
(12)            with open(input, 'rb') as file:
(16)                data = self._read(file)
(12)            self.parts = self._split(data)
(8)         self.decrypted = self._decrypt(self.parts[1], 'eexec')
(8)         self._abbr = {'RD': 'RD', 'ND': 'ND', 'NP': 'NP'}
(8)         self._parse()
(4)     def _read(self, file):
(8)         """Read the font from a file, decoding into usable parts."""
(8)         rawdata = file.read()
(8)         if not rawdata.startswith(b'\x80'):
(12)            return rawdata
(8)         data = b''
(8)         while rawdata:
(12)            if not rawdata.startswith(b'\x80'):
(16)                raise RuntimeError('Broken pfb file (expected byte 128, '
(35)                                   'got %d)' % rawdata[0])
(12)            type = rawdata[1]
(12)            if type in (1, 2):
(16)                length, = struct.unpack('<i', rawdata[2:6])
(16)                segment = rawdata[6:6 + length]
(16)                rawdata = rawdata[6 + length:]
(12)            if type == 1:       # ASCII text: include verbatim
(16)                data += segment
(12)            elif type == 2:     # binary data: encode in hexadecimal
(16)                data += binascii.hexlify(segment)
(12)            elif type == 3:     # end of file
(16)                break
(12)            else:
(16)                raise RuntimeError('Unknown segment type %d in pfb file' %
type)
(8)         return data
(4)     def _split(self, data):
(8)         """
(8)         Split the Type 1 font into its three main parts.
(8)         The three parts are: (1) the cleartext part, which ends in a
(8)         eexec operator; (2) the encrypted part; (3) the fixed part,
(8)         which contains 512 ASCII zeros possibly divided on various
(8)         lines, a cleartomark operator, and possibly something else.
(8)         """
(8)         idx = data.index(b'eexec')
(8)         idx += len(b'eexec')
(8)         while data[idx] in b' \t\r\n':
(12)            idx += 1
(8)         len1 = idx
(8)         idx = data.rindex(b'cleartomark') - 1
(8)         zeros = 512
(8)         while zeros and data[idx] in b'0' or data[idx] in b'\r\n':
(12)            if data[idx] in b'0':
(16)                zeros -= 1
(12)            idx -= 1
(8)         if zeros:
(12)            _log.info('Insufficiently many zeros in Type 1 font')
(8)         idx1 = len1 + ((idx - len1 + 2) & ~1)  # ensure an even number of
bytes
(8)         binary = binascii.unhexlify(data[len1:idx1])
(8)         return data[:len1], binary, data[idx+1:]
```

```
    @staticmethod
    def _decrypt(ciphertext, key, ndiscard=4):
        """
        Decrypt ciphertext using the Type-1 font algorithm.
        The algorithm is described in Adobe's "Adobe Type 1 Font Format".
        The key argument can be an integer, or one of the strings
        'eexec' and 'charstring', which map to the key specified for the
        corresponding part of Type-1 fonts.
        The ndiscard argument should be an integer, usually 4.
        That number of bytes is discarded from the beginning of plaintext.
        """
        key = _api.check_getitem({'eexec': 55665, 'charstring': 4330},
key=key)
        plaintext = []
        for byte in ciphertext:
            plaintext.append(byte ^ (key >> 8))
            key = ((key+byte) * 52845 + 22719) & 0xffff
        return bytes(plaintext[ndiscard:])
    @staticmethod
    def _encrypt(plaintext, key, ndiscard=4):
        """
        Encrypt plaintext using the Type-1 font algorithm.
        The algorithm is described in Adobe's "Adobe Type 1 Font Format".
        The key argument can be an integer, or one of the strings
        'eexec' and 'charstring', which map to the key specified for the
        corresponding part of Type-1 fonts.
        The ndiscard argument should be an integer, usually 4. That
        number of bytes is prepended to the plaintext before encryption.
        This function prepends NUL bytes for reproducibility, even though
        the original algorithm uses random bytes, presumably to avoid
        cryptanalysis.
        """
        key = _api.check_getitem({'eexec': 55665, 'charstring': 4330},
key=key)
        ciphertext = []
        for byte in b'\0' * ndiscard + plaintext:
            c = byte ^ (key >> 8)
            ciphertext.append(c)
            key = ((key + c) * 52845 + 22719) & 0xffff
        return bytes(ciphertext)
    def _parse(self):
        """
        Find the values of various font properties. This limited kind
        of parsing is described in Chapter 10 "Adobe Type Manager
        Compatibility" of the Type-1 spec.
        """
        prop = {'Weight': 'Regular', 'ItalicAngle': 0.0, 'isFixedPitch':
False,
                'UnderlinePosition': -100, 'UnderlineThickness': 50}
        pos = {}
        data = self.parts[0] + self.decrypted
        source = _tokenize(data, True)
        while True:
            try:
                token = next(source)
            except StopIteration:
                break
            if token.is_delim():
                _expression(token, source, data)
            if token.is_slash_name():
                key = token.value()
                keypos = token.pos
            else:
                continue
            if key in ('Subrs', 'CharStrings', 'Encoding', 'OtherSubrs'):
                prop[key], endpos = {
                    'Subrs': self._parse_subrs,
                    'CharStrings': self._parse_charstrings,
                    'Encoding': self._parse_encoding,
```

```
(20)                                'OtherSubrs': self._parse_othersubrs
(16)                            }[key](source, data)
(16)                            pos.setdefault(key, []).append((keypos, endpos))
(16)                            continue
(12)                        try:
(16)                            token = next(source)
(12)                        except StopIteration:
(16)                            break
(12)                        if isinstance(token, _KeywordToken):
(16)                            continue
(12)                        if token.is_delim():
(16)                            value = _expression(token, source, data).raw
(12)                        else:
(16)                            value = token.value()
(12)                        try:
(16)                            kw = next(
(20)                                kw for kw in source
(20)                                if not kw.is_keyword('readonly', 'noaccess',
'executeonly')
(16)                            )
(12)                        except StopIteration:
(16)                            break
(12)                        if kw.is_keyword('def', self._abbr['ND'], self._abbr['NP']):
(16)                            prop[key] = value
(16)                            pos.setdefault(key, []).append((keypos, kw.endpos()))
(12)                        if value == '{noaccess def}':
(16)                            self._abbr['ND'] = key
(12)                        elif value == '{noaccess put}':
(16)                            self._abbr['NP'] = key
(12)                        elif value == '{string currentfile exch readstring pop}':
(16)                            self._abbr['RD'] = key
(8)                     if 'FontName' not in prop:
(12)                        prop['FontName'] = (prop.get('FullName') or
(32)                                            prop.get('FamilyName') or
(32)                                            'Unknown')
(8)                     if 'FullName' not in prop:
(12)                        prop['FullName'] = prop['FontName']
(8)                     if 'FamilyName' not in prop:
(12)                        extras = ('(?i)([ -](regular|plain|italic|oblique|(semi)?bold|'
(22)                                  '(ultra)?light|extra|condensed))+$')
(12)                        prop['FamilyName'] = re.sub(extras, '', prop['FullName'])
(8)                     ndiscard = prop.get('lenIV', 4)
(8)                     cs = prop['CharStrings']
(8)                     for key, value in cs.items():
(12)                        cs[key] = self._decrypt(value, 'charstring', ndiscard)
(8)                     if 'Subrs' in prop:
(12)                        prop['Subrs'] = [
(16)                            self._decrypt(value, 'charstring', ndiscard)
(16)                            for value in prop['Subrs']
(12)                        ]
(8)                     self.prop = prop
(8)                     self._pos = pos
(4)                 def _parse_subrs(self, tokens, _data):
(8)                     count_token = next(tokens)
(8)                     if not count_token.is_number():
(12)                        raise RuntimeError(
(16)                            f"Token following /Subrs must be a number, was {count_token}"
(12)                        )
(8)                     count = count_token.value()
(8)                     array = [None] * count
(8)                     next(t for t in tokens if t.is_keyword('array'))
(8)                     for _ in range(count):
(12)                        next(t for t in tokens if t.is_keyword('dup'))
(12)                        index_token = next(tokens)
(12)                        if not index_token.is_number():
(16)                            raise RuntimeError(
(20)                                "Token following dup in Subrs definition must be a "
(20)                                f"number, was {index_token}"
(16)                            )
```

```
(12)                        nbytes_token = next(tokens)
(12)                        if not nbytes_token.is_number():
(16)                            raise RuntimeError(
(20)                                "Second token following dup in Subrs definition must "
(20)                                f"be a number, was {nbytes_token}"
(16)                            )
(12)                        token = next(tokens)
(12)                        if not token.is_keyword(self._abbr['RD']):
(16)                            raise RuntimeError(
(20)                                f"Token preceding subr must be {self._abbr['RD']}, "
(20)                                f"was {token}"
(16)                            )
(12)                        binary_token = tokens.send(1+nbytes_token.value())
(12)                        array[index_token.value()] = binary_token.value()
(8)                     return array, next(tokens).endpos()
(4)                 @staticmethod
(4)                 def _parse_charstrings(tokens, _data):
(8)                     count_token = next(tokens)
(8)                     if not count_token.is_number():
(12)                        raise RuntimeError(
(16)                            "Token following /CharStrings must be a number, "
(16)                            f"was {count_token}"
(12)                        )
(8)                     count = count_token.value()
(8)                     charstrings = {}
(8)                     next(t for t in tokens if t.is_keyword('begin'))
(8)                     while True:
(12)                        token = next(t for t in tokens
(25)                                     if t.is_keyword('end') or t.is_slash_name())
(12)                        if token.raw == 'end':
(16)                            return charstrings, token.endpos()
(12)                        glyphname = token.value()
(12)                        nbytes_token = next(tokens)
(12)                        if not nbytes_token.is_number():
(16)                            raise RuntimeError(
(20)                                f"Token following /{glyphname} in CharStrings definition "
(20)                                f"must be a number, was {nbytes_token}"
(16)                            )
(12)                        next(tokens)  # usually RD or |-
(12)                        binary_token = tokens.send(1+nbytes_token.value())
(12)                        charstrings[glyphname] = binary_token.value()
(4)                 @staticmethod
(4)                 def _parse_encoding(tokens, _data):
(8)                     encoding = {}
(8)                     while True:
(12)                        token = next(t for t in tokens
(25)                                     if t.is_keyword('StandardEncoding', 'dup', 'def'))
(12)                        if token.is_keyword('StandardEncoding'):
(16)                            return _StandardEncoding, token.endpos()
(12)                        if token.is_keyword('def'):
(16)                            return encoding, token.endpos()
(12)                        index_token = next(tokens)
(12)                        if not index_token.is_number():
(16)                            _log.warning(
(20)                                f"Parsing encoding: expected number, got {index_token}"
(16)                            )
(16)                            continue
(12)                        name_token = next(tokens)
(12)                        if not name_token.is_slash_name():
(16)                            _log.warning(
(20)                                f"Parsing encoding: expected slash-name, got {name_token}"
(16)                            )
(16)                            continue
(12)                        encoding[index_token.value()] = name_token.value()
(4)                 @staticmethod
(4)                 def _parse_othersubrs(tokens, data):
(8)                     init_pos = None
(8)                     while True:
(12)                        token = next(tokens)
```

```
(12)                        if init_pos is None:
(16)                            init_pos = token.pos
(12)                        if token.is_delim():
(16)                            _expression(token, tokens, data)
(12)                        elif token.is_keyword('def', 'ND', '|-'):
(16)                            return data[init_pos:token.endpos()], token.endpos()
(4)                 def transform(self, effects):
(8)                     """
(8)                     Return a new font that is slanted and/or extended.
(8)                     Parameters
(8)                     ----------
(8)                     effects : dict
(12)                        A dict with optional entries:
(12)                        - 'slant' : float, default: 0
(16)                            Tangent of the angle that the font is to be slanted to the
(16)                            right. Negative values slant to the left.
(12)                        - 'extend' : float, default: 1
(16)                            Scaling factor for the font width. Values less than 1 condense
(16)                            the glyphs.
(8)                     Returns
(8)                     -------
(8)                     `Type1Font`
(8)                     """
(8)                     fontname = self.prop['FontName']
(8)                     italicangle = self.prop['ItalicAngle']
(8)                     array = [
(12)                        float(x) for x in (self.prop['FontMatrix']
(31)                                           .lstrip('[').rstrip(']').split())
(8)                     ]
(8)                     oldmatrix = np.eye(3, 3)
(8)                     oldmatrix[0:3, 0] = array[::2]
(8)                     oldmatrix[0:3, 1] = array[1::2]
(8)                     modifier = np.eye(3, 3)
(8)                     if 'slant' in effects:
(12)                        slant = effects['slant']
(12)                        fontname += f'_Slant_{int(1000 * slant)}'
(12)                        italicangle = round(
(16)                            float(italicangle) - np.arctan(slant) / np.pi * 180,
(16)                            5
(12)                        )
(12)                        modifier[1, 0] = slant
(8)                     if 'extend' in effects:
(12)                        extend = effects['extend']
(12)                        fontname += f'_Extend_{int(1000 * extend)}'
(12)                        modifier[0, 0] = extend
(8)                     newmatrix = np.dot(modifier, oldmatrix)
(8)                     array[::2] = newmatrix[0:3, 0]
(8)                     array[1::2] = newmatrix[0:3, 1]
(8)                     fontmatrix = (
(12)                        f"[{' '.join(_format_approx(x, 6) for x in array)}]"
(8)                     )
(8)                     replacements = (
(12)                        [(x, f'/FontName/{fontname} def')
(13)                         for x in self._pos['FontName']]
(12)                        + [(x, f'/ItalicAngle {italicangle} def')
(15)                           for x in self._pos['ItalicAngle']]
(12)                        + [(x, f'/FontMatrix {fontmatrix} readonly def')
(15)                           for x in self._pos['FontMatrix']]
(12)                        + [(x, '') for x in self._pos.get('UniqueID', [])]
(8)                     )
(8)                     data = bytearray(self.parts[0])
(8)                     data.extend(self.decrypted)
(8)                     len0 = len(self.parts[0])
(8)                     for (pos0, pos1), value in sorted(replacements, reverse=True):
(12)                        data[pos0:pos1] = value.encode('ascii', 'replace')
(12)                        if pos0 < len(self.parts[0]):
(16)                            if pos1 >= len(self.parts[0]):
(20)                                raise RuntimeError(
(24)                                    f"text to be replaced with {value} spans "
```

```
(24)                                    "the eexec boundary"
(20)                                )
(16)                            len0 += len(value) - pos1 + pos0
(8)                     data = bytes(data)
(8)                     return Type1Font((
(12)                        data[:len0],
(12)                        self._encrypt(data[len0:], 'eexec'),
(12)                        self.parts[2]
(8)                     ))
(0)             _StandardEncoding = {
(4)                 **{ord(letter): letter for letter in string.ascii_letters},
(4)                 0: '.notdef',
(4)                 32: 'space',
(4)                 33: 'exclam',
(4)                 34: 'quotedbl',
(4)                 35: 'numbersign',
(4)                 36: 'dollar',
(4)                 37: 'percent',
(4)                 38: 'ampersand',
(4)                 39: 'quoteright',
(4)                 40: 'parenleft',
(4)                 41: 'parenright',
(4)                 42: 'asterisk',
(4)                 43: 'plus',
(4)                 44: 'comma',
(4)                 45: 'hyphen',
(4)                 46: 'period',
(4)                 47: 'slash',
(4)                 48: 'zero',
(4)                 49: 'one',
(4)                 50: 'two',
(4)                 51: 'three',
(4)                 52: 'four',
(4)                 53: 'five',
(4)                 54: 'six',
(4)                 55: 'seven',
(4)                 56: 'eight',
(4)                 57: 'nine',
(4)                 58: 'colon',
(4)                 59: 'semicolon',
(4)                 60: 'less',
(4)                 61: 'equal',
(4)                 62: 'greater',
(4)                 63: 'question',
(4)                 64: 'at',
(4)                 91: 'bracketleft',
(4)                 92: 'backslash',
(4)                 93: 'bracketright',
(4)                 94: 'asciicircum',
(4)                 95: 'underscore',
(4)                 96: 'quoteleft',
(4)                 123: 'braceleft',
(4)                 124: 'bar',
(4)                 125: 'braceright',
(4)                 126: 'asciitilde',
(4)                 161: 'exclamdown',
(4)                 162: 'cent',
(4)                 163: 'sterling',
(4)                 164: 'fraction',
(4)                 165: 'yen',
(4)                 166: 'florin',
(4)                 167: 'section',
(4)                 168: 'currency',
(4)                 169: 'quotesingle',
(4)                 170: 'quotedblleft',
(4)                 171: 'guillemotleft',
(4)                 172: 'guilsinglleft',
(4)                 173: 'guilsinglright',
(4)                 174: 'fi',
```

```
683: (4)                175: 'fl',
684: (4)                177: 'endash',
685: (4)                178: 'dagger',
686: (4)                179: 'daggerdbl',
687: (4)                180: 'periodcentered',
688: (4)                182: 'paragraph',
689: (4)                183: 'bullet',
690: (4)                184: 'quotesinglbase',
691: (4)                185: 'quotedblbase',
692: (4)                186: 'quotedblright',
693: (4)                187: 'guillemotright',
694: (4)                188: 'ellipsis',
695: (4)                189: 'perthousand',
696: (4)                191: 'questiondown',
697: (4)                193: 'grave',
698: (4)                194: 'acute',
699: (4)                195: 'circumflex',
700: (4)                196: 'tilde',
701: (4)                197: 'macron',
702: (4)                198: 'breve',
703: (4)                199: 'dotaccent',
704: (4)                200: 'dieresis',
705: (4)                202: 'ring',
706: (4)                203: 'cedilla',
707: (4)                205: 'hungarumlaut',
708: (4)                206: 'ogonek',
709: (4)                207: 'caron',
710: (4)                208: 'emdash',
711: (4)                225: 'AE',
712: (4)                227: 'ordfeminine',
713: (4)                232: 'Lslash',
714: (4)                233: 'Oslash',
715: (4)                234: 'OE',
716: (4)                235: 'ordmasculine',
717: (4)                241: 'ae',
718: (4)                245: 'dotlessi',
719: (4)                248: 'lslash',
720: (4)                249: 'oslash',
721: (4)                250: 'oe',
722: (4)                251: 'germandbls',
723: (0)            }

----------------------------------------

File 21 - animation.py:

1: (0)              import abc
2: (0)              import base64
3: (0)              import contextlib
4: (0)              from io import BytesIO, TextIOWrapper
5: (0)              import itertools
6: (0)              import logging
7: (0)              from pathlib import Path
8: (0)              import shutil
9: (0)              import subprocess
10: (0)             import sys
11: (0)             from tempfile import TemporaryDirectory
12: (0)             import uuid
13: (0)             import warnings
14: (0)             import numpy as np
15: (0)             from PIL import Image
16: (0)             import matplotlib as mpl
17: (0)             from matplotlib._animation_data import (
18: (4)                 DISPLAY_TEMPLATE, INCLUDED_FRAMES, JS_INCLUDE, STYLE_INCLUDE)
19: (0)             from matplotlib import _api, cbook
20: (0)             import matplotlib.colors as mcolors
21: (0)             _log = logging.getLogger(__name__)
22: (0)             subprocess_creation_flags = (
23: (4)                 subprocess.CREATE_NO_WINDOW if sys.platform == 'win32' else 0)
```

```
(0)  def adjusted_figsize(w, h, dpi, n):
(4)      """
(4)      Compute figure size so that pixels are a multiple of n.
(4)      Parameters
(4)      ----------
(4)      w, h : float
(8)          Size in inches.
(4)      dpi : float
(8)          The dpi.
(4)      n : int
(8)          The target multiple.
(4)      Returns
(4)      -------
(4)      wnew, hnew : float
(8)          The new figure size in inches.
(4)      """
(4)      def correct_roundoff(x, dpi, n):
(8)          if int(x*dpi) % n != 0:
(12)             if int(np.nextafter(x, np.inf)*dpi) % n == 0:
(16)                 x = np.nextafter(x, np.inf)
(12)             elif int(np.nextafter(x, -np.inf)*dpi) % n == 0:
(16)                 x = np.nextafter(x, -np.inf)
(8)          return x
(4)      wnew = int(w * dpi / n) * n / dpi
(4)      hnew = int(h * dpi / n) * n / dpi
(4)      return correct_roundoff(wnew, dpi, n), correct_roundoff(hnew, dpi, n)
(0)  class MovieWriterRegistry:
(4)      """Registry of available writer classes by human readable name."""
(4)      def __init__(self):
(8)          self._registered = dict()
(4)      def register(self, name):
(8)          """
(8)          Decorator for registering a class under a name.
(8)          Example use::
(12)             @registry.register(name)
(12)             class Foo:
(16)                 pass
(8)          """
(8)          def wrapper(writer_cls):
(12)             self._registered[name] = writer_cls
(12)             return writer_cls
(8)          return wrapper
(4)      def is_available(self, name):
(8)          """
(8)          Check if given writer is available by name.
(8)          Parameters
(8)          ----------
(8)          name : str
(8)          Returns
(8)          -------
(8)          bool
(8)          """
(8)          try:
(12)             cls = self._registered[name]
(8)          except KeyError:
(12)             return False
(8)          return cls.isAvailable()
(4)      def __iter__(self):
(8)          """Iterate over names of available writer class."""
(8)          for name in self._registered:
(12)             if self.is_available(name):
(16)                 yield name
(4)      def list(self):
(8)          """Get a list of available MovieWriters."""
(8)          return [*self]
(4)      def __getitem__(self, name):
(8)          """Get an available writer class from its name."""
(8)          if self.is_available(name):
(12)             return self._registered[name]
```

```
93: (8)                     raise RuntimeError(f"Requested MovieWriter ({name}) not available")
94: (0)             writers = MovieWriterRegistry()
95: (0)             class AbstractMovieWriter(abc.ABC):
96: (4)                 """
97: (4)                 Abstract base class for writing movies, providing a way to grab frames by
98: (4)                 calling `~AbstractMovieWriter.grab_frame`.
99: (4)                 `setup` is called to start the process and `finish` is called afterwards.
100: (4)                `saving` is provided as a context manager to facilitate this process as ::
101: (8)                    with moviewriter.saving(fig, outfile='myfile.mp4', dpi=100):
102: (12)                       moviewriter.grab_frame(**savefig_kwargs)
103: (4)                The use of the context manager ensures that `setup` and `finish` are
104: (4)                performed as necessary.
105: (4)                An instance of a concrete subclass of this class can be given as the
106: (4)                ``writer`` argument of `Animation.save()`.
107: (4)                """
108: (4)                def __init__(self, fps=5, metadata=None, codec=None, bitrate=None):
109: (8)                    self.fps = fps
110: (8)                    self.metadata = metadata if metadata is not None else {}
111: (8)                    self.codec = mpl._val_or_rc(codec, 'animation.codec')
112: (8)                    self.bitrate = mpl._val_or_rc(bitrate, 'animation.bitrate')
113: (4)                @abc.abstractmethod
114: (4)                def setup(self, fig, outfile, dpi=None):
115: (8)                    """
116: (8)                    Setup for writing the movie file.
117: (8)                    Parameters
118: (8)                    ----------
119: (8)                    fig : `~matplotlib.figure.Figure`
120: (12)                       The figure object that contains the information for frames.
121: (8)                    outfile : str
122: (12)                       The filename of the resulting movie file.
123: (8)                    dpi : float, default: ``fig.dpi``
124: (12)                       The DPI (or resolution) for the file.  This controls the size
125: (12)                       in pixels of the resulting movie file.
126: (8)                    """
127: (8)                    Path(outfile).parent.resolve(strict=True)
128: (8)                    self.outfile = outfile
129: (8)                    self.fig = fig
130: (8)                    if dpi is None:
131: (12)                       dpi = self.fig.dpi
132: (8)                    self.dpi = dpi
133: (4)                @property
134: (4)                def frame_size(self):
135: (8)                    """A tuple ``(width, height)`` in pixels of a movie frame."""
136: (8)                    w, h = self.fig.get_size_inches()
137: (8)                    return int(w * self.dpi), int(h * self.dpi)
138: (4)                @abc.abstractmethod
139: (4)                def grab_frame(self, **savefig_kwargs):
140: (8)                    """
141: (8)                    Grab the image information from the figure and save as a movie frame.
142: (8)                    All keyword arguments in *savefig_kwargs* are passed on to the
143: (8)                    `~.Figure.savefig` call that saves the figure.  However, several
144: (8)                    keyword arguments that are supported by `~.Figure.savefig` may not be
145: (8)                    passed as they are controlled by the MovieWriter:
146: (8)                    - *dpi*, *bbox_inches*:  These may not be passed because each frame of
the
147: (11)                      animation much be exactly the same size in pixels.
148: (8)                    - *format*: This is controlled by the MovieWriter.
149: (8)                    """
150: (4)                @abc.abstractmethod
151: (4)                def finish(self):
152: (8)                    """Finish any processing for writing the movie."""
153: (4)                @contextlib.contextmanager
154: (4)                def saving(self, fig, outfile, dpi, *args, **kwargs):
155: (8)                    """
156: (8)                    Context manager to facilitate writing the movie file.
157: (8)                    ``*args, **kw`` are any parameters that should be passed to `setup`.
158: (8)                    """
159: (8)                    if mpl.rcParams['savefig.bbox'] == 'tight':
160: (12)                       _log.info("Disabling savefig.bbox = 'tight', as it may cause "
```

```
161: (22)                                "frame size to vary, which is inappropriate for "
162: (22)                                "animation.")
163: (8)                    self.setup(fig, outfile, dpi, *args, **kwargs)
164: (8)                    with mpl.rc_context({'savefig.bbox': None}):
165: (12)                       try:
166: (16)                           yield self
167: (12)                       finally:
168: (16)                           self.finish()
169: (0)            class MovieWriter(AbstractMovieWriter):
170: (4)                """
171: (4)                Base class for writing movies.
172: (4)                This is a base class for MovieWriter subclasses that write a movie frame
173: (4)                data to a pipe. You cannot instantiate this class directly.
174: (4)                See examples for how to use its subclasses.
175: (4)                Attributes
176: (4)                ----------
177: (4)                frame_format : str
178: (8)                    The format used in writing frame data, defaults to 'rgba'.
179: (4)                fig : `~matplotlib.figure.Figure`
180: (8)                    The figure to capture data from.
181: (8)                    This must be provided by the subclasses.
182: (4)                """
183: (4)                supported_formats = ["rgba"]
184: (4)                def __init__(self, fps=5, codec=None, bitrate=None, extra_args=None,
185: (17)                             metadata=None):
186: (8)                    """
187: (8)                    Parameters
188: (8)                    ----------
189: (8)                    fps : int, default: 5
190: (12)                       Movie frame rate (per second).
191: (8)                    codec : str or None, default: :rc:`animation.codec`
192: (12)                       The codec to use.
193: (8)                    bitrate : int, default: :rc:`animation.bitrate`
194: (12)                       The bitrate of the movie, in kilobits per second.  Higher values
195: (12)                       means higher quality movies, but increase the file size.  A value
196: (12)                       of -1 lets the underlying movie encoder select the bitrate.
197: (8)                    extra_args : list of str or None, optional
198: (12)                       Extra command-line arguments passed to the underlying movie
encoder. These
199: (12)                       arguments are passed last to the encoder, just before the
filename. The
200: (12)                       default, None, means to use :rc:`animation.[name-of-encoder]_args`
for the
201: (12)                       builtin writers.
202: (8)                    metadata : dict[str, str], default: {}
203: (12)                       A dictionary of keys and values for metadata to include in the
204: (12)                       output file. Some keys that may be of use include:
205: (12)                       title, artist, genre, subject, copyright, srcform, comment.
206: (8)                    """
207: (8)                    if type(self) is MovieWriter:
208: (12)                       raise TypeError(
209: (16)                           'MovieWriter cannot be instantiated directly. Please use one '
210: (16)                           'of its subclasses.')
211: (8)                    super().__init__(fps=fps, metadata=metadata, codec=codec,
212: (25)                                    bitrate=bitrate)
213: (8)                    self.frame_format = self.supported_formats[0]
214: (8)                    self.extra_args = extra_args
215: (4)                def _adjust_frame_size(self):
216: (8)                    if self.codec == 'h264':
217: (12)                       wo, ho = self.fig.get_size_inches()
218: (12)                       w, h = adjusted_figsize(wo, ho, self.dpi, 2)
219: (12)                       if (wo, ho) != (w, h):
220: (16)                           self.fig.set_size_inches(w, h, forward=True)
221: (16)                           _log.info('figure size in inches has been adjusted '
222: (26)                                     'from %s x %s to %s x %s', wo, ho, w, h)
223: (8)                    else:
224: (12)                       w, h = self.fig.get_size_inches()
225: (8)                    _log.debug('frame size in pixels is %s x %s', *self.frame_size)
226: (8)                    return w, h
```

```
227: (4)                def setup(self, fig, outfile, dpi=None):
228: (8)                    super().setup(fig, outfile, dpi=dpi)
229: (8)                    self._w, self._h = self._adjust_frame_size()
230: (8)                    self._run()
231: (4)                def _run(self):
232: (8)                    command = self._args()
233: (8)                    _log.info('MovieWriter._run: running command: %s',
234: (18)                             cbook._pformat_subprocess(command))
235: (8)                    PIPE = subprocess.PIPE
236: (8)                    self._proc = subprocess.Popen(
237: (12)                       command, stdin=PIPE, stdout=PIPE, stderr=PIPE,
238: (12)                       creationflags=subprocess_creation_flags)
239: (4)                def finish(self):
240: (8)                    """Finish any processing for writing the movie."""
241: (8)                    out, err = self._proc.communicate()
242: (8)                    out = TextIOWrapper(BytesIO(out)).read()
243: (8)                    err = TextIOWrapper(BytesIO(err)).read()
244: (8)                    if out:
245: (12)                       _log.log(
246: (16)                           logging.WARNING if self._proc.returncode else logging.DEBUG,
247: (16)                           "MovieWriter stdout:\n%s", out)
248: (8)                    if err:
249: (12)                       _log.log(
250: (16)                           logging.WARNING if self._proc.returncode else logging.DEBUG,
251: (16)                           "MovieWriter stderr:\n%s", err)
252: (8)                    if self._proc.returncode:
253: (12)                       raise subprocess.CalledProcessError(
254: (16)                           self._proc.returncode, self._proc.args, out, err)
255: (4)                def grab_frame(self, **savefig_kwargs):
256: (8)                    _validate_grabframe_kwargs(savefig_kwargs)
257: (8)                    _log.debug('MovieWriter.grab_frame: Grabbing frame.')
258: (8)                    self.fig.set_size_inches(self._w, self._h)
259: (8)                    self.fig.savefig(self._proc.stdin, format=self.frame_format,
260: (25)                                     dpi=self.dpi, **savefig_kwargs)
261: (4)                def _args(self):
262: (8)                    """Assemble list of encoder-specific command-line arguments."""
263: (8)                    return NotImplementedError("args needs to be implemented by
subclass.")
264: (4)                @classmethod
265: (4)                def bin_path(cls):
266: (8)                    """
267: (8)                    Return the binary path to the commandline tool used by a specific
268: (8)                    subclass. This is a class method so that the tool can be looked for
269: (8)                    before making a particular MovieWriter subclass available.
270: (8)                    """
271: (8)                    return str(mpl.rcParams[cls._exec_key])
272: (4)                @classmethod
273: (4)                def isAvailable(cls):
274: (8)                    """Return whether a MovieWriter subclass is actually available."""
275: (8)                    return shutil.which(cls.bin_path()) is not None
276: (0)            class FileMovieWriter(MovieWriter):
277: (4)                """
278: (4)                `MovieWriter` for writing to individual files and stitching at the end.
279: (4)                This must be sub-classed to be useful.
280: (4)                """
281: (4)                def __init__(self, *args, **kwargs):
282: (8)                    super().__init__(*args, **kwargs)
283: (8)                    self.frame_format = mpl.rcParams['animation.frame_format']
284: (4)                def setup(self, fig, outfile, dpi=None, frame_prefix=None):
285: (8)                    """
286: (8)                    Setup for writing the movie file.
287: (8)                    Parameters
288: (8)                    ----------
289: (8)                    fig : `~matplotlib.figure.Figure`
290: (12)                       The figure to grab the rendered frames from.
291: (8)                    outfile : str
292: (12)                       The filename of the resulting movie file.
293: (8)                    dpi : float, default: ``fig.dpi``
294: (12)                       The dpi of the output file. This, with the figure size,
```

```
295: (12)                        controls the size in pixels of the resulting movie file.
296: (8)                     frame_prefix : str, optional
297: (12)                        The filename prefix to use for temporary files.  If *None* (the
298: (12)                        default), files are written to a temporary directory which is
299: (12)                        deleted by `finish`; if not *None*, no temporary files are
300: (12)                        deleted.
301: (8)                     """
302: (8)                     Path(outfile).parent.resolve(strict=True)
303: (8)                     self.fig = fig
304: (8)                     self.outfile = outfile
305: (8)                     if dpi is None:
306: (12)                        dpi = self.fig.dpi
307: (8)                     self.dpi = dpi
308: (8)                     self._adjust_frame_size()
309: (8)                     if frame_prefix is None:
310: (12)                        self._tmpdir = TemporaryDirectory()
311: (12)                        self.temp_prefix = str(Path(self._tmpdir.name, 'tmp'))
312: (8)                     else:
313: (12)                        self._tmpdir = None
314: (12)                        self.temp_prefix = frame_prefix
315: (8)                     self._frame_counter = 0  # used for generating sequential file names
316: (8)                     self._temp_paths = list()
317: (8)                     self.fname_format_str = '%s%%07d.%s'
318: (4)                 def __del__(self):
319: (8)                     if hasattr(self, '_tmpdir') and self._tmpdir:
320: (12)                        self._tmpdir.cleanup()
321: (4)                 @property
322: (4)                 def frame_format(self):
323: (8)                     """
324: (8)                     Format (png, jpeg, etc.) to use for saving the frames, which can be
325: (8)                     decided by the individual subclasses.
326: (8)                     """
327: (8)                     return self._frame_format
328: (4)                 @frame_format.setter
329: (4)                 def frame_format(self, frame_format):
330: (8)                     if frame_format in self.supported_formats:
331: (12)                        self._frame_format = frame_format
332: (8)                     else:
333: (12)                        _api.warn_external(
334: (16)                            f"Ignoring file format {frame_format!r} which is not "
335: (16)                            f"supported by {type(self).__name__}; using "
336: (16)                            f"{self.supported_formats[0]} instead.")
337: (12)                        self._frame_format = self.supported_formats[0]
338: (4)                 def _base_temp_name(self):
339: (8)                     return self.fname_format_str % (self.temp_prefix, self.frame_format)
340: (4)                 def grab_frame(self, **savefig_kwargs):
341: (8)                     _validate_grabframe_kwargs(savefig_kwargs)
342: (8)                     path = Path(self._base_temp_name() % self._frame_counter)
343: (8)                     self._temp_paths.append(path)  # Record the filename for later use.
344: (8)                     self._frame_counter += 1  # Ensures each created name is unique.
345: (8)                     _log.debug('FileMovieWriter.grab_frame: Grabbing frame %d to path=%s',
346: (19)                                self._frame_counter, path)
347: (8)                     with open(path, 'wb') as sink:  # Save figure to the sink.
348: (12)                        self.fig.savefig(sink, format=self.frame_format, dpi=self.dpi,
349: (29)                                         **savefig_kwargs)
350: (4)                 def finish(self):
351: (8)                     try:
352: (12)                        self._run()
353: (12)                        super().finish()
354: (8)                     finally:
355: (12)                        if self._tmpdir:
356: (16)                            _log.debug(
357: (20)                                'MovieWriter: clearing temporary path=%s', self._tmpdir
358: (16)                            )
359: (16)                            self._tmpdir.cleanup()
360: (0)             @writers.register('pillow')
361: (0)             class PillowWriter(AbstractMovieWriter):
362: (4)                 @classmethod
363: (4)                 def isAvailable(cls):
```

```
364: (8)                    return True
365: (4)                def setup(self, fig, outfile, dpi=None):
366: (8)                    super().setup(fig, outfile, dpi=dpi)
367: (8)                    self._frames = []
368: (4)                def grab_frame(self, **savefig_kwargs):
369: (8)                    _validate_grabframe_kwargs(savefig_kwargs)
370: (8)                    buf = BytesIO()
371: (8)                    self.fig.savefig(
372: (12)                       buf, **{**savefig_kwargs, "format": "rgba", "dpi": self.dpi})
373: (8)                    self._frames.append(Image.frombuffer(
374: (12)                       "RGBA", self.frame_size, buf.getbuffer(), "raw", "RGBA", 0, 1))
375: (4)                def finish(self):
376: (8)                    self._frames[0].save(
377: (12)                       self.outfile, save_all=True, append_images=self._frames[1:],
378: (12)                       duration=int(1000 / self.fps), loop=0)
379: (0)            class FFMpegBase:
380: (4)                """
381: (4)                Mixin class for FFMpeg output.
382: (4)                This is a base class for the concrete `FFMpegWriter` and
`FFMpegFileWriter`
383: (4)                classes.
384: (4)                """
385: (4)                _exec_key = 'animation.ffmpeg_path'
386: (4)                _args_key = 'animation.ffmpeg_args'
387: (4)                @property
388: (4)                def output_args(self):
389: (8)                    args = []
390: (8)                    if Path(self.outfile).suffix == '.gif':
391: (12)                       self.codec = 'gif'
392: (8)                    else:
393: (12)                       args.extend(['-vcodec', self.codec])
394: (8)                    extra_args = (self.extra_args if self.extra_args is not None
395: (22)                                 else mpl.rcParams[self._args_key])
396: (8)                    if self.codec == 'h264' and '-pix_fmt' not in extra_args:
397: (12)                       args.extend(['-pix_fmt', 'yuv420p'])
398: (8)                    elif self.codec == 'gif' and '-filter_complex' not in extra_args:
399: (12)                       args.extend(['-filter_complex',
400: (25)                                    'split [a][b];[a] palettegen [p];[b][p] paletteuse'])
401: (8)                    if self.bitrate > 0:
402: (12)                       args.extend(['-b', '%dk' % self.bitrate])  # %dk: bitrate in kbps.
403: (8)                    for k, v in self.metadata.items():
404: (12)                       args.extend(['-metadata', f'{k}={v}'])
405: (8)                    args.extend(extra_args)
406: (8)                    return args + ['-y', self.outfile]
407: (0)            @writers.register('ffmpeg')
408: (0)            class FFMpegWriter(FFMpegBase, MovieWriter):
409: (4)                """
410: (4)                Pipe-based ffmpeg writer.
411: (4)                Frames are streamed directly to ffmpeg via a pipe and written in a single
pass.
412: (4)                This effectively works as a slideshow input to ffmpeg with the fps passed
as
413: (4)                ``-framerate``, so see also `their notes on frame rates`_ for further
details.
414: (4)                .. _their notes on frame rates:
https://trac.ffmpeg.org/wiki/Slideshow#Framerates
415: (4)                """
416: (4)                def _args(self):
417: (8)                    args = [self.bin_path(), '-f', 'rawvideo', '-vcodec', 'rawvideo',
418: (16)                           '-s', '%dx%d' % self.frame_size, '-pix_fmt',
self.frame_format,
419: (16)                           '-framerate', str(self.fps)]
420: (8)                    if _log.getEffectiveLevel() > logging.DEBUG:
421: (12)                       args += ['-loglevel', 'error']
422: (8)                    args += ['-i', 'pipe:'] + self.output_args
423: (8)                    return args
424: (0)            @writers.register('ffmpeg_file')
425: (0)            class FFMpegFileWriter(FFMpegBase, FileMovieWriter):
426: (4)                """
```

```
427: (4)                File-based ffmpeg writer.
428: (4)                Frames are written to temporary files on disk and then stitched together
at the end.
429: (4)                This effectively works as a slideshow input to ffmpeg with the fps passed
as
430: (4)                ``-framerate``, so see also `their notes on frame rates`_ for further
details.
431: (4)                .. _their notes on frame rates:
https://trac.ffmpeg.org/wiki/Slideshow#Framerates
432: (4)                """
433: (4)                supported_formats = ['png', 'jpeg', 'tiff', 'raw', 'rgba']
434: (4)                def _args(self):
435: (8)                    args = []
436: (8)                    if self.frame_format in {'raw', 'rgba'}:
437: (12)                       args += [
438: (16)                           '-f', 'image2', '-vcodec', 'rawvideo',
439: (16)                           '-video_size', '%dx%d' % self.frame_size,
440: (16)                           '-pixel_format', 'rgba',
441: (12)                       ]
442: (8)                    args += ['-framerate', str(self.fps), '-i', self._base_temp_name()]
443: (8)                    if not self._tmpdir:
444: (12)                       args += ['-frames:v', str(self._frame_counter)]
445: (8)                    if _log.getEffectiveLevel() > logging.DEBUG:
446: (12)                       args += ['-loglevel', 'error']
447: (8)                    return [self.bin_path(), *args, *self.output_args]
448: (0)            class ImageMagickBase:
449: (4)                """
450: (4)                Mixin class for ImageMagick output.
451: (4)                This is a base class for the concrete `ImageMagickWriter` and
452: (4)                `ImageMagickFileWriter` classes, which define an ``input_names`` attribute
453: (4)                (or property) specifying the input names passed to ImageMagick.
454: (4)                """
455: (4)                _exec_key = 'animation.convert_path'
456: (4)                _args_key = 'animation.convert_args'
457: (4)                def _args(self):
458: (8)                    fmt = "rgba" if self.frame_format == "raw" else self.frame_format
459: (8)                    extra_args = (self.extra_args if self.extra_args is not None
460: (22)                                 else mpl.rcParams[self._args_key])
461: (8)                    return [
462: (12)                       self.bin_path(),
463: (12)                       "-size", "%ix%i" % self.frame_size,
464: (12)                       "-depth", "8",
465: (12)                       "-delay", str(100 / self.fps),
466: (12)                       "-loop", "0",
467: (12)                       f"{fmt}:{self.input_names}",
468: (12)                       *extra_args,
469: (12)                       self.outfile,
470: (8)                    ]
471: (4)                @classmethod
472: (4)                def bin_path(cls):
473: (8)                    binpath = super().bin_path()
474: (8)                    if binpath == 'convert':
475: (12)                       binpath = mpl._get_executable_info('magick').executable
476: (8)                    return binpath
477: (4)                @classmethod
478: (4)                def isAvailable(cls):
479: (8)                    try:
480: (12)                       return super().isAvailable()
481: (8)                    except mpl.ExecutableNotFoundError as _enf:
482: (12)                       _log.debug('ImageMagick unavailable due to: %s', _enf)
483: (12)                       return False
484: (0)            @writers.register('imagemagick')
485: (0)            class ImageMagickWriter(ImageMagickBase, MovieWriter):
486: (4)                """
487: (4)                Pipe-based animated gif writer.
488: (4)                Frames are streamed directly to ImageMagick via a pipe and written
489: (4)                in a single pass.
490: (4)                """
491: (4)                input_names = "-"  # stdin
```

```
492: (0)            @writers.register('imagemagick_file')
493: (0)            class ImageMagickFileWriter(ImageMagickBase, FileMovieWriter):
494: (4)                """
495: (4)                File-based animated gif writer.
496: (4)                Frames are written to temporary files on disk and then stitched
497: (4)                together at the end.
498: (4)                """
499: (4)                supported_formats = ['png', 'jpeg', 'tiff', 'raw', 'rgba']
500: (4)                input_names = property(
501: (8)                    lambda self: f'{self.temp_prefix}*.{self.frame_format}')
502: (0)            def _included_frames(frame_count, frame_format, frame_dir):
503: (4)                return INCLUDED_FRAMES.format(Nframes=frame_count,
504: (34)                                              frame_dir=frame_dir,
505: (34)                                              frame_format=frame_format)
506: (0)            def _embedded_frames(frame_list, frame_format):
507: (4)                """frame_list should be a list of base64-encoded png files"""
508: (4)                if frame_format == 'svg':
509: (8)                    frame_format = 'svg+xml'
510: (4)                template = '  frames[{0}] = "{2}"\n'
511: (4)                return "\n" + "".join(
512: (8)                    template.format(i, frame_format, frame_data.replace('\n', '\\\n'))
513: (8)                    for i, frame_data in enumerate(frame_list))
514: (0)            @writers.register('html')
515: (0)            class HTMLWriter(FileMovieWriter):
516: (4)                """Writer for JavaScript-based HTML movies."""
517: (4)                supported_formats = ['png', 'jpeg', 'tiff', 'svg']
518: (4)                @classmethod
519: (4)                def isAvailable(cls):
520: (8)                    return True
521: (4)                def __init__(self, fps=30, codec=None, bitrate=None, extra_args=None,
522: (17)                            metadata=None, embed_frames=False, default_mode='loop',
523: (17)                            embed_limit=None):
524: (8)                    if extra_args:
525: (12)                       _log.warning("HTMLWriter ignores 'extra_args'")
526: (8)                    extra_args = ()  # Don't lookup nonexistent rcParam[args_key].
527: (8)                    self.embed_frames = embed_frames
528: (8)                    self.default_mode = default_mode.lower()
529: (8)                    _api.check_in_list(['loop', 'once', 'reflect'],
530: (27)                                      default_mode=self.default_mode)
531: (8)                    self._bytes_limit = mpl._val_or_rc(embed_limit,
'animation.embed_limit')
532: (8)                    self._bytes_limit *= 1024 * 1024
533: (8)                    super().__init__(fps, codec, bitrate, extra_args, metadata)
534: (4)                def setup(self, fig, outfile, dpi=None, frame_dir=None):
535: (8)                    outfile = Path(outfile)
536: (8)                    _api.check_in_list(['.html', '.htm'],
outfile_extension=outfile.suffix)
537: (8)                    self._saved_frames = []
538: (8)                    self._total_bytes = 0
539: (8)                    self._hit_limit = False
540: (8)                    if not self.embed_frames:
541: (12)                       if frame_dir is None:
542: (16)                           frame_dir = outfile.with_name(outfile.stem + '_frames')
543: (12)                       frame_dir.mkdir(parents=True, exist_ok=True)
544: (12)                       frame_prefix = frame_dir / 'frame'
545: (8)                    else:
546: (12)                       frame_prefix = None
547: (8)                    super().setup(fig, outfile, dpi, frame_prefix)
548: (8)                    self._clear_temp = False
549: (4)                def grab_frame(self, **savefig_kwargs):
550: (8)                    _validate_grabframe_kwargs(savefig_kwargs)
551: (8)                    if self.embed_frames:
552: (12)                       if self._hit_limit:
553: (16)                           return
554: (12)                       f = BytesIO()
555: (12)                       self.fig.savefig(f, format=self.frame_format,
556: (29)                                        dpi=self.dpi, **savefig_kwargs)
557: (12)                       imgdata64 = base64.encodebytes(f.getvalue()).decode('ascii')
558: (12)                       self._total_bytes += len(imgdata64)
```

```
559: (12)                        if self._total_bytes >= self._bytes_limit:
560: (16)                            _log.warning(
561: (20)                                "Animation size has reached %s bytes, exceeding the limit
"
562: (20)                                "of %s. If you're sure you want a larger animation "
563: (20)                                "embedded, set the animation.embed_limit rc parameter to "
564: (20)                                "a larger value (in MB). This and further frames will be "
565: (20)                                "dropped.", self._total_bytes, self._bytes_limit)
566: (16)                            self._hit_limit = True
567: (12)                        else:
568: (16)                            self._saved_frames.append(imgdata64)
569: (8)                    else:
570: (12)                        return super().grab_frame(**savefig_kwargs)
571: (4)                def finish(self):
572: (8)                    if self.embed_frames:
573: (12)                        fill_frames = _embedded_frames(self._saved_frames,
574: (43)                                                       self.frame_format)
575: (12)                        frame_count = len(self._saved_frames)
576: (8)                    else:
577: (12)                        frame_count = len(self._temp_paths)
578: (12)                        fill_frames = _included_frames(
579: (16)                            frame_count, self.frame_format,
580: (16)                            self._temp_paths[0].parent.relative_to(self.outfile.parent))
581: (8)                    mode_dict = dict(once_checked='',
582: (25)                                     loop_checked='',
583: (25)                                     reflect_checked='')
584: (8)                    mode_dict[self.default_mode + '_checked'] = 'checked'
585: (8)                    interval = 1000 // self.fps
586: (8)                    with open(self.outfile, 'w') as of:
587: (12)                        of.write(JS_INCLUDE + STYLE_INCLUDE)
588: (12)                        of.write(DISPLAY_TEMPLATE.format(id=uuid.uuid4().hex,
589: (45)                                                         Nframes=frame_count,
590: (45)                                                         fill_frames=fill_frames,
591: (45)                                                         interval=interval,
592: (45)                                                         **mode_dict))
593: (8)                    if self._tmpdir:
594: (12)                        _log.debug('MovieWriter: clearing temporary path=%s',
self._tmpdir)
595: (12)                        self._tmpdir.cleanup()
596: (0)            class Animation:
597: (4)                """
598: (4)                A base class for Animations.
599: (4)                This class is not usable as is, and should be subclassed to provide needed
600: (4)                behavior.
601: (4)                .. note::
602: (8)                    You must store the created Animation in a variable that lives as long
603: (8)                    as the animation should run. Otherwise, the Animation object will be
604: (8)                    garbage-collected and the animation stops.
605: (4)                Parameters
606: (4)                ----------
607: (4)                fig : `~matplotlib.figure.Figure`
608: (8)                    The figure object used to get needed events, such as draw or resize.
609: (4)                event_source : object, optional
610: (8)                    A class that can run a callback when desired events
611: (8)                    are generated, as well as be stopped and started.
612: (8)                    Examples include timers (see `TimedAnimation`) and file
613: (8)                    system notifications.
614: (4)                blit : bool, default: False
615: (8)                    Whether blitting is used to optimize drawing.  If the backend does not
616: (8)                    support blitting, then this parameter has no effect.
617: (4)                See Also
618: (4)                --------
619: (4)                FuncAnimation,  ArtistAnimation
620: (4)                """
621: (4)                def __init__(self, fig, event_source=None, blit=False):
622: (8)                    self._draw_was_started = False
623: (8)                    self._fig = fig
624: (8)                    self._blit = blit and fig.canvas.supports_blit
625: (8)                    self.frame_seq = self.new_frame_seq()
```

```
626: (8)                    self.event_source = event_source
627: (8)                    self._first_draw_id = fig.canvas.mpl_connect('draw_event',
self._start)
628: (8)                    self._close_id = self._fig.canvas.mpl_connect('close_event',
629: (54)                                                                 self._stop)
630: (8)                    if self._blit:
631: (12)                       self._setup_blit()
632: (4)                def __del__(self):
633: (8)                    if not getattr(self, '_draw_was_started', True):
634: (12)                       warnings.warn(
635: (16)                           'Animation was deleted without rendering anything. This is '
636: (16)                           'most likely not intended. To prevent deletion, assign the '
637: (16)                           'Animation to a variable, e.g. `anim`, that exists until you '
638: (16)                           'output the Animation using `plt.show()` or '
639: (16)                           '`anim.save()`.'
640: (12)                       )
641: (4)                def _start(self, *args):
642: (8)                    """
643: (8)                    Starts interactive animation. Adds the draw frame command to the GUI
644: (8)                    handler, calls show to start the event loop.
645: (8)                    """
646: (8)                    if self._fig.canvas.is_saving():
647: (12)                       return
648: (8)                    self._fig.canvas.mpl_disconnect(self._first_draw_id)
649: (8)                    self._init_draw()
650: (8)                    self.event_source.add_callback(self._step)
651: (8)                    self.event_source.start()
652: (4)                def _stop(self, *args):
653: (8)                    if self._blit:
654: (12)                       self._fig.canvas.mpl_disconnect(self._resize_id)
655: (8)                    self._fig.canvas.mpl_disconnect(self._close_id)
656: (8)                    self.event_source.remove_callback(self._step)
657: (8)                    self.event_source = None
658: (4)                def save(self, filename, writer=None, fps=None, dpi=None, codec=None,
659: (13)                         bitrate=None, extra_args=None, metadata=None, extra_anim=None,
660: (13)                         savefig_kwargs=None, *, progress_callback=None):
661: (8)                    """
662: (8)                    Save the animation as a movie file by drawing every frame.
663: (8)                    Parameters
664: (8)                    ----------
665: (8)                    filename : str
666: (12)                       The output filename, e.g., :file:`mymovie.mp4`.
667: (8)                    writer : `MovieWriter` or str, default: :rc:`animation.writer`
668: (12)                       A `MovieWriter` instance to use or a key that identifies a
669: (12)                       class to use, such as 'ffmpeg'.
670: (8)                    fps : int, optional
671: (12)                       Movie frame rate (per second).  If not set, the frame rate from
the
672: (12)                       animation's frame interval.
673: (8)                    dpi : float, default: :rc:`savefig.dpi`
674: (12)                       Controls the dots per inch for the movie frames.  Together with
675: (12)                       the figure's size in inches, this controls the size of the movie.
676: (8)                    codec : str, default: :rc:`animation.codec`.
677: (12)                       The video codec to use.  Not all codecs are supported by a given
678: (12)                       `MovieWriter`.
679: (8)                    bitrate : int, default: :rc:`animation.bitrate`
680: (12)                       The bitrate of the movie, in kilobits per second.  Higher values
681: (12)                       means higher quality movies, but increase the file size.  A value
682: (12)                       of -1 lets the underlying movie encoder select the bitrate.
683: (8)                    extra_args : list of str or None, optional
684: (12)                       Extra command-line arguments passed to the underlying movie
encoder. These
685: (12)                       arguments are passed last to the encoder, just before the output
filename.
686: (12)                       The default, None, means to use :rc:`animation.[name-of-
encoder]_args` for
687: (12)                       the builtin writers.
688: (8)                    metadata : dict[str, str], default: {}
689: (12)                       Dictionary of keys and values for metadata to include in
```

```
690: (12)                        the output file. Some keys that may be of use include:
691: (12)                        title, artist, genre, subject, copyright, srcform, comment.
692: (8)                    extra_anim : list, default: []
693: (12)                        Additional `Animation` objects that should be included
694: (12)                        in the saved movie file. These need to be from the same
695: (12)                        `.Figure` instance. Also, animation frames will
696: (12)                        just be simply combined, so there should be a 1:1 correspondence
697: (12)                        between the frames from the different animations.
698: (8)                    savefig_kwargs : dict, default: {}
699: (12)                        Keyword arguments passed to each `~.Figure.savefig` call used to
700: (12)                        save the individual frames.
701: (8)                    progress_callback : function, optional
702: (12)                        A callback function that will be called for every frame to notify
703: (12)                        the saving progress. It must have the signature ::
704: (16)                            def func(current_frame: int, total_frames: int) -> Any
705: (12)                        where *current_frame* is the current frame number and
*total_frames* is the
706: (12)                        total number of frames to be saved. *total_frames* is set to None,
if the
707: (12)                        total number of frames cannot be determined. Return values may
exist but are
708: (12)                        ignored.
709: (12)                        Example code to write the progress to stdout::
710: (16)                            progress_callback = lambda i, n: print(f'Saving frame
{i}/{n}')
711: (8)                    Notes
712: (8)                    -----
713: (8)                    *fps*, *codec*, *bitrate*, *extra_args* and *metadata* are used to
714: (8)                    construct a `.MovieWriter` instance and can only be passed if
715: (8)                    *writer* is a string.  If they are passed as non-*None* and *writer*
716: (8)                    is a `.MovieWriter`, a `RuntimeError` will be raised.
717: (8)                    """
718: (8)                    all_anim = [self]
719: (8)                    if extra_anim is not None:
720: (12)                       all_anim.extend(anim for anim in extra_anim
721: (28)                                       if anim._fig is self._fig)
722: (8)                    for anim in all_anim:
723: (12)                       anim._draw_was_started = True
724: (8)                    if writer is None:
725: (12)                       writer = mpl.rcParams['animation.writer']
726: (8)                    elif (not isinstance(writer, str) and
727: (14)                         any(arg is not None
728: (18)                             for arg in (fps, codec, bitrate, extra_args, metadata))):
729: (12)                       raise RuntimeError('Passing in values for arguments '
730: (31)                                          'fps, codec, bitrate, extra_args, or metadata '
731: (31)                                          'is not supported when writer is an existing '
732: (31)                                          'MovieWriter instance. These should instead be
'
733: (31)                                          'passed as arguments when creating the '
734: (31)                                          'MovieWriter instance.')
735: (8)                    if savefig_kwargs is None:
736: (12)                       savefig_kwargs = {}
737: (8)                    else:
738: (12)                       savefig_kwargs = dict(savefig_kwargs)
739: (8)                    if fps is None and hasattr(self, '_interval'):
740: (12)                       fps = 1000. / self._interval
741: (8)                    dpi = mpl._val_or_rc(dpi, 'savefig.dpi')
742: (8)                    if dpi == 'figure':
743: (12)                       dpi = self._fig.dpi
744: (8)                    writer_kwargs = {}
745: (8)                    if codec is not None:
746: (12)                       writer_kwargs['codec'] = codec
747: (8)                    if bitrate is not None:
748: (12)                       writer_kwargs['bitrate'] = bitrate
749: (8)                    if extra_args is not None:
750: (12)                       writer_kwargs['extra_args'] = extra_args
751: (8)                    if metadata is not None:
752: (12)                       writer_kwargs['metadata'] = metadata
753: (8)                    if isinstance(writer, str):
```

```
(12)                        try:
(16)                            writer_cls = writers[writer]
(12)                        except RuntimeError:  # Raised if not available.
(16)                            writer_cls = PillowWriter  # Always available.
(16)                            _log.warning("MovieWriter %s unavailable; using Pillow "
(29)                                         "instead.", writer)
(12)                        writer = writer_cls(fps, **writer_kwargs)
(8)                     _log.info('Animation.save using %s', type(writer))
(8)                     if 'bbox_inches' in savefig_kwargs:
(12)                        _log.warning("Warning: discarding the 'bbox_inches' argument in "
(25)                                     "'savefig_kwargs' as it may cause frame size "
(25)                                     "to vary, which is inappropriate for animation.")
(12)                        savefig_kwargs.pop('bbox_inches')
(8)                     facecolor = savefig_kwargs.get('facecolor',
(39)                                                   mpl.rcParams['savefig.facecolor'])
(8)                     if facecolor == 'auto':
(12)                        facecolor = self._fig.get_facecolor()
(8)                     def _pre_composite_to_white(color):
(12)                        r, g, b, a = mcolors.to_rgba(color)
(12)                        return a * np.array([r, g, b]) + 1 - a
(8)                     savefig_kwargs['facecolor'] = _pre_composite_to_white(facecolor)
(8)                     savefig_kwargs['transparent'] = False   # just to be safe!
(8)                     with writer.saving(self._fig, filename, dpi), \
(13)                         cbook._setattr_cm(self._fig.canvas, _is_saving=True,
manager=None):
(12)                        for anim in all_anim:
(16)                            anim._init_draw()  # Clear the initial frame
(12)                        frame_number = 0
(12)                        save_count_list = [getattr(a, '_save_count', None)
(31)                                           for a in all_anim]
(12)                        if None in save_count_list:
(16)                            total_frames = None
(12)                        else:
(16)                            total_frames = sum(save_count_list)
(12)                        for data in zip(*[a.new_saved_frame_seq() for a in all_anim]):
(16)                            for anim, d in zip(all_anim, data):
(20)                                anim._draw_next_frame(d, blit=False)
(20)                                if progress_callback is not None:
(24)                                    progress_callback(frame_number, total_frames)
(24)                                    frame_number += 1
(16)                            writer.grab_frame(**savefig_kwargs)
(4)                 def _step(self, *args):
(8)                     """
(8)                     Handler for getting events. By default, gets the next frame in the
(8)                     sequence and hands the data off to be drawn.
(8)                     """
(8)                     try:
(12)                        framedata = next(self.frame_seq)
(12)                        self._draw_next_frame(framedata, self._blit)
(12)                        return True
(8)                     except StopIteration:
(12)                        return False
(4)                 def new_frame_seq(self):
(8)                     """Return a new sequence of frame information."""
(8)                     return iter(self._framedata)
(4)                 def new_saved_frame_seq(self):
(8)                     """Return a new sequence of saved/cached frame information."""
(8)                     return self.new_frame_seq()
(4)                 def _draw_next_frame(self, framedata, blit):
(8)                     self._pre_draw(framedata, blit)
(8)                     self._draw_frame(framedata)
(8)                     self._post_draw(framedata, blit)
(4)                 def _init_draw(self):
(8)                     self._draw_was_started = True
(4)                 def _pre_draw(self, framedata, blit):
(8)                     if blit:
(12)                        self._blit_clear(self._drawn_artists)
(4)                 def _draw_frame(self, framedata):
(8)                     raise NotImplementedError('Needs to be implemented by subclasses to'
```

```
(34)                                            ' actually make an animation.')
(4)                 def _post_draw(self, framedata, blit):
(8)                     if blit and self._drawn_artists:
(12)                        self._blit_draw(self._drawn_artists)
(8)                     else:
(12)                        self._fig.canvas.draw_idle()
(4)                 def _blit_draw(self, artists):
(8)                     updated_ax = {a.axes for a in artists}
(8)                     for ax in updated_ax:
(12)                        cur_view = ax._get_view()
(12)                        view, bg = self._blit_cache.get(ax, (object(), None))
(12)                        if cur_view != view:
(16)                            self._blit_cache[ax] = (
(20)                                cur_view, ax.figure.canvas.copy_from_bbox(ax.bbox))
(8)                     for a in artists:
(12)                        a.axes.draw_artist(a)
(8)                     for ax in updated_ax:
(12)                        ax.figure.canvas.blit(ax.bbox)
(4)                 def _blit_clear(self, artists):
(8)                     axes = {a.axes for a in artists}
(8)                     for ax in axes:
(12)                        try:
(16)                            view, bg = self._blit_cache[ax]
(12)                        except KeyError:
(16)                            continue
(12)                        if ax._get_view() == view:
(16)                            ax.figure.canvas.restore_region(bg)
(12)                        else:
(16)                            self._blit_cache.pop(ax)
(4)                 def _setup_blit(self):
(8)                     self._blit_cache = dict()
(8)                     self._drawn_artists = []
(8)                     self._post_draw(None, self._blit)
(8)                     self._init_draw()
(8)                     self._resize_id = self._fig.canvas.mpl_connect('resize_event',
(55)                                                                    self._on_resize)
(4)                 def _on_resize(self, event):
(8)                     self._fig.canvas.mpl_disconnect(self._resize_id)
(8)                     self.event_source.stop()
(8)                     self._blit_cache.clear()
(8)                     self._init_draw()
(8)                     self._resize_id = self._fig.canvas.mpl_connect('draw_event',
(55)                                                                    self._end_redraw)
(4)                 def _end_redraw(self, event):
(8)                     self._post_draw(None, False)
(8)                     self.event_source.start()
(8)                     self._fig.canvas.mpl_disconnect(self._resize_id)
(8)                     self._resize_id = self._fig.canvas.mpl_connect('resize_event',
(55)                                                                    self._on_resize)
(4)                 def to_html5_video(self, embed_limit=None):
(8)                     """
(8)                     Convert the animation to an HTML5 ``<video>`` tag.
(8)                     This saves the animation as an h264 video, encoded in base64
(8)                     directly into the HTML5 video tag. This respects
:rc:`animation.writer`
(8)                     and :rc:`animation.bitrate`. This also makes use of the
(8)                     *interval* to control the speed, and uses the *repeat*
(8)                     parameter to decide whether to loop.
(8)                     Parameters
(8)                     ----------
(8)                     embed_limit : float, optional
(12)                        Limit, in MB, of the returned animation. No animation is created
(12)                        if the limit is exceeded.
(12)                        Defaults to :rc:`animation.embed_limit` = 20.0.
(8)                     Returns
(8)                     -------
(8)                     str
(12)                        An HTML5 video tag with the animation embedded as base64 encoded
(12)                        h264 video.
```

```
890: (12)                        If the *embed_limit* is exceeded, this returns the string
891: (12)                        "Video too large to embed."
892: (8)                     """
893: (8)                     VIDEO_TAG = r'''<video {size} {options}>
894: (2)               <source type="video/mp4" src="data:video/mp4base64,{video}">
895: (2)               Your browser does not support the video tag.
896: (0)             </video>'''
897: (8)                     if not hasattr(self, '_base64_video'):
898: (12)                        embed_limit = mpl._val_or_rc(embed_limit, 'animation.embed_limit')
899: (12)                        embed_limit *= 1024 * 1024
900: (12)                        with TemporaryDirectory() as tmpdir:
901: (16)                            path = Path(tmpdir, "temp.m4v")
902: (16)                            Writer = writers[mpl.rcParams['animation.writer']]
903: (16)                            writer = Writer(codec='h264',
904: (32)                                            bitrate=mpl.rcParams['animation.bitrate'],
905: (32)                                            fps=1000. / self._interval)
906: (16)                            self.save(str(path), writer=writer)
907: (16)                            vid64 = base64.encodebytes(path.read_bytes())
908: (12)                        vid_len = len(vid64)
909: (12)                        if vid_len >= embed_limit:
910: (16)                            _log.warning(
911: (20)                                "Animation movie is %s bytes, exceeding the limit of %s. "
912: (20)                                "If you're sure you want a large animation embedded, set "
913: (20)                                "the animation.embed_limit rc parameter to a larger value
"
914: (20)                                "(in MB).", vid_len, embed_limit)
915: (12)                        else:
916: (16)                            self._base64_video = vid64.decode('ascii')
917: (16)                            self._video_size = 'width="{}" height="{}"'.format(
918: (24)                                    *writer.frame_size)
919: (8)                     if hasattr(self, '_base64_video'):
920: (12)                        options = ['controls', 'autoplay']
921: (12)                        if getattr(self, '_repeat', False):
922: (16)                            options.append('loop')
923: (12)                        return VIDEO_TAG.format(video=self._base64_video,
924: (36)                                                size=self._video_size,
925: (36)                                                options=' '.join(options))
926: (8)                     else:
927: (12)                        return 'Video too large to embed.'
928: (4)                 def to_jshtml(self, fps=None, embed_frames=True, default_mode=None):
929: (8)                     """
930: (8)                     Generate HTML representation of the animation.
931: (8)                     Parameters
932: (8)                     ----------
933: (8)                     fps : int, optional
934: (12)                        Movie frame rate (per second). If not set, the frame rate from
935: (12)                        the animation's frame interval.
936: (8)                     embed_frames : bool, optional
937: (8)                     default_mode : str, optional
938: (12)                        What to do when the animation ends. Must be one of ``{'loop',
939: (12)                        'once', 'reflect'}``. Defaults to ``'loop'`` if the *repeat*
940: (12)                        parameter is True, otherwise ``'once'``.
941: (8)                     """
942: (8)                     if fps is None and hasattr(self, '_interval'):
943: (12)                        fps = 1000 / self._interval
944: (8)                     if default_mode is None:
945: (12)                        default_mode = 'loop' if getattr(self, '_repeat',
946: (45)                                                         False) else 'once'
947: (8)                     if not hasattr(self, "_html_representation"):
948: (12)                        with TemporaryDirectory() as tmpdir:
949: (16)                            path = Path(tmpdir, "temp.html")
950: (16)                            writer = HTMLWriter(fps=fps,
951: (36)                                                embed_frames=embed_frames,
952: (36)                                                default_mode=default_mode)
953: (16)                            self.save(str(path), writer=writer)
954: (16)                            self._html_representation = path.read_text()
955: (8)                     return self._html_representation
956: (4)                 def _repr_html_(self):
957: (8)                     """IPython display hook for rendering."""
```

```
958: (8)                    fmt = mpl.rcParams['animation.html']
959: (8)                    if fmt == 'html5':
960: (12)                       return self.to_html5_video()
961: (8)                    elif fmt == 'jshtml':
962: (12)                       return self.to_jshtml()
963: (4)                def pause(self):
964: (8)                    """Pause the animation."""
965: (8)                    self.event_source.stop()
966: (8)                    if self._blit:
967: (12)                       for artist in self._drawn_artists:
968: (16)                           artist.set_animated(False)
969: (4)                def resume(self):
970: (8)                    """Resume the animation."""
971: (8)                    self.event_source.start()
972: (8)                    if self._blit:
973: (12)                       for artist in self._drawn_artists:
974: (16)                           artist.set_animated(True)
975: (0)            class TimedAnimation(Animation):
976: (4)                """
977: (4)                `Animation` subclass for time-based animation.
978: (4)                A new frame is drawn every *interval* milliseconds.
979: (4)                .. note::
980: (8)                    You must store the created Animation in a variable that lives as long
981: (8)                    as the animation should run. Otherwise, the Animation object will be
982: (8)                    garbage-collected and the animation stops.
983: (4)                Parameters
984: (4)                ----------
985: (4)                fig : `~matplotlib.figure.Figure`
986: (8)                    The figure object used to get needed events, such as draw or resize.
987: (4)                interval : int, default: 200
988: (8)                    Delay between frames in milliseconds.
989: (4)                repeat_delay : int, default: 0
990: (8)                    The delay in milliseconds between consecutive animation runs, if
991: (8)                    *repeat* is True.
992: (4)                repeat : bool, default: True
993: (8)                    Whether the animation repeats when the sequence of frames is
completed.
994: (4)                blit : bool, default: False
995: (8)                    Whether blitting is used to optimize drawing.
996: (4)                """
997: (4)                def __init__(self, fig, interval=200, repeat_delay=0, repeat=True,
998: (17)                            event_source=None, *args, **kwargs):
999: (8)                    self._interval = interval
1000: (8)                   self._repeat_delay = repeat_delay if repeat_delay is not None else 0
1001: (8)                   self._repeat = repeat
1002: (8)                   if event_source is None:
1003: (12)                      event_source = fig.canvas.new_timer(interval=self._interval)
1004: (8)                   super().__init__(fig, event_source=event_source, *args, **kwargs)
1005: (4)               def _step(self, *args):
1006: (8)                   """Handler for getting events."""
1007: (8)                   still_going = super()._step(*args)
1008: (8)                   if not still_going:
1009: (12)                      if self._repeat:
1010: (16)                          self._init_draw()
1011: (16)                          self.frame_seq = self.new_frame_seq()
1012: (16)                          self.event_source.interval = self._repeat_delay
1013: (16)                          return True
1014: (12)                      else:
1015: (16)                          self.pause()
1016: (16)                          if self._blit:
1017: (20)                              self._fig.canvas.mpl_disconnect(self._resize_id)
1018: (16)                          self._fig.canvas.mpl_disconnect(self._close_id)
1019: (16)                          self.event_source = None
1020: (16)                          return False
1021: (8)                   self.event_source.interval = self._interval
1022: (8)                   return True
1023: (0)           class ArtistAnimation(TimedAnimation):
1024: (4)               """
1025: (4)               `TimedAnimation` subclass that creates an animation by using a fixed
```

```
(4)                 set of `.Artist` objects.
(4)                 Before creating an instance, all plotting should have taken place
(4)                 and the relevant artists saved.
(4)                 .. note::
(8)                     You must store the created Animation in a variable that lives as long
(8)                     as the animation should run. Otherwise, the Animation object will be
(8)                     garbage-collected and the animation stops.
(4)                 Parameters
(4)                 ----------
(4)                 fig : `~matplotlib.figure.Figure`
(8)                     The figure object used to get needed events, such as draw or resize.
(4)                 artists : list
(8)                     Each list entry is a collection of `.Artist` objects that are made
(8)                     visible on the corresponding frame.  Other artists are made invisible.
(4)                 interval : int, default: 200
(8)                     Delay between frames in milliseconds.
(4)                 repeat_delay : int, default: 0
(8)                     The delay in milliseconds between consecutive animation runs, if
(8)                     *repeat* is True.
(4)                 repeat : bool, default: True
(8)                     Whether the animation repeats when the sequence of frames is
completed.
(4)                 blit : bool, default: False
(8)                     Whether blitting is used to optimize drawing.
(4)                 """
(4)                 def __init__(self, fig, artists, *args, **kwargs):
(8)                     self._drawn_artists = []
(8)                     self._framedata = artists
(8)                     super().__init__(fig, *args, **kwargs)
(4)                 def _init_draw(self):
(8)                     super()._init_draw()
(8)                     figs = set()
(8)                     for f in self.new_frame_seq():
(12)                        for artist in f:
(16)                            artist.set_visible(False)
(16)                            artist.set_animated(self._blit)
(16)                            if artist.get_figure() not in figs:
(20)                                figs.add(artist.get_figure())
(8)                     for fig in figs:
(12)                        fig.canvas.draw_idle()
(4)                 def _pre_draw(self, framedata, blit):
(8)                     """Clears artists from the last frame."""
(8)                     if blit:
(12)                        self._blit_clear(self._drawn_artists)
(8)                     else:
(12)                        for artist in self._drawn_artists:
(16)                            artist.set_visible(False)
(4)                 def _draw_frame(self, artists):
(8)                     self._drawn_artists = artists
(8)                     for artist in artists:
(12)                        artist.set_visible(True)
(0)             class FuncAnimation(TimedAnimation):
(4)                 """
(4)                 `TimedAnimation` subclass that makes an animation by repeatedly calling
(4)                 a function *func*.
(4)                 .. note::
(8)                     You must store the created Animation in a variable that lives as long
(8)                     as the animation should run. Otherwise, the Animation object will be
(8)                     garbage-collected and the animation stops.
(4)                 Parameters
(4)                 ----------
(4)                 fig : `~matplotlib.figure.Figure`
(8)                     The figure object used to get needed events, such as draw or resize.
(4)                 func : callable
(8)                     The function to call at each frame.  The first argument will
(8)                     be the next value in *frames*.   Any additional positional
(8)                     arguments can be supplied using `functools.partial` or via the *fargs*
(8)                     parameter.
(8)                     The required signature is::
```

```
            def func(frame, *fargs) -> iterable_of_artists
        It is often more convenient to provide the arguments using
        `functools.partial`. In this way it is also possible to pass keyword
        arguments. To pass a function with both positional and keyword
        arguments, set all arguments as keyword arguments, just leaving the
        *frame* argument unset::
            def func(frame, art, *, y=None):
                ...
            ani = FuncAnimation(fig, partial(func, art=ln, y='foo'))
        If ``blit == True``, *func* must return an iterable of all artists
        that were modified or created. This information is used by the blitting

        algorithm to determine which parts of the figure have to be updated.
        The return value is unused if ``blit == False`` and may be omitted in
        that case.
    frames : iterable, int, generator function, or None, optional
        Source of data to pass *func* and each frame of the animation
        - If an iterable, then simply use the values provided.  If the
          iterable has a length, it will override the *save_count* kwarg.
        - If an integer, then equivalent to passing ``range(frames)``
        - If a generator function, then must have the signature::
              def gen_function() -> obj
        - If *None*, then equivalent to passing ``itertools.count``.
        In all of these cases, the values in *frames* is simply passed through
        to the user-supplied *func* and thus can be of any type.
    init_func : callable, optional
        A function used to draw a clear frame. If not given, the results of
        drawing from the first item in the frames sequence will be used. This
        function will be called once before the first frame.
        The required signature is::
            def init_func() -> iterable_of_artists
        If ``blit == True``, *init_func* must return an iterable of artists
        to be re-drawn. This information is used by the blitting algorithm to
        determine which parts of the figure have to be updated.  The return
        value is unused if ``blit == False`` and may be omitted in that case.
    fargs : tuple or None, optional
        Additional arguments to pass to each call to *func*. Note: the use of
        `functools.partial` is preferred over *fargs*. See *func* for details.
    save_count : int, optional
        Fallback for the number of values from *frames* to cache. This is
        only used if the number of frames cannot be inferred from *frames*,
        i.e. when it's an iterator without length or a generator.
    interval : int, default: 200
        Delay between frames in milliseconds.
    repeat_delay : int, default: 0
        The delay in milliseconds between consecutive animation runs, if
        *repeat* is True.
    repeat : bool, default: True
        Whether the animation repeats when the sequence of frames is completed.
    blit : bool, default: False
        Whether blitting is used to optimize drawing.  Note: when using
        blitting, any animated artists will be drawn according to their zorder;

        however, they will be drawn on top of any previous artists, regardless
        of their zorder.
    cache_frame_data : bool, default: True
        Whether frame data is cached.  Disabling cache might be helpful when
        frames contain large objects.
    """
    def __init__(self, fig, func, frames=None, init_func=None, fargs=None,
                 save_count=None, *, cache_frame_data=True, **kwargs):
        if fargs:
            self._args = fargs
        else:
            self._args = ()
        self._func = func
        self._init_func = init_func
        self._save_count = save_count
```

```
(8)                 if frames is None:
(12)                    self._iter_gen = itertools.count
(8)                 elif callable(frames):
(12)                    self._iter_gen = frames
(8)                 elif np.iterable(frames):
(12)                    if kwargs.get('repeat', True):
(16)                        self._tee_from = frames
(16)                        def iter_frames(frames=frames):
(20)                            this, self._tee_from = itertools.tee(self._tee_from, 2)
(20)                            yield from this
(16)                        self._iter_gen = iter_frames
(12)                    else:
(16)                        self._iter_gen = lambda: iter(frames)
(12)                    if hasattr(frames, '__len__'):
(16)                        self._save_count = len(frames)
(16)                        if save_count is not None:
(20)                            _api.warn_external(
(24)                                f"You passed in an explicit {save_count=} "
(24)                                "which is being ignored in favor of "
(24)                                f"{len(frames)=}."
(20)                            )
(8)                 else:
(12)                    self._iter_gen = lambda: iter(range(frames))
(12)                    self._save_count = frames
(12)                    if save_count is not None:
(16)                        _api.warn_external(
(20)                            f"You passed in an explicit {save_count=} which is being "
(20)                            f"ignored in favor of {frames=}."
(16)                        )
(8)                 if self._save_count is None and cache_frame_data:
(12)                    _api.warn_external(
(16)                        f"{frames=!r} which we can infer the length of, "
(16)                        "did not pass an explicit *save_count* "
(16)                        f"and passed {cache_frame_data=}.  To avoid a possibly "
(16)                        "unbounded cache, frame data caching has been disabled. "
(16)                        "To suppress this warning either pass "
(16)                        "`cache_frame_data=False` or `save_count=MAX_FRAMES`."
(12)                    )
(12)                    cache_frame_data = False
(8)                 self._cache_frame_data = cache_frame_data
(8)                 self._save_seq = []
(8)                 super().__init__(fig, **kwargs)
(8)                 self._save_seq = []
(4)             def new_frame_seq(self):
(8)                 return self._iter_gen()
(4)             def new_saved_frame_seq(self):
(8)                 if self._save_seq:
(12)                    self._old_saved_seq = list(self._save_seq)
(12)                    return iter(self._old_saved_seq)
(8)                 else:
(12)                    if self._save_count is None:
(16)                        frame_seq = self.new_frame_seq()
(16)                        def gen():
(20)                            try:
(24)                                while True:
(28)                                    yield next(frame_seq)
(20)                            except StopIteration:
(24)                                pass
(16)                        return gen()
(12)                    else:
(16)                        return itertools.islice(self.new_frame_seq(),
self._save_count)
(4)             def _init_draw(self):
(8)                 super()._init_draw()
(8)                 if self._init_func is None:
(12)                    try:
(16)                        frame_data = next(self.new_frame_seq())
(12)                    except StopIteration:
(16)                        warnings.warn(
```

```
1228: (20)                                "Can not start iterating the frames for the initial draw.
"
1229: (20)                                "This can be caused by passing in a 0 length sequence "
1230: (20)                                "for *frames*.\n\n"
1231: (20)                                "If you passed *frames* as a generator "
1232: (20)                                "it may be exhausted due to a previous display or save."
1233: (16)                            )
1234: (16)                            return
1235: (12)                        self._draw_frame(frame_data)
1236: (8)                    else:
1237: (12)                        self._drawn_artists = self._init_func()
1238: (12)                        if self._blit:
1239: (16)                            if self._drawn_artists is None:
1240: (20)                                raise RuntimeError('The init_func must return a '
1241: (39)                                                   'sequence of Artist objects.')
1242: (16)                            for a in self._drawn_artists:
1243: (20)                                a.set_animated(self._blit)
1244: (8)                    self._save_seq = []
1245: (4)                def _draw_frame(self, framedata):
1246: (8)                    if self._cache_frame_data:
1247: (12)                        self._save_seq.append(framedata)
1248: (12)                        self._save_seq = self._save_seq[-self._save_count:]
1249: (8)                    self._drawn_artists = self._func(framedata, *self._args)
1250: (8)                    if self._blit:
1251: (12)                        err = RuntimeError('The animation function must return a sequence
'
1252: (31)                                           'of Artist objects.')
1253: (12)                        try:
1254: (16)                            iter(self._drawn_artists)
1255: (12)                        except TypeError:
1256: (16)                            raise err from None
1257: (12)                        for i in self._drawn_artists:
1258: (16)                            if not isinstance(i, mpl.artist.Artist):
1259: (20)                                raise err
1260: (12)                        self._drawn_artists = sorted(self._drawn_artists,
1261: (41)                                                     key=lambda x: x.get_zorder())
1262: (12)                        for a in self._drawn_artists:
1263: (16)                            a.set_animated(self._blit)
1264: (0)            def _validate_grabframe_kwargs(savefig_kwargs):
1265: (4)                if mpl.rcParams['savefig.bbox'] == 'tight':
1266: (8)                    raise ValueError(
1267: (12)                        f"{mpl.rcParams['savefig.bbox']=} must not be 'tight' as it "
1268: (12)                        "may cause frame size to vary, which is inappropriate for
animation."
1269: (8)                    )
1270: (4)                for k in ('dpi', 'bbox_inches', 'format'):
1271: (8)                    if k in savefig_kwargs:
1272: (12)                        raise TypeError(
1273: (16)                            f"grab_frame got an unexpected keyword argument {k!r}"
1274: (12)                        )
```

File 22 - artist.py:

```
1: (0)              from collections import namedtuple
2: (0)              import contextlib
3: (0)              from functools import cache, wraps
4: (0)              import inspect
5: (0)              from inspect import Signature, Parameter
6: (0)              import logging
7: (0)              from numbers import Number, Real
8: (0)              import re
9: (0)              import warnings
10: (0)             import numpy as np
11: (0)             import matplotlib as mpl
12: (0)             from . import _api, cbook
13: (0)             from .colors import BoundaryNorm
14: (0)             from .cm import ScalarMappable
```

```
(0)             from .path import Path
(0)             from .transforms import (BboxBase, Bbox, IdentityTransform, Transform,
TransformedBbox,
(25)                                     TransformedPatchPath, TransformedPath)
(0)             _log = logging.getLogger(__name__)
(0)             def _prevent_rasterization(draw):
(4)                 @wraps(draw)
(4)                 def draw_wrapper(artist, renderer, *args, **kwargs):
(8)                     if renderer._raster_depth == 0 and renderer._rasterizing:
(12)                        renderer.stop_rasterizing()
(12)                        renderer._rasterizing = False
(8)                     return draw(artist, renderer, *args, **kwargs)
(4)                 draw_wrapper._supports_rasterization = False
(4)                 return draw_wrapper
(0)             def allow_rasterization(draw):
(4)                 """
(4)                 Decorator for Artist.draw method. Provides routines
(4)                 that run before and after the draw call. The before and after functions
(4)                 are useful for changing artist-dependent renderer attributes or making
(4)                 other setup function calls, such as starting and flushing a mixed-mode
(4)                 renderer.
(4)                 """
(4)                 @wraps(draw)
(4)                 def draw_wrapper(artist, renderer):
(8)                     try:
(12)                        if artist.get_rasterized():
(16)                            if renderer._raster_depth == 0 and not renderer._rasterizing:
(20)                                renderer.start_rasterizing()
(20)                                renderer._rasterizing = True
(16)                            renderer._raster_depth += 1
(12)                        else:
(16)                            if renderer._raster_depth == 0 and renderer._rasterizing:
(20)                                renderer.stop_rasterizing()
(20)                                renderer._rasterizing = False
(12)                        if artist.get_agg_filter() is not None:
(16)                            renderer.start_filter()
(12)                        return draw(artist, renderer)
(8)                     finally:
(12)                        if artist.get_agg_filter() is not None:
(16)                            renderer.stop_filter(artist.get_agg_filter())
(12)                        if artist.get_rasterized():
(16)                            renderer._raster_depth -= 1
(12)                        if (renderer._rasterizing and artist.figure and
(20)                                artist.figure.suppressComposite):
(16)                            renderer.stop_rasterizing()
(16)                            renderer.start_rasterizing()
(4)                 draw_wrapper._supports_rasterization = True
(4)                 return draw_wrapper
(0)             def _finalize_rasterization(draw):
(4)                 """
(4)                 Decorator for Artist.draw method. Needed on the outermost artist, i.e.
(4)                 Figure, to finish up if the render is still in rasterized mode.
(4)                 """
(4)                 @wraps(draw)
(4)                 def draw_wrapper(artist, renderer, *args, **kwargs):
(8)                     result = draw(artist, renderer, *args, **kwargs)
(8)                     if renderer._rasterizing:
(12)                        renderer.stop_rasterizing()
(12)                        renderer._rasterizing = False
(8)                     return result
(4)                 return draw_wrapper
(0)             def _stale_axes_callback(self, val):
(4)                 if self.axes:
(8)                     self.axes.stale = val
(0)             _XYPair = namedtuple("_XYPair", "x y")
(0)             class _Unset:
(4)                 def __repr__(self):
(8)                     return "<UNSET>"
(0)             _UNSET = _Unset()
```

```
83: (0)             class Artist:
84: (4)                 """
85: (4)                 Abstract base class for objects that render into a FigureCanvas.
86: (4)                 Typically, all visible elements in a figure are subclasses of Artist.
87: (4)                 """
88: (4)                 zorder = 0
89: (4)                 def __init_subclass__(cls):
90: (8)                     if not hasattr(cls.draw, "_supports_rasterization"):
91: (12)                        cls.draw = _prevent_rasterization(cls.draw)
92: (8)                     if not hasattr(cls.set, '_autogenerated_signature'):
93: (12)                        return
94: (8)                     cls.set = lambda self, **kwargs: Artist.set(self, **kwargs)
95: (8)                     cls.set.__name__ = "set"
96: (8)                     cls.set.__qualname__ = f"{cls.__qualname__}.set"
97: (8)                     cls._update_set_signature_and_docstring()
98: (4)                 _PROPERTIES_EXCLUDED_FROM_SET = [
99: (8)                     'navigate_mode',  # not a user-facing function
100: (8)                    'figure',         # changing the figure is such a profound operation
101: (8)                    '3d_properties',  # cannot be used as a keyword due to leading digit
102: (4)                ]
103: (4)                @classmethod
104: (4)                def _update_set_signature_and_docstring(cls):
105: (8)                    """
106: (8)                    Update the signature of the set function to list all properties
107: (8)                    as keyword arguments.
108: (8)                    Property aliases are not listed in the signature for brevity, but
109: (8)                    are still accepted as keyword arguments.
110: (8)                    """
111: (8)                    cls.set.__signature__ = Signature(
112: (12)                       [Parameter("self", Parameter.POSITIONAL_OR_KEYWORD),
113: (13)                        *[Parameter(prop, Parameter.KEYWORD_ONLY, default=_UNSET)
114: (15)                          for prop in ArtistInspector(cls).get_setters()
115: (15)                          if prop not in Artist._PROPERTIES_EXCLUDED_FROM_SET]])
116: (8)                    cls.set._autogenerated_signature = True
117: (8)                    cls.set.__doc__ = (
118: (12)                       "Set multiple properties at once.\n\n"
119: (12)                       "Supported properties are\n\n"
120: (12)                       + kwdoc(cls))
121: (4)                def __init__(self):
122: (8)                    self._stale = True
123: (8)                    self.stale_callback = None
124: (8)                    self._axes = None
125: (8)                    self.figure = None
126: (8)                    self._transform = None
127: (8)                    self._transformSet = False
128: (8)                    self._visible = True
129: (8)                    self._animated = False
130: (8)                    self._alpha = None
131: (8)                    self.clipbox = None
132: (8)                    self._clippath = None
133: (8)                    self._clipon = True
134: (8)                    self._label = ''
135: (8)                    self._picker = None
136: (8)                    self._rasterized = False
137: (8)                    self._agg_filter = None
138: (8)                    self._mouseover = type(self).get_cursor_data != Artist.get_cursor_data
139: (8)                    self._callbacks = cbook.CallbackRegistry(signals=["pchanged"])
140: (8)                    try:
141: (12)                       self.axes = None
142: (8)                    except AttributeError:
143: (12)                       pass
144: (8)                    self._remove_method = None
145: (8)                    self._url = None
146: (8)                    self._gid = None
147: (8)                    self._snap = None
148: (8)                    self._sketch = mpl.rcParams['path.sketch']
149: (8)                    self._path_effects = mpl.rcParams['path.effects']
150: (8)                    self._sticky_edges = _XYPair([], [])
151: (8)                    self._in_layout = True
```

```
(4)     def __getstate__(self):
(8)         d = self.__dict__.copy()
(8)         d['stale_callback'] = None
(8)         return d
(4)     def remove(self):
(8)         """
(8)         Remove the artist from the figure if possible.
(8)         The effect will not be visible until the figure is redrawn, e.g.,
(8)         with `.FigureCanvasBase.draw_idle`.  Call `~.axes.Axes.relim` to
(8)         update the Axes limits if desired.
(8)         Note: `~.axes.Axes.relim` will not see collections even if the
(8)         collection was added to the Axes with *autolim* = True.
(8)         Note: there is no support for removing the artist's legend entry.
(8)         """
(8)         if self._remove_method is not None:
(12)            self._remove_method(self)
(12)            self.stale_callback = None
(12)            _ax_flag = False
(12)            if hasattr(self, 'axes') and self.axes:
(16)                self.axes._mouseover_set.discard(self)
(16)                self.axes.stale = True
(16)                self.axes = None  # decouple the artist from the Axes
(16)                _ax_flag = True
(12)            if self.figure:
(16)                if not _ax_flag:
(20)                    self.figure.stale = True
(16)                self.figure = None
(8)         else:
(12)            raise NotImplementedError('cannot remove artist')
(4)     def have_units(self):
(8)         """Return whether units are set on any axis."""
(8)         ax = self.axes
(8)         return ax and any(axis.have_units() for axis in ax._axis_map.values())
(4)     def convert_xunits(self, x):
(8)         """
(8)         Convert *x* using the unit type of the xaxis.
(8)         If the artist is not contained in an Axes or if the xaxis does not
(8)         have units, *x* itself is returned.
(8)         """
(8)         ax = getattr(self, 'axes', None)
(8)         if ax is None or ax.xaxis is None:
(12)            return x
(8)         return ax.xaxis.convert_units(x)
(4)     def convert_yunits(self, y):
(8)         """
(8)         Convert *y* using the unit type of the yaxis.
(8)         If the artist is not contained in an Axes or if the yaxis does not
(8)         have units, *y* itself is returned.
(8)         """
(8)         ax = getattr(self, 'axes', None)
(8)         if ax is None or ax.yaxis is None:
(12)            return y
(8)         return ax.yaxis.convert_units(y)
(4)     @property
(4)     def axes(self):
(8)         """The `~.axes.Axes` instance the artist resides in, or *None*."""
(8)         return self._axes
(4)     @axes.setter
(4)     def axes(self, new_axes):
(8)         if (new_axes is not None and self._axes is not None
(16)                and new_axes != self._axes):
(12)            raise ValueError("Can not reset the Axes. You are probably trying
to reuse "
(29)                                 "an artist in more than one Axes which is not
supported")
(8)         self._axes = new_axes
(8)         if new_axes is not None and new_axes is not self:
(12)            self.stale_callback = _stale_axes_callback
(4)     @property
```

```
    def stale(self):
        """
        Whether the artist is 'stale' and needs to be re-drawn for the output
        to match the internal state of the artist.
        """
        return self._stale
    @stale.setter
    def stale(self, val):
        self._stale = val
        if self._animated:
            return
        if val and self.stale_callback is not None:
            self.stale_callback(self, val)
    def get_window_extent(self, renderer=None):
        """
        Get the artist's bounding box in display space.
        The bounding box' width and height are nonnegative.
        Subclasses should override for inclusion in the bounding box
        "tight" calculation. Default is to return an empty bounding
        box at 0, 0.
        Be careful when using this function, the results will not update
        if the artist window extent of the artist changes.  The extent
        can change due to any changes in the transform stack, such as
        changing the Axes limits, the figure size, or the canvas used
        (as is done when saving a figure).  This can lead to unexpected
        behavior where interactive figures will look fine on the screen,
        but will save incorrectly.
        """
        return Bbox([[0, 0], [0, 0]])
    def get_tightbbox(self, renderer=None):
        """
        Like `.Artist.get_window_extent`, but includes any clipping.
        Parameters
        ----------
        renderer : `~matplotlib.backend_bases.RendererBase` subclass, optional
            renderer that will be used to draw the figures (i.e.
            ``fig.canvas.get_renderer()``)
        Returns
        -------
        `.Bbox` or None
            The enclosing bounding box (in figure pixel coordinates).
            Returns None if clipping results in no intersection.
        """
        bbox = self.get_window_extent(renderer)
        if self.get_clip_on():
            clip_box = self.get_clip_box()
            if clip_box is not None:
                bbox = Bbox.intersection(bbox, clip_box)
            clip_path = self.get_clip_path()
            if clip_path is not None and bbox is not None:
                clip_path = clip_path.get_fully_transformed_path()
                bbox = Bbox.intersection(bbox, clip_path.get_extents())
        return bbox
    def add_callback(self, func):
        """
        Add a callback function that will be called whenever one of the
        `.Artist`'s properties changes.
        Parameters
        ----------
        func : callable
            The callback function. It must have the signature::
                def func(artist: Artist) -> Any
            where *artist* is the calling `.Artist`. Return values may exist
            but are ignored.
        Returns
        -------
        int
            The observer id associated with the callback. This id can be
            used for removing the callback with `.remove_callback` later.
```

```
288: (8)                    See Also
289: (8)                    --------
290: (8)                    remove_callback
291: (8)                    """
292: (8)                    return self._callbacks.connect("pchanged", lambda: func(self))
293: (4)                def remove_callback(self, oid):
294: (8)                    """
295: (8)                    Remove a callback based on its observer id.
296: (8)                    See Also
297: (8)                    --------
298: (8)                    add_callback
299: (8)                    """
300: (8)                    self._callbacks.disconnect(oid)
301: (4)                def pchanged(self):
302: (8)                    """
303: (8)                    Call all of the registered callbacks.
304: (8)                    This function is triggered internally when a property is changed.
305: (8)                    See Also
306: (8)                    --------
307: (8)                    add_callback
308: (8)                    remove_callback
309: (8)                    """
310: (8)                    self._callbacks.process("pchanged")
311: (4)                def is_transform_set(self):
312: (8)                    """
313: (8)                    Return whether the Artist has an explicitly set transform.
314: (8)                    This is *True* after `.set_transform` has been called.
315: (8)                    """
316: (8)                    return self._transformSet
317: (4)                def set_transform(self, t):
318: (8)                    """
319: (8)                    Set the artist transform.
320: (8)                    Parameters
321: (8)                    ----------
322: (8)                    t : `~matplotlib.transforms.Transform`
323: (8)                    """
324: (8)                    self._transform = t
325: (8)                    self._transformSet = True
326: (8)                    self.pchanged()
327: (8)                    self.stale = True
328: (4)                def get_transform(self):
329: (8)                    """Return the `.Transform` instance used by this artist."""
330: (8)                    if self._transform is None:
331: (12)                       self._transform = IdentityTransform()
332: (8)                    elif (not isinstance(self._transform, Transform)
333: (14)                         and hasattr(self._transform, '_as_mpl_transform')):
334: (12)                       self._transform = self._transform._as_mpl_transform(self.axes)
335: (8)                    return self._transform
336: (4)                def get_children(self):
337: (8)                    r"""Return a list of the child `.Artist`\s of this `.Artist`."""
338: (8)                    return []
339: (4)                def _different_canvas(self, event):
340: (8)                    """
341: (8)                    Check whether an *event* occurred on a canvas other that this artist's
canvas.
342: (8)                    If this method returns True, the event definitely occurred on a
different
343: (8)                    canvas; if it returns False, either it occurred on the same canvas, or
we may
344: (8)                    not have enough information to know.
345: (8)                    Subclasses should start their definition of `contains` as follows::
346: (12)                       if self._different_canvas(mouseevent):
347: (16)                           return False, {}
348: (8)                    """
349: (8)                    return (getattr(event, "canvas", None) is not None and self.figure is
not None
350: (16)                           and event.canvas is not self.figure.canvas)
351: (4)                def contains(self, mouseevent):
352: (8)                    """
```

```
(8)                     Test whether the artist contains the mouse event.
(8)                     Parameters
(8)                     ----------
(8)                     mouseevent : `~matplotlib.backend_bases.MouseEvent`
(8)                     Returns
(8)                     -------
(8)                     contains : bool
(12)                        Whether any values are within the radius.
(8)                     details : dict
(12)                        An artist-specific dictionary of details of the event context,
(12)                        such as which points are contained in the pick radius. See the
(12)                        individual Artist subclasses for details.
(8)                     """
(8)                     _log.warning("%r needs 'contains' method", self.__class__.__name__)
(8)                     return False, {}
(4)                 def pickable(self):
(8)                     """
(8)                     Return whether the artist is pickable.
(8)                     See Also
(8)                     --------
(8)                     .Artist.set_picker, .Artist.get_picker, .Artist.pick
(8)                     """
(8)                     return self.figure is not None and self._picker is not None
(4)                 def pick(self, mouseevent):
(8)                     """
(8)                     Process a pick event.
(8)                     Each child artist will fire a pick event if *mouseevent* is over
(8)                     the artist and the artist has picker set.
(8)                     See Also
(8)                     --------
(8)                     .Artist.set_picker, .Artist.get_picker, .Artist.pickable
(8)                     """
(8)                     from .backend_bases import PickEvent  # Circular import.
(8)                     if self.pickable():
(12)                        picker = self.get_picker()
(12)                        if callable(picker):
(16)                            inside, prop = picker(self, mouseevent)
(12)                        else:
(16)                            inside, prop = self.contains(mouseevent)
(12)                        if inside:
(16)                            PickEvent("pick_event", self.figure.canvas,
(26)                                      mouseevent, self, **prop)._process()
(8)                     for a in self.get_children():
(12)                        ax = getattr(a, 'axes', None)
(12)                        if (isinstance(a, mpl.figure.SubFigure)
(20)                                or mouseevent.inaxes is None or ax is None
(20)                                or mouseevent.inaxes == ax):
(16)                            a.pick(mouseevent)
(4)                 def set_picker(self, picker):
(8)                     """
(8)                     Define the picking behavior of the artist.
(8)                     Parameters
(8)                     ----------
(8)                     picker : None or bool or float or callable
(12)                        This can be one of the following:
(12)                        - *None*: Picking is disabled for this artist (default).
(12)                        - A boolean: If *True* then picking will be enabled and the
(14)                          artist will fire a pick event if the mouse event is over
(14)                          the artist.
(12)                        - A float: If picker is a number it is interpreted as an
(14)                          epsilon tolerance in points and the artist will fire
(14)                          off an event if its data is within epsilon of the mouse
(14)                          event.  For some artists like lines and patch collections,
(14)                          the artist may provide additional data to the pick event
(14)                          that is generated, e.g., the indices of the data within
(14)                          epsilon of the pick event
(12)                        - A function: If picker is callable, it is a user supplied
(14)                          function which determines whether the artist is hit by the
(14)                          mouse event::
```

```
(16)                            hit, props = picker(artist, mouseevent)
(14)                          to determine the hit test.  if the mouse event is over the
(14)                          artist, return *hit=True* and props is a dictionary of
(14)                          properties you want added to the PickEvent attributes.
(8)                     """
(8)                     self._picker = picker
(4)                 def get_picker(self):
(8)                     """
(8)                     Return the picking behavior of the artist.
(8)                     The possible values are described in `.Artist.set_picker`.
(8)                     See Also
(8)                     --------
(8)                     .Artist.set_picker, .Artist.pickable, .Artist.pick
(8)                     """
(8)                     return self._picker
(4)                 def get_url(self):
(8)                     """Return the url."""
(8)                     return self._url
(4)                 def set_url(self, url):
(8)                     """
(8)                     Set the url for the artist.
(8)                     Parameters
(8)                     ----------
(8)                     url : str
(8)                     """
(8)                     self._url = url
(4)                 def get_gid(self):
(8)                     """Return the group id."""
(8)                     return self._gid
(4)                 def set_gid(self, gid):
(8)                     """
(8)                     Set the (group) id for the artist.
(8)                     Parameters
(8)                     ----------
(8)                     gid : str
(8)                     """
(8)                     self._gid = gid
(4)                 def get_snap(self):
(8)                     """
(8)                     Return the snap setting.
(8)                     See `.set_snap` for details.
(8)                     """
(8)                     if mpl.rcParams['path.snap']:
(12)                        return self._snap
(8)                     else:
(12)                        return False
(4)                 def set_snap(self, snap):
(8)                     """
(8)                     Set the snapping behavior.
(8)                     Snapping aligns positions with the pixel grid, which results in
(8)                     clearer images. For example, if a black line of 1px width was
(8)                     defined at a position in between two pixels, the resulting image
(8)                     would contain the interpolated value of that line in the pixel grid,
(8)                     which would be a grey value on both adjacent pixel positions. In
(8)                     contrast, snapping will move the line to the nearest integer pixel
(8)                     value, so that the resulting image will really contain a 1px wide
(8)                     black line.
(8)                     Snapping is currently only supported by the Agg and MacOSX backends.
(8)                     Parameters
(8)                     ----------
(8)                     snap : bool or None
(12)                        Possible values:
(12)                        - *True*: Snap vertices to the nearest pixel center.
(12)                        - *False*: Do not modify vertex positions.
(12)                        - *None*: (auto) If the path contains only rectilinear line
(14)                          segments, round to the nearest pixel center.
(8)                     """
(8)                     self._snap = snap
(8)                     self.stale = True
```

```
    def get_sketch_params(self):
        """
        Return the sketch parameters for the artist.
        Returns
        -------
        tuple or None
            A 3-tuple with the following elements:
            - *scale*: The amplitude of the wiggle perpendicular to the
              source line.
            - *length*: The length of the wiggle along the line.
            - *randomness*: The scale factor by which the length is
              shrunken or expanded.
            Returns *None* if no sketch parameters were set.
        """
        return self._sketch
    def set_sketch_params(self, scale=None, length=None, randomness=None):
        """
        Set the sketch parameters.
        Parameters
        ----------
        scale : float, optional
            The amplitude of the wiggle perpendicular to the source
            line, in pixels.  If scale is `None`, or not provided, no
            sketch filter will be provided.
        length : float, optional
             The length of the wiggle along the line, in pixels
             (default 128.0)
        randomness : float, optional
            The scale factor by which the length is shrunken or
            expanded (default 16.0)
            The PGF backend uses this argument as an RNG seed and not as
            described above. Using the same seed yields the same random shape.
            .. ACCEPTS: (scale: float, length: float, randomness: float)
        """
        if scale is None:
            self._sketch = None
        else:
            self._sketch = (scale, length or 128.0, randomness or 16.0)
        self.stale = True
    def set_path_effects(self, path_effects):
        """
        Set the path effects.
        Parameters
        ----------
        path_effects : list of `.AbstractPathEffect`
        """
        self._path_effects = path_effects
        self.stale = True
    def get_path_effects(self):
        return self._path_effects
    def get_figure(self):
        """Return the `.Figure` instance the artist belongs to."""
        return self.figure
    def set_figure(self, fig):
        """
        Set the `.Figure` instance the artist belongs to.
        Parameters
        ----------
        fig : `~matplotlib.figure.Figure`
        """
        if self.figure is fig:
            return
        if self.figure is not None:
            raise RuntimeError("Can not put single artist in "
                               "more than one figure")
        self.figure = fig
        if self.figure and self.figure is not self:
            self.pchanged()
        self.stale = True
```

```
    def set_clip_box(self, clipbox):
        """
        Set the artist's clip `.Bbox`.
        Parameters
        ----------
        clipbox : `~matplotlib.transforms.BboxBase` or None
            Will typically be created from a `.TransformedBbox`. For instance,
            ``TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)`` is the
default

            clipping for an artist added to an Axes.
        """
        _api.check_isinstance((BboxBase, None), clipbox=clipbox)
        if clipbox != self.clipbox:
            self.clipbox = clipbox
            self.pchanged()
            self.stale = True
    def set_clip_path(self, path, transform=None):
        """
        Set the artist's clip path.
        Parameters
        ----------
        path : `~matplotlib.patches.Patch` or `.Path` or `.TransformedPath` or
None

            The clip path. If given a `.Path`, *transform* must be provided as
            well. If *None*, a previously set clip path is removed.
        transform : `~matplotlib.transforms.Transform`, optional
            Only used if *path* is a `.Path`, in which case the given `.Path`
            is converted to a `.TransformedPath` using *transform*.
        Notes
        -----
        For efficiency, if *path* is a `.Rectangle` this method will set the
        clipping box to the corresponding rectangle and set the clipping path
        to ``None``.
        For technical reasons (support of `~.Artist.set`), a tuple
        (*path*, *transform*) is also accepted as a single positional
        parameter.
        .. ACCEPTS: Patch or (Path, Transform) or None
        """
        from matplotlib.patches import Patch, Rectangle
        success = False
        if transform is None:
            if isinstance(path, Rectangle):
                self.clipbox = TransformedBbox(Bbox.unit(),
                                               path.get_transform())
                self._clippath = None
                success = True
            elif isinstance(path, Patch):
                self._clippath = TransformedPatchPath(path)
                success = True
            elif isinstance(path, tuple):
                path, transform = path
        if path is None:
            self._clippath = None
            success = True
        elif isinstance(path, Path):
            self._clippath = TransformedPath(path, transform)
            success = True
        elif isinstance(path, TransformedPatchPath):
            self._clippath = path
            success = True
        elif isinstance(path, TransformedPath):
            self._clippath = path
            success = True
        if not success:
            raise TypeError(
                "Invalid arguments to set_clip_path, of type "
                f"{type(path).__name__} and {type(transform).__name__}")
        self.pchanged()
        self.stale = True
```

```
    def get_alpha(self):
        """
        Return the alpha value used for blending - not supported on all
        backends.
        """
        return self._alpha
    def get_visible(self):
        """Return the visibility."""
        return self._visible
    def get_animated(self):
        """Return whether the artist is animated."""
        return self._animated
    def get_in_layout(self):
        """
        Return boolean flag, ``True`` if artist is included in layout
        calculations.
        E.g. :ref:`constrainedlayout_guide`,
        `.Figure.tight_layout()`, and
        ``fig.savefig(fname, bbox_inches='tight')``.
        """
        return self._in_layout
    def _fully_clipped_to_axes(self):
        """
        Return a boolean flag, ``True`` if the artist is clipped to the Axes
        and can thus be skipped in layout calculations. Requires `get_clip_on`
        is True, one of `clip_box` or `clip_path` is set, ``clip_box.extents``
        is equivalent to ``ax.bbox.extents`` (if set), and
``clip_path._patch``
        is equivalent to ``ax.patch`` (if set).
        """
        clip_box = self.get_clip_box()
        clip_path = self.get_clip_path()
        return (self.axes is not None
                and self.get_clip_on()
                and (clip_box is not None or clip_path is not None)
                and (clip_box is None
                     or np.all(clip_box.extents == self.axes.bbox.extents))
                and (clip_path is None
                     or isinstance(clip_path, TransformedPatchPath)
                     and clip_path._patch is self.axes.patch))
    def get_clip_on(self):
        """Return whether the artist uses clipping."""
        return self._clipon
    def get_clip_box(self):
        """Return the clipbox."""
        return self.clipbox
    def get_clip_path(self):
        """Return the clip path."""
        return self._clippath
    def get_transformed_clip_path_and_affine(self):
        """
        Return the clip path with the non-affine part of its
        transformation applied, and the remaining affine part of its
        transformation.
        """
        if self._clippath is not None:
            return self._clippath.get_transformed_path_and_affine()
        return None, None
    def set_clip_on(self, b):
        """
        Set whether the artist uses clipping.
        When False, artists will be visible outside the Axes which
        can lead to unexpected results.
        Parameters
        ----------
        b : bool
        """
        self._clipon = b
        self.pchanged()
```

```
(8)                     self.stale = True
(4)                 def _set_gc_clip(self, gc):
(8)                     """Set the clip properly for the gc."""
(8)                     if self._clipon:
(12)                        if self.clipbox is not None:
(16)                            gc.set_clip_rectangle(self.clipbox)
(12)                        gc.set_clip_path(self._clippath)
(8)                     else:
(12)                        gc.set_clip_rectangle(None)
(12)                        gc.set_clip_path(None)
(4)                 def get_rasterized(self):
(8)                     """Return whether the artist is to be rasterized."""
(8)                     return self._rasterized
(4)                 def set_rasterized(self, rasterized):
(8)                     """
(8)                     Force rasterized (bitmap) drawing for vector graphics output.
(8)                     Rasterized drawing is not supported by all artists. If you try to
(8)                     enable this on an artist that does not support it, the command has no
(8)                     effect and a warning will be issued.
(8)                     This setting is ignored for pixel-based output.
(8)                     See also :doc:`/gallery/misc/rasterization_demo`.
(8)                     Parameters
(8)                     ----------
(8)                     rasterized : bool
(8)                     """
(8)                     supports_rasterization = getattr(self.draw,
(41)                                                     "_supports_rasterization", False)
(8)                     if rasterized and not supports_rasterization:
(12)                        _api.warn_external(f"Rasterization of '{self}' will be ignored")
(8)                     self._rasterized = rasterized
(4)                 def get_agg_filter(self):
(8)                     """Return filter function to be used for agg filter."""
(8)                     return self._agg_filter
(4)                 def set_agg_filter(self, filter_func):
(8)                     """
(8)                     Set the agg filter.
(8)                     Parameters
(8)                     ----------
(8)                     filter_func : callable
(12)                        A filter function, which takes a (m, n, depth) float array
(12)                        and a dpi value, and returns a (m, n, depth) array and two
(12)                        offsets from the bottom left corner of the image
(12)                        .. ACCEPTS: a filter function, which takes a (m, n, 3) float array
(16)                            and a dpi value, and returns a (m, n, 3) array and two offsets
(16)                            from the bottom left corner of the image
(8)                     """
(8)                     self._agg_filter = filter_func
(8)                     self.stale = True
(4)                 def draw(self, renderer):
(8)                     """
(8)                     Draw the Artist (and its children) using the given renderer.
(8)                     This has no effect if the artist is not visible (`.Artist.get_visible`
(8)                     returns False).
(8)                     Parameters
(8)                     ----------
(8)                     renderer : `~matplotlib.backend_bases.RendererBase` subclass.
(8)                     Notes
(8)                     -----
(8)                     This method is overridden in the Artist subclasses.
(8)                     """
(8)                     if not self.get_visible():
(12)                        return
(8)                     self.stale = False
(4)                 def set_alpha(self, alpha):
(8)                     """
(8)                     Set the alpha value used for blending - not supported on all backends.
(8)                     Parameters
(8)                     ----------
(8)                     alpha : scalar or None
```

```
764: (12)                        *alpha* must be within the 0-1 range, inclusive.
765: (8)                    """
766: (8)                    if alpha is not None and not isinstance(alpha, Real):
767: (12)                        raise TypeError(
768: (16)                            f'alpha must be numeric or None, not {type(alpha)}')
769: (8)                    if alpha is not None and not (0 <= alpha <= 1):
770: (12)                        raise ValueError(f'alpha ({alpha}) is outside 0-1 range')
771: (8)                    if alpha != self._alpha:
772: (12)                        self._alpha = alpha
773: (12)                        self.pchanged()
774: (12)                        self.stale = True
775: (4)                def _set_alpha_for_array(self, alpha):
776: (8)                    """
777: (8)                    Set the alpha value used for blending - not supported on all backends.
778: (8)                    Parameters
779: (8)                    ----------
780: (8)                    alpha : array-like or scalar or None
781: (12)                        All values must be within the 0-1 range, inclusive.
782: (12)                        Masked values and nans are not supported.
783: (8)                    """
784: (8)                    if isinstance(alpha, str):
785: (12)                        raise TypeError("alpha must be numeric or None, not a string")
786: (8)                    if not np.iterable(alpha):
787: (12)                        Artist.set_alpha(self, alpha)
788: (12)                        return
789: (8)                    alpha = np.asarray(alpha)
790: (8)                    if not (0 <= alpha.min() and alpha.max() <= 1):
791: (12)                        raise ValueError('alpha must be between 0 and 1, inclusive, '
792: (29)                                         f'but min is {alpha.min()}, max is
{alpha.max()}')
793: (8)                    self._alpha = alpha
794: (8)                    self.pchanged()
795: (8)                    self.stale = True
796: (4)                def set_visible(self, b):
797: (8)                    """
798: (8)                    Set the artist's visibility.
799: (8)                    Parameters
800: (8)                    ----------
801: (8)                    b : bool
802: (8)                    """
803: (8)                    if b != self._visible:
804: (12)                        self._visible = b
805: (12)                        self.pchanged()
806: (12)                        self.stale = True
807: (4)                def set_animated(self, b):
808: (8)                    """
809: (8)                    Set whether the artist is intended to be used in an animation.
810: (8)                    If True, the artist is excluded from regular drawing of the figure.
811: (8)                    You have to call `.Figure.draw_artist` / `.Axes.draw_artist`
812: (8)                    explicitly on the artist. This approach is used to speed up animations
813: (8)                    using blitting.
814: (8)                    See also `matplotlib.animation` and
815: (8)                    :ref:`blitting`.
816: (8)                    Parameters
817: (8)                    ----------
818: (8)                    b : bool
819: (8)                    """
820: (8)                    if self._animated != b:
821: (12)                        self._animated = b
822: (12)                        self.pchanged()
823: (4)                def set_in_layout(self, in_layout):
824: (8)                    """
825: (8)                    Set if artist is to be included in layout calculations,
826: (8)                    E.g. :ref:`constrainedlayout_guide`,
827: (8)                    `.Figure.tight_layout()`, and
828: (8)                    ``fig.savefig(fname, bbox_inches='tight')``.
829: (8)                    Parameters
830: (8)                    ----------
831: (8)                    in_layout : bool
```

```
(8)                 """
(8)                 self._in_layout = in_layout
(4)             def get_label(self):
(8)                 """Return the label used for this artist in the legend."""
(8)                 return self._label
(4)             def set_label(self, s):
(8)                 """
(8)                 Set a label that will be displayed in the legend.
(8)                 Parameters
(8)                 ----------
(8)                 s : object
(12)                    *s* will be converted to a string by calling `str`.
(8)                 """
(8)                 label = str(s) if s is not None else None
(8)                 if label != self._label:
(12)                    self._label = label
(12)                    self.pchanged()
(12)                    self.stale = True
(4)             def get_zorder(self):
(8)                 """Return the artist's zorder."""
(8)                 return self.zorder
(4)             def set_zorder(self, level):
(8)                 """
(8)                 Set the zorder for the artist.  Artists with lower zorder
(8)                 values are drawn first.
(8)                 Parameters
(8)                 ----------
(8)                 level : float
(8)                 """
(8)                 if level is None:
(12)                    level = self.__class__.zorder
(8)                 if level != self.zorder:
(12)                    self.zorder = level
(12)                    self.pchanged()
(12)                    self.stale = True
(4)             @property
(4)             def sticky_edges(self):
(8)                 """
(8)                 ``x`` and ``y`` sticky edge lists for autoscaling.
(8)                 When performing autoscaling, if a data limit coincides with a value in
(8)                 the corresponding sticky_edges list, then no margin will be added--the
(8)                 view limit "sticks" to the edge. A typical use case is histograms,
(8)                 where one usually expects no margin on the bottom edge (0) of the
(8)                 histogram.
(8)                 Moreover, margin expansion "bumps" against sticky edges and cannot
(8)                 cross them.  For example, if the upper data limit is 1.0, the upper
(8)                 view limit computed by simple margin application is 1.2, but there is
a
(8)                 sticky edge at 1.1, then the actual upper view limit will be 1.1.
(8)                 This attribute cannot be assigned to; however, the ``x`` and ``y``
(8)                 lists can be modified in place as needed.
(8)                 Examples
(8)                 --------
(8)                 >>> artist.sticky_edges.x[:] = (xmin, xmax)
(8)                 >>> artist.sticky_edges.y[:] = (ymin, ymax)
(8)                 """
(8)                 return self._sticky_edges
(4)             def update_from(self, other):
(8)                 """Copy properties from *other* to *self*."""
(8)                 self._transform = other._transform
(8)                 self._transformSet = other._transformSet
(8)                 self._visible = other._visible
(8)                 self._alpha = other._alpha
(8)                 self.clipbox = other.clipbox
(8)                 self._clipon = other._clipon
(8)                 self._clippath = other._clippath
(8)                 self._label = other._label
(8)                 self._sketch = other._sketch
(8)                 self._path_effects = other._path_effects
```

```
(8)                 self.sticky_edges.x[:] = other.sticky_edges.x.copy()
(8)                 self.sticky_edges.y[:] = other.sticky_edges.y.copy()
(8)                 self.pchanged()
(8)                 self.stale = True
(4)             def properties(self):
(8)                 """Return a dictionary of all the properties of the artist."""
(8)                 return ArtistInspector(self).properties()
(4)             def _update_props(self, props, errfmt):
(8)                 """
(8)                 Helper for `.Artist.set` and `.Artist.update`.
(8)                 *errfmt* is used to generate error messages for invalid property
(8)                 names; it gets formatted with ``type(self)`` and the property name.
(8)                 """
(8)                 ret = []
(8)                 with cbook._setattr_cm(self, eventson=False):
(12)                    for k, v in props.items():
(16)                        if k == "axes":
(20)                            ret.append(setattr(self, k, v))
(16)                        else:
(20)                            func = getattr(self, f"set_{k}", None)
(20)                            if not callable(func):
(24)                                raise AttributeError(
(28)                                    errfmt.format(cls=type(self), prop_name=k))
(20)                            ret.append(func(v))
(8)                 if ret:
(12)                    self.pchanged()
(12)                    self.stale = True
(8)                 return ret
(4)             def update(self, props):
(8)                 """
(8)                 Update this artist's properties from the dict *props*.
(8)                 Parameters
(8)                 ----------
(8)                 props : dict
(8)                 """
(8)                 return self._update_props(
(12)                    props, "{cls.__name__!r} object has no property {prop_name!r}")
(4)             def _internal_update(self, kwargs):
(8)                 """
(8)                 Update artist properties without prenormalizing them, but generating
(8)                 errors as if calling `set`.
(8)                 The lack of prenormalization is to maintain backcompatibility.
(8)                 """
(8)                 return self._update_props(
(12)                    kwargs, "{cls.__name__}.set() got an unexpected keyword argument "
(12)                    "{prop_name!r}")
(4)             def set(self, **kwargs):
(8)                 return self._internal_update(cbook.normalize_kwargs(kwargs, self))
(4)             @contextlib.contextmanager
(4)             def _cm_set(self, **kwargs):
(8)                 """
(8)                 `.Artist.set` context-manager that restores original values at exit.
(8)                 """
(8)                 orig_vals = {k: getattr(self, f"get_{k}")() for k in kwargs}
(8)                 try:
(12)                    self.set(**kwargs)
(12)                    yield
(8)                 finally:
(12)                    self.set(**orig_vals)
(4)             def findobj(self, match=None, include_self=True):
(8)                 """
(8)                 Find artist objects.
(8)                 Recursively find all `.Artist` instances contained in the artist.
(8)                 Parameters
(8)                 ----------
(8)                 match
(12)                    A filter criterion for the matches. This can be
(12)                    - *None*: Return all objects contained in artist.
(12)                    - A function with signature ``def match(artist: Artist) -> bool``.
```

```
(14)                        The result will only contain artists for which the function
(14)                        returns *True*.
(12)                      - A class instance: e.g., `.Line2D`. The result will only contain
(14)                        artists of this class or its subclasses (``isinstance`` check).
(8)                   include_self : bool
(12)                      Include *self* in the list to be checked for a match.
(8)                   Returns
(8)                   -------
(8)                   list of `.Artist`
(8)                   """
(8)                   if match is None:  # always return True
(12)                      def matchfunc(x):
(16)                          return True
(8)                   elif isinstance(match, type) and issubclass(match, Artist):
(12)                      def matchfunc(x):
(16)                          return isinstance(x, match)
(8)                   elif callable(match):
(12)                      matchfunc = match
(8)                   else:
(12)                      raise ValueError('match must be None, a matplotlib.artist.Artist '
(29)                                       'subclass, or a callable')
(8)                   artists = sum([c.findobj(matchfunc) for c in self.get_children()], [])
(8)                   if include_self and matchfunc(self):
(12)                      artists.append(self)
(8)                   return artists
(4)               def get_cursor_data(self, event):
(8)                   """
(8)                   Return the cursor data for a given event.
(8)                   .. note::
(12)                      This method is intended to be overridden by artist subclasses.
(12)                      As an end-user of Matplotlib you will most likely not call this
(12)                      method yourself.
(8)                   Cursor data can be used by Artists to provide additional context
(8)                   information for a given event. The default implementation just returns
(8)                   *None*.
(8)                   Subclasses can override the method and return arbitrary data. However,
(8)                   when doing so, they must ensure that `.format_cursor_data` can convert
(8)                   the data to a string representation.
(8)                   The only current use case is displaying the z-value of an `.AxesImage`
(8)                   in the status bar of a plot window, while moving the mouse.
(8)                   Parameters
(8)                   ----------
(8)                   event : `~matplotlib.backend_bases.MouseEvent`
(8)                   See Also
(8)                   --------
(8)                   format_cursor_data
(8)                   """
(8)                   return None
(4)               def format_cursor_data(self, data):
(8)                   """
(8)                   Return a string representation of *data*.
(8)                   .. note::
(12)                      This method is intended to be overridden by artist subclasses.
(12)                      As an end-user of Matplotlib you will most likely not call this
(12)                      method yourself.
(8)                   The default implementation converts ints and floats and arrays of ints
(8)                   and floats into a comma-separated string enclosed in square brackets,
(8)                   unless the artist has an associated colorbar, in which case scalar
(8)                   values are formatted using the colorbar's formatter.
(8)                   See Also
(8)                   --------
(8)                   get_cursor_data
(8)                   """
(8)                   if np.ndim(data) == 0 and isinstance(self, ScalarMappable):
(12)                      n = self.cmap.N
(12)                      if np.ma.getmask(data):
(16)                          return "[]"
(12)                      normed = self.norm(data)
(12)                      if np.isfinite(normed):
```

```
(16)                        if isinstance(self.norm, BoundaryNorm):
(20)                            cur_idx = np.argmin(np.abs(self.norm.boundaries - data))
(20)                            neigh_idx = max(0, cur_idx - 1)
(20)                            delta = np.diff(
(24)                                self.norm.boundaries[neigh_idx:cur_idx + 2]
(20)                            ).max()
(16)                        elif self.norm.vmin == self.norm.vmax:
(20)                            delta = np.abs(self.norm.vmin * .1)
(16)                        else:
(20)                            neighbors = self.norm.inverse(
(24)                                (int(normed * n) + np.array([0, 1])) / n)
(20)                            delta = abs(neighbors - data).max()
(16)                        g_sig_digits = cbook._g_sig_digits(data, delta)
(12)                    else:
(16)                        g_sig_digits = 3  # Consistent with default below.
(12)                    return f"[{data:-#.{g_sig_digits}g}]"
(8)                 else:
(12)                    try:
(16)                        data[0]
(12)                    except (TypeError, IndexError):
(16)                        data = [data]
(12)                    data_str = ', '.join(f'{item:0.3g}' for item in data
(33)                                         if isinstance(item, Number))
(12)                    return "[" + data_str + "]"
(4)             def get_mouseover(self):
(8)                 """
(8)                 Return whether this artist is queried for custom context information
(8)                 when the mouse cursor moves over it.
(8)                 """
(8)                 return self._mouseover
(4)             def set_mouseover(self, mouseover):
(8)                 """
(8)                 Set whether this artist is queried for custom context information when
(8)                 the mouse cursor moves over it.
(8)                 Parameters
(8)                 ----------
(8)                 mouseover : bool
(8)                 See Also
(8)                 --------
(8)                 get_cursor_data
(8)                 .ToolCursorPosition
(8)                 .NavigationToolbar2
(8)                 """
(8)                 self._mouseover = bool(mouseover)
(8)                 ax = self.axes
(8)                 if ax:
(12)                    if self._mouseover:
(16)                        ax._mouseover_set.add(self)
(12)                    else:
(16)                        ax._mouseover_set.discard(self)
(4)             mouseover = property(get_mouseover, set_mouseover)  # backcompat.
(0)         def _get_tightbbox_for_layout_only(obj, *args, **kwargs):
(4)             """
(4)             Matplotlib's `.Axes.get_tightbbox` and `.Axis.get_tightbbox` support a
(4)             *for_layout_only* kwarg; this helper tries to use the kwarg but skips it
(4)             when encountering third-party subclasses that do not support it.
(4)             """
(4)             try:
(8)                 return obj.get_tightbbox(*args, **{**kwargs, "for_layout_only": True})
(4)             except TypeError:
(8)                 return obj.get_tightbbox(*args, **kwargs)
(0)         class ArtistInspector:
(4)             """
(4)             A helper class to inspect an `~matplotlib.artist.Artist` and return
(4)             information about its settable properties and their current values.
(4)             """
(4)             def __init__(self, o):
(8)                 r"""
(8)                 Initialize the artist inspector with an `Artist` or an iterable of
```

```
1107: (8)                   `Artist`\s.  If an iterable is used, we assume it is a homogeneous
1108: (8)                   sequence (all `Artist`\s are of the same type) and it is your
1109: (8)                   responsibility to make sure this is so.
1110: (8)                   """
1111: (8)                   if not isinstance(o, Artist):
1112: (12)                      if np.iterable(o):
1113: (16)                          o = list(o)
1114: (16)                          if len(o):
1115: (20)                              o = o[0]
1116: (8)                   self.oorig = o
1117: (8)                   if not isinstance(o, type):
1118: (12)                      o = type(o)
1119: (8)                   self.o = o
1120: (8)                   self.aliasd = self.get_aliases()
1121: (4)               def get_aliases(self):
1122: (8)                   """
1123: (8)                   Get a dict mapping property fullnames to sets of aliases for each
alias
1124: (8)                   in the :class:`~matplotlib.artist.ArtistInspector`.
1125: (8)                   e.g., for lines::
1126: (10)                    {'markerfacecolor': {'mfc'},
1127: (11)                     'linewidth'      : {'lw'},
1128: (10)                    }
1129: (8)                   """
1130: (8)                   names = [name for name in dir(self.o)
1131: (17)                           if name.startswith(('set_', 'get_'))
1132: (20)                              and callable(getattr(self.o, name))]
1133: (8)                   aliases = {}
1134: (8)                   for name in names:
1135: (12)                      func = getattr(self.o, name)
1136: (12)                      if not self.is_alias(func):
1137: (16)                          continue
1138: (12)                      propname = re.search(f"`({name[:4]}.*)`",  # get_.*/set_.*
1139: (33)                                           inspect.getdoc(func)).group(1)
1140: (12)                      aliases.setdefault(propname[4:], set()).add(name[4:])
1141: (8)                   return aliases
1142: (4)               _get_valid_values_regex = re.compile(
1143: (8)                   r"\n\s*(?:\.\.\s+)?ACCEPTS:\s*((?:.|\n)*?)(?:$|(?:\n\n))"
1144: (4)               )
1145: (4)               def get_valid_values(self, attr):
1146: (8)                   """
1147: (8)                   Get the legal arguments for the setter associated with *attr*.
1148: (8)                   This is done by querying the docstring of the setter for a line that
1149: (8)                   begins with "ACCEPTS:" or ".. ACCEPTS:", and then by looking for a
1150: (8)                   numpydoc-style documentation for the setter's first argument.
1151: (8)                   """
1152: (8)                   name = 'set_%s' % attr
1153: (8)                   if not hasattr(self.o, name):
1154: (12)                      raise AttributeError(f'{self.o} has no function {name}')
1155: (8)                   func = getattr(self.o, name)
1156: (8)                   if hasattr(func, '_kwarg_doc'):
1157: (12)                      return func._kwarg_doc
1158: (8)                   docstring = inspect.getdoc(func)
1159: (8)                   if docstring is None:
1160: (12)                      return 'unknown'
1161: (8)                   if docstring.startswith('Alias for '):
1162: (12)                      return None
1163: (8)                   match = self._get_valid_values_regex.search(docstring)
1164: (8)                   if match is not None:
1165: (12)                      return re.sub("\n *", " ", match.group(1))
1166: (8)                   param_name = func.__code__.co_varnames[1]
1167: (8)                   match = re.search(fr"(?m)^ *\*?{param_name} : (.+)", docstring)
1168: (8)                   if match:
1169: (12)                      return match.group(1)
1170: (8)                   return 'unknown'
1171: (4)               def _replace_path(self, source_class):
1172: (8)                   """
1173: (8)                   Changes the full path to the public API path that is used
1174: (8)                   in sphinx. This is needed for links to work.
```

```
(8)                 """
(8)                 replace_dict = {'_base._AxesBase': 'Axes',
(24)                                '_axes.Axes': 'Axes'}
(8)                 for key, value in replace_dict.items():
(12)                    source_class = source_class.replace(key, value)
(8)                 return source_class
(4)             def get_setters(self):
(8)                 """
(8)                 Get the attribute strings with setters for object.
(8)                 For example, for a line, return ``['markerfacecolor', 'linewidth',
(8)                 ....]``.
(8)                 """
(8)                 setters = []
(8)                 for name in dir(self.o):
(12)                    if not name.startswith('set_'):
(16)                        continue
(12)                    func = getattr(self.o, name)
(12)                    if (not callable(func)
(20)                            or self.number_of_parameters(func) < 2
(20)                            or self.is_alias(func)):
(16)                        continue
(12)                    setters.append(name[4:])
(8)                 return setters
(4)             @staticmethod
(4)             @cache
(4)             def number_of_parameters(func):
(8)                 """Return number of parameters of the callable *func*."""
(8)                 return len(inspect.signature(func).parameters)
(4)             @staticmethod
(4)             @cache
(4)             def is_alias(method):
(8)                 """
(8)                 Return whether the object *method* is an alias for another method.
(8)                 """
(8)                 ds = inspect.getdoc(method)
(8)                 if ds is None:
(12)                    return False
(8)                 return ds.startswith('Alias for ')
(4)             def aliased_name(self, s):
(8)                 """
(8)                 Return 'PROPNAME or alias' if *s* has an alias, else return
'PROPNAME'.
(8)                 For example, for the line markerfacecolor property, which has an
(8)                 alias, return 'markerfacecolor or mfc' and for the transform
(8)                 property, which does not, return 'transform'.
(8)                 """
(8)                 aliases = ''.join(' or %s' % x for x in sorted(self.aliasd.get(s,
[])))
(8)                 return s + aliases
(4)             _NOT_LINKABLE = {
(8)                 'matplotlib.image._ImageBase.set_alpha',
(8)                 'matplotlib.image._ImageBase.set_array',
(8)                 'matplotlib.image._ImageBase.set_data',
(8)                 'matplotlib.image._ImageBase.set_filternorm',
(8)                 'matplotlib.image._ImageBase.set_filterrad',
(8)                 'matplotlib.image._ImageBase.set_interpolation',
(8)                 'matplotlib.image._ImageBase.set_interpolation_stage',
(8)                 'matplotlib.image._ImageBase.set_resample',
(8)                 'matplotlib.text._AnnotationBase.set_annotation_clip',
(4)             }
(4)             def aliased_name_rest(self, s, target):
(8)                 """
(8)                 Return 'PROPNAME or alias' if *s* has an alias, else return
'PROPNAME',
(8)                 formatted for reST.
(8)                 For example, for the line markerfacecolor property, which has an
(8)                 alias, return 'markerfacecolor or mfc' and for the transform
(8)                 property, which does not, return 'transform'.
(8)                 """
```

```
(8)                 if target in self._NOT_LINKABLE:
(12)                    return f'``{s}``'
(8)                 aliases = ''.join(' or %s' % x for x in sorted(self.aliasd.get(s, []))) 
(8)                 return f':meth:`{s} <{target}>`{aliases}'
(4)             def pprint_setters(self, prop=None, leadingspace=2):
(8)                 """
(8)                 If *prop* is *None*, return a list of strings of all settable
(8)                 properties and their valid values.
(8)                 If *prop* is not *None*, it is a valid property name and that
(8)                 property will be returned as a string of property : valid
(8)                 values.
(8)                 """
(8)                 if leadingspace:
(12)                    pad = ' ' * leadingspace
(8)                 else:
(12)                    pad = ''
(8)                 if prop is not None:
(12)                    accepts = self.get_valid_values(prop)
(12)                    return f'{pad}{prop}: {accepts}'
(8)                 lines = []
(8)                 for prop in sorted(self.get_setters()):
(12)                    accepts = self.get_valid_values(prop)
(12)                    name = self.aliased_name(prop)
(12)                    lines.append(f'{pad}{name}: {accepts}')
(8)                 return lines
(4)             def pprint_setters_rest(self, prop=None, leadingspace=4):
(8)                 """
(8)                 If *prop* is *None*, return a list of reST-formatted strings of all
(8)                 settable properties and their valid values.
(8)                 If *prop* is not *None*, it is a valid property name and that
(8)                 property will be returned as a string of "property : valid"
(8)                 values.
(8)                 """
(8)                 if leadingspace:
(12)                    pad = ' ' * leadingspace
(8)                 else:
(12)                    pad = ''
(8)                 if prop is not None:
(12)                    accepts = self.get_valid_values(prop)
(12)                    return f'{pad}{prop}: {accepts}'
(8)                 prop_and_qualnames = []
(8)                 for prop in sorted(self.get_setters()):
(12)                    for cls in self.o.__mro__:
(16)                        method = getattr(cls, f"set_{prop}", None)
(16)                        if method and method.__doc__ is not None:
(20)                            break
(12)                    else:  # No docstring available.
(16)                        method = getattr(self.o, f"set_{prop}")
(12)                    prop_and_qualnames.append(
(16)                        (prop, f"{method.__module__}.{method.__qualname__}"))
(8)                 names = [self.aliased_name_rest(prop, target)
(17)                         .replace('_base._AxesBase', 'Axes')
(17)                         .replace('_axes.Axes', 'Axes')
(17)                         for prop, target in prop_and_qualnames]
(8)                 accepts = [self.get_valid_values(prop)
(19)                           for prop, _ in prop_and_qualnames]
(8)                 col0_len = max(len(n) for n in names)
(8)                 col1_len = max(len(a) for a in accepts)
(8)                 table_formatstr = pad + '   ' + '=' * col0_len + '   ' + '=' * col1_len
(8)                 return [
(12)                    '',
(12)                    pad + '.. table::',
(12)                    pad + '   :class: property-table',
(12)                    '',
(12)                    table_formatstr,
(12)                    pad + '   ' + 'Property'.ljust(col0_len)
(12)                    + '   ' + 'Description'.ljust(col1_len),
```

```
(12)                        table_formatstr,
(12)                        *[pad + '    ' + n.ljust(col0_len) + '    ' + a.ljust(col1_len)
(14)                          for n, a in zip(names, accepts)],
(12)                        table_formatstr,
(12)                        '',
(8)                     ]
(4)                 def properties(self):
(8)                     """Return a dictionary mapping property name -> value."""
(8)                     o = self.oorig
(8)                     getters = [name for name in dir(o)
(19)                               if name.startswith('get_') and callable(getattr(o, name))]
(8)                     getters.sort()
(8)                     d = {}
(8)                     for name in getters:
(12)                        func = getattr(o, name)
(12)                        if self.is_alias(func):
(16)                            continue
(12)                        try:
(16)                            with warnings.catch_warnings():
(20)                                warnings.simplefilter('ignore')
(20)                                val = func()
(12)                        except Exception:
(16)                            continue
(12)                        else:
(16)                            d[name[4:]] = val
(8)                     return d
(4)                 def pprint_getters(self):
(8)                     """Return the getters and actual values as list of strings."""
(8)                     lines = []
(8)                     for name, val in sorted(self.properties().items()):
(12)                        if getattr(val, 'shape', ()) != () and len(val) > 6:
(16)                            s = str(val[:6]) + '...'
(12)                        else:
(16)                            s = str(val)
(12)                        s = s.replace('\n', ' ')
(12)                        if len(s) > 50:
(16)                            s = s[:50] + '...'
(12)                        name = self.aliased_name(name)
(12)                        lines.append(f'    {name} = {s}')
(8)                     return lines
(0)             def getp(obj, property=None):
(4)                 """
(4)                 Return the value of an `.Artist`'s *property*, or print all of them.
(4)                 Parameters
(4)                 ----------
(4)                 obj : `~matplotlib.artist.Artist`
(8)                     The queried artist; e.g., a `.Line2D`, a `.Text`, or an `~.axes.Axes`.
(4)                 property : str or None, default: None
(8)                     If *property* is 'somename', this function returns
(8)                     ``obj.get_somename()``.
(8)                     If it's None (or unset), it *prints* all gettable properties from
(8)                     *obj*.  Many properties have aliases for shorter typing, e.g. 'lw' is
(8)                     an alias for 'linewidth'.  In the output, aliases and full property
(8)                     names will be listed as:
(10)                      property or alias = value
(8)                     e.g.:
(10)                      linewidth or lw = 2
(4)                 See Also
(4)                 --------
(4)                 setp
(4)                 """
(4)                 if property is None:
(8)                     insp = ArtistInspector(obj)
(8)                     ret = insp.pprint_getters()
(8)                     print('\n'.join(ret))
(8)                     return
(4)                 return getattr(obj, 'get_' + property)()
(0)             get = getp
(0)             def setp(obj, *args, file=None, **kwargs):
```

```
(4)                 """
(4)                 Set one or more properties on an `.Artist`, or list allowed values.
(4)                 Parameters
(4)                 ----------
(4)                 obj : `~matplotlib.artist.Artist` or list of `.Artist`
(8)                     The artist(s) whose properties are being set or queried.  When setting
(8)                     properties, all artists are affected; when querying the allowed values,
(8)                     only the first instance in the sequence is queried.
(8)                     For example, two lines can be made thicker and red with a single call:
(8)                     >>> x = arange(0, 1, 0.01)
(8)                     >>> lines = plot(x, sin(2*pi*x), x, sin(4*pi*x))
(8)                     >>> setp(lines, linewidth=2, color='r')
(4)                 file : file-like, default: `sys.stdout`
(8)                     Where `setp` writes its output when asked to list allowed values.
(8)                     >>> with open('output.log') as file:
(8)                     ...     setp(line, file=file)
(8)                     The default, ``None``, means `sys.stdout`.
(4)                 *args, **kwargs
(8)                     The properties to set.  The following combinations are supported:
(8)                     - Set the linestyle of a line to be dashed:
(10)                      >>> line, = plot([1, 2, 3])
(10)                      >>> setp(line, linestyle='--')
(8)                     - Set multiple properties at once:
(10)                      >>> setp(line, linewidth=2, color='r')
(8)                     - List allowed values for a line's linestyle:
(10)                      >>> setp(line, 'linestyle')
(10)                      linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
(8)                     - List all properties that can be set, and their allowed values:
(10)                      >>> setp(line)
(10)                      agg_filter: a filter function, ...
(10)                      [long output listing omitted]
(8)                     `setp` also supports MATLAB style string/value pairs.  For example, the
(8)                     following are equivalent:
(8)                     >>> setp(lines, 'linewidth', 2, 'color', 'r')  # MATLAB style
(8)                     >>> setp(lines, linewidth=2, color='r')        # Python style
(4)                 See Also
(4)                 --------
(4)                 getp
(4)                 """
(4)                 if isinstance(obj, Artist):
(8)                     objs = [obj]
(4)                 else:
(8)                     objs = list(cbook.flatten(obj))
(4)                 if not objs:
(8)                     return
(4)                 insp = ArtistInspector(objs[0])
(4)                 if not kwargs and len(args) < 2:
(8)                     if args:
(12)                        print(insp.pprint_setters(prop=args[0]), file=file)
(8)                     else:
(12)                        print('\n'.join(insp.pprint_setters()), file=file)
(8)                     return
(4)                 if len(args) % 2:
(8)                     raise ValueError('The set args must be string, value pairs')
(4)                 funcvals = dict(zip(args[::2], args[1::2]))
(4)                 ret = [o.update(funcvals) for o in objs] + [o.set(**kwargs) for o in objs]
(4)                 return list(cbook.flatten(ret))
(0)             def kwdoc(artist):
(4)                 r"""
(4)                 Inspect an `~matplotlib.artist.Artist` class (using `.ArtistInspector`) and
(4)                 return information about its settable properties and their current values.
(4)                 Parameters
(4)                 ----------
(4)                 artist : `~matplotlib.artist.Artist` or an iterable of `Artist`\s
(4)                 Returns
(4)                 -------
```

```
1443: (4)               str
1444: (8)                   The settable properties of *artist*, as plain text if
1445: (8)                   :rc:`docstring.hardcopy` is False and as a rst table (intended for
1446: (8)                   use in Sphinx) if it is True.
1447: (4)               """
1448: (4)               ai = ArtistInspector(artist)
1449: (4)               return ('\n'.join(ai.pprint_setters_rest(leadingspace=4))
1450: (12)                      if mpl.rcParams['docstring.hardcopy'] else
1451: (12)                      'Properties:\n' + '\n'.join(ai.pprint_setters(leadingspace=4)))
1452: (0)           Artist._update_set_signature_and_docstring()
```

File 23 - axis.py:

```
1: (0)              """
2: (0)              Classes for the ticks and x- and y-axis.
3: (0)              """
4: (0)              import datetime
5: (0)              import functools
6: (0)              import logging
7: (0)              from numbers import Real
8: (0)              import warnings
9: (0)              import numpy as np
10: (0)             import matplotlib as mpl
11: (0)             from matplotlib import _api, cbook
12: (0)             import matplotlib.artist as martist
13: (0)             import matplotlib.colors as mcolors
14: (0)             import matplotlib.lines as mlines
15: (0)             import matplotlib.scale as mscale
16: (0)             import matplotlib.text as mtext
17: (0)             import matplotlib.ticker as mticker
18: (0)             import matplotlib.transforms as mtransforms
19: (0)             import matplotlib.units as munits
20: (0)             _log = logging.getLogger(__name__)
21: (0)             GRIDLINE_INTERPOLATION_STEPS = 180
22: (0)             _line_inspector = martist.ArtistInspector(mlines.Line2D)
23: (0)             _line_param_names = _line_inspector.get_setters()
24: (0)             _line_param_aliases = [list(d)[0] for d in _line_inspector.aliasd.values()]
25: (0)             _gridline_param_names = ['grid_' + name
26: (25)                                     for name in _line_param_names + _line_param_aliases]
27: (0)             class Tick(martist.Artist):
28: (4)                 """
29: (4)                 Abstract base class for the axis ticks, grid lines and labels.
30: (4)                 Ticks mark a position on an Axis. They contain two lines as markers and
31: (4)                 two labels; one each for the bottom and top positions (in case of an
32: (4)                 `.XAxis`) or for the left and right positions (in case of a `.YAxis`).
33: (4)                 Attributes
34: (4)                 ----------
35: (4)                 tick1line : `~matplotlib.lines.Line2D`
36: (8)                     The left/bottom tick marker.
37: (4)                 tick2line : `~matplotlib.lines.Line2D`
38: (8)                     The right/top tick marker.
39: (4)                 gridline : `~matplotlib.lines.Line2D`
40: (8)                     The grid line associated with the label position.
41: (4)                 label1 : `~matplotlib.text.Text`
42: (8)                     The left/bottom tick label.
43: (4)                 label2 : `~matplotlib.text.Text`
44: (8)                     The right/top tick label.
45: (4)                 """
46: (4)                 def __init__(
47: (8)                     self, axes, loc, *,
48: (8)                     size=None,  # points
49: (8)                     width=None,
50: (8)                     color=None,
51: (8)                     tickdir=None,
52: (8)                     pad=None,
53: (8)                     labelsize=None,
54: (8)                     labelcolor=None,
```

```
 55: (8)                     labelfontfamily=None,
 56: (8)                     zorder=None,
 57: (8)                     gridOn=None,  # defaults to axes.grid depending on axes.grid.which
 58: (8)                     tick1On=True,
 59: (8)                     tick2On=True,
 60: (8)                     label1On=True,
 61: (8)                     label2On=False,
 62: (8)                     major=True,
 63: (8)                     labelrotation=0,
 64: (8)                     grid_color=None,
 65: (8)                     grid_linestyle=None,
 66: (8)                     grid_linewidth=None,
 67: (8)                     grid_alpha=None,
 68: (8)                     **kwargs,  # Other Line2D kwargs applied to gridlines.
 69: (4)                 ):
 70: (8)                     """
 71: (8)                     bbox is the Bound2D bounding box in display coords of the Axes
 72: (8)                     loc is the tick location in data coords
 73: (8)                     size is the tick size in points
 74: (8)                     """
 75: (8)                     super().__init__()
 76: (8)                     if gridOn is None:
 77: (12)                        which = mpl.rcParams['axes.grid.which']
 78: (12)                        if major and (which in ('both', 'major')):
 79: (16)                            gridOn = mpl.rcParams['axes.grid']
 80: (12)                        elif (not major) and (which in ('both', 'minor')):
 81: (16)                            gridOn = mpl.rcParams['axes.grid']
 82: (12)                        else:
 83: (16)                            gridOn = False
 84: (8)                     self.set_figure(axes.figure)
 85: (8)                     self.axes = axes
 86: (8)                     self._loc = loc
 87: (8)                     self._major = major
 88: (8)                     name = self.__name__
 89: (8)                     major_minor = "major" if major else "minor"
 90: (8)                     if size is None:
 91: (12)                        size = mpl.rcParams[f"{name}.{major_minor}.size"]
 92: (8)                     self._size = size
 93: (8)                     if width is None:
 94: (12)                        width = mpl.rcParams[f"{name}.{major_minor}.width"]
 95: (8)                     self._width = width
 96: (8)                     if color is None:
 97: (12)                        color = mpl.rcParams[f"{name}.color"]
 98: (8)                     if pad is None:
 99: (12)                        pad = mpl.rcParams[f"{name}.{major_minor}.pad"]
100: (8)                     self._base_pad = pad
101: (8)                     if labelcolor is None:
102: (12)                        labelcolor = mpl.rcParams[f"{name}.labelcolor"]
103: (8)                     if cbook._str_equal(labelcolor, 'inherit'):
104: (12)                        labelcolor = mpl.rcParams[f"{name}.color"]
105: (8)                     if labelsize is None:
106: (12)                        labelsize = mpl.rcParams[f"{name}.labelsize"]
107: (8)                     self._set_labelrotation(labelrotation)
108: (8)                     if zorder is None:
109: (12)                        if major:
110: (16)                            zorder = mlines.Line2D.zorder + 0.01
111: (12)                        else:
112: (16)                            zorder = mlines.Line2D.zorder
113: (8)                     self._zorder = zorder
114: (8)                     grid_color = mpl._val_or_rc(grid_color, "grid.color")
115: (8)                     grid_linestyle = mpl._val_or_rc(grid_linestyle, "grid.linestyle")
116: (8)                     grid_linewidth = mpl._val_or_rc(grid_linewidth, "grid.linewidth")
117: (8)                     if grid_alpha is None and not mcolors._has_alpha_channel(grid_color):
118: (12)                        grid_alpha = mpl.rcParams["grid.alpha"]
119: (8)                     grid_kw = {k[5:]: v for k, v in kwargs.items()}
120: (8)                     self.tick1line = mlines.Line2D(
121: (12)                        [], [],
122: (12)                        color=color, linestyle="none", zorder=zorder, visible=tick1On,
123: (12)                        markeredgecolor=color, markersize=size, markeredgewidth=width,
```

```
124: (8)                    )
125: (8)                    self.tick2line = mlines.Line2D(
126: (12)                       [], [],
127: (12)                       color=color, linestyle="none", zorder=zorder, visible=tick2On,
128: (12)                       markeredgecolor=color, markersize=size, markeredgewidth=width,
129: (8)                    )
130: (8)                    self.gridline = mlines.Line2D(
131: (12)                       [], [],
132: (12)                       color=grid_color, alpha=grid_alpha, visible=gridOn,
133: (12)                       linestyle=grid_linestyle, linewidth=grid_linewidth, marker="",
134: (12)                       **grid_kw,
135: (8)                    )
136: (8)                    self.gridline.get_path()._interpolation_steps = \
137: (12)                       GRIDLINE_INTERPOLATION_STEPS
138: (8)                    self.label1 = mtext.Text(
139: (12)                       np.nan, np.nan,
140: (12)                       fontsize=labelsize, color=labelcolor, visible=label1On,
141: (12)                       fontfamily=labelfontfamily, rotation=self._labelrotation[1])
142: (8)                    self.label2 = mtext.Text(
143: (12)                       np.nan, np.nan,
144: (12)                       fontsize=labelsize, color=labelcolor, visible=label2On,
145: (12)                       fontfamily=labelfontfamily, rotation=self._labelrotation[1])
146: (8)                    self._apply_tickdir(tickdir)
147: (8)                    for artist in [self.tick1line, self.tick2line, self.gridline,
148: (23)                                  self.label1, self.label2]:
149: (12)                       self._set_artist_props(artist)
150: (8)                    self.update_position(loc)
151: (4)                def _set_labelrotation(self, labelrotation):
152: (8)                    if isinstance(labelrotation, str):
153: (12)                       mode = labelrotation
154: (12)                       angle = 0
155: (8)                    elif isinstance(labelrotation, (tuple, list)):
156: (12)                       mode, angle = labelrotation
157: (8)                    else:
158: (12)                       mode = 'default'
159: (12)                       angle = labelrotation
160: (8)                    _api.check_in_list(['auto', 'default'], labelrotation=mode)
161: (8)                    self._labelrotation = (mode, angle)
162: (4)                @property
163: (4)                def _pad(self):
164: (8)                    return self._base_pad + self.get_tick_padding()
165: (4)                def _apply_tickdir(self, tickdir):
166: (8)                    """Set tick direction.  Valid values are 'out', 'in', 'inout'."""
167: (8)                    if tickdir is None:
168: (12)                       tickdir = mpl.rcParams[f'{self.__name__}.direction']
169: (8)                    else:
170: (12)                       _api.check_in_list(['in', 'out', 'inout'], tickdir=tickdir)
171: (8)                    self._tickdir = tickdir
172: (4)                def get_tickdir(self):
173: (8)                    return self._tickdir
174: (4)                def get_tick_padding(self):
175: (8)                    """Get the length of the tick outside of the Axes."""
176: (8)                    padding = {
177: (12)                       'in': 0.0,
178: (12)                       'inout': 0.5,
179: (12)                       'out': 1.0
180: (8)                    }
181: (8)                    return self._size * padding[self._tickdir]
182: (4)                def get_children(self):
183: (8)                    children = [self.tick1line, self.tick2line,
184: (20)                               self.gridline, self.label1, self.label2]
185: (8)                    return children
186: (4)                @_api.rename_parameter("3.8", "clippath", "path")
187: (4)                def set_clip_path(self, path, transform=None):
188: (8)                    super().set_clip_path(path, transform)
189: (8)                    self.gridline.set_clip_path(path, transform)
190: (8)                    self.stale = True
191: (4)                def contains(self, mouseevent):
192: (8)                    """
```

```
(8)                     Test whether the mouse event occurred in the Tick marks.
(8)                     This function always returns false.  It is more useful to test if the
(8)                     axis as a whole contains the mouse rather than the set of tick marks.
(8)                     """
(8)                     return False, {}
(4)                 def set_pad(self, val):
(8)                     """
(8)                     Set the tick label pad in points
(8)                     Parameters
(8)                     ----------
(8)                     val : float
(8)                     """
(8)                     self._apply_params(pad=val)
(8)                     self.stale = True
(4)                 def get_pad(self):
(8)                     """Get the value of the tick label pad in points."""
(8)                     return self._base_pad
(4)                 def get_loc(self):
(8)                     """Return the tick location (data coords) as a scalar."""
(8)                     return self._loc
(4)                 @martist.allow_rasterization
(4)                 def draw(self, renderer):
(8)                     if not self.get_visible():
(12)                        self.stale = False
(12)                        return
(8)                     renderer.open_group(self.__name__, gid=self.get_gid())
(8)                     for artist in [self.gridline, self.tick1line, self.tick2line,
(23)                                    self.label1, self.label2]:
(12)                        artist.draw(renderer)
(8)                     renderer.close_group(self.__name__)
(8)                     self.stale = False
(4)                 @_api.deprecated("3.8")
(4)                 def set_label1(self, s):
(8)                     """
(8)                     Set the label1 text.
(8)                     Parameters
(8)                     ----------
(8)                     s : str
(8)                     """
(8)                     self.label1.set_text(s)
(8)                     self.stale = True
(4)                 set_label = set_label1
(4)                 @_api.deprecated("3.8")
(4)                 def set_label2(self, s):
(8)                     """
(8)                     Set the label2 text.
(8)                     Parameters
(8)                     ----------
(8)                     s : str
(8)                     """
(8)                     self.label2.set_text(s)
(8)                     self.stale = True
(4)                 def set_url(self, url):
(8)                     """
(8)                     Set the url of label1 and label2.
(8)                     Parameters
(8)                     ----------
(8)                     url : str
(8)                     """
(8)                     super().set_url(url)
(8)                     self.label1.set_url(url)
(8)                     self.label2.set_url(url)
(8)                     self.stale = True
(4)                 def _set_artist_props(self, a):
(8)                     a.set_figure(self.figure)
(4)                 def get_view_interval(self):
(8)                     """
(8)                     Return the view limits ``(min, max)`` of the axis the tick belongs to.
(8)                     """
```

```
262: (8)                    raise NotImplementedError('Derived must override')
263: (4)                def _apply_params(self, **kwargs):
264: (8)                    for name, target in [("gridOn", self.gridline),
265: (29)                                         ("tick1On", self.tick1line),
266: (29)                                         ("tick2On", self.tick2line),
267: (29)                                         ("label1On", self.label1),
268: (29)                                         ("label2On", self.label2)]:
269: (12)                        if name in kwargs:
270: (16)                            target.set_visible(kwargs.pop(name))
271: (8)                    if any(k in kwargs for k in ['size', 'width', 'pad', 'tickdir']):
272: (12)                        self._size = kwargs.pop('size', self._size)
273: (12)                        self._width = kwargs.pop('width', self._width)
274: (12)                        self._base_pad = kwargs.pop('pad', self._base_pad)
275: (12)                        self._apply_tickdir(kwargs.pop('tickdir', self._tickdir))
276: (12)                        for line in (self.tick1line, self.tick2line):
277: (16)                            line.set_markersize(self._size)
278: (16)                            line.set_markeredgewidth(self._width)
279: (12)                        trans = self._get_text1_transform()[0]
280: (12)                        self.label1.set_transform(trans)
281: (12)                        trans = self._get_text2_transform()[0]
282: (12)                        self.label2.set_transform(trans)
283: (8)                    tick_kw = {k: v for k, v in kwargs.items() if k in ['color',
'zorder']}
284: (8)                    if 'color' in kwargs:
285: (12)                        tick_kw['markeredgecolor'] = kwargs['color']
286: (8)                    self.tick1line.set(**tick_kw)
287: (8)                    self.tick2line.set(**tick_kw)
288: (8)                    for k, v in tick_kw.items():
289: (12)                        setattr(self, '_' + k, v)
290: (8)                    if 'labelrotation' in kwargs:
291: (12)                        self._set_labelrotation(kwargs.pop('labelrotation'))
292: (12)                        self.label1.set(rotation=self._labelrotation[1])
293: (12)                        self.label2.set(rotation=self._labelrotation[1])
294: (8)                    label_kw = {k[5:]: v for k, v in kwargs.items()
295: (20)                                if k in ['labelsize', 'labelcolor', 'labelfontfamily']}
296: (8)                    self.label1.set(**label_kw)
297: (8)                    self.label2.set(**label_kw)
298: (8)                    grid_kw = {k[5:]: v for k, v in kwargs.items()
299: (19)                               if k in _gridline_param_names}
300: (8)                    self.gridline.set(**grid_kw)
301: (4)                def update_position(self, loc):
302: (8)                    """Set the location of tick in data coords with scalar *loc*."""
303: (8)                    raise NotImplementedError('Derived must override')
304: (4)                def _get_text1_transform(self):
305: (8)                    raise NotImplementedError('Derived must override')
306: (4)                def _get_text2_transform(self):
307: (8)                    raise NotImplementedError('Derived must override')
308: (0)            class XTick(Tick):
309: (4)                """
310: (4)                Contains all the Artists needed to make an x tick - the tick line,
311: (4)                the label text and the grid line
312: (4)                """
313: (4)                __name__ = 'xtick'
314: (4)                def __init__(self, *args, **kwargs):
315: (8)                    super().__init__(*args, **kwargs)
316: (8)                    ax = self.axes
317: (8)                    self.tick1line.set(
318: (12)                       data=([0], [0]), transform=ax.get_xaxis_transform("tick1"))
319: (8)                    self.tick2line.set(
320: (12)                       data=([0], [1]), transform=ax.get_xaxis_transform("tick2"))
321: (8)                    self.gridline.set(
322: (12)                       data=([0, 0], [0, 1]), transform=ax.get_xaxis_transform("grid"))
323: (8)                    trans, va, ha = self._get_text1_transform()
324: (8)                    self.label1.set(
325: (12)                       x=0, y=0,
326: (12)                       verticalalignment=va, horizontalalignment=ha, transform=trans,
327: (8)                    )
328: (8)                    trans, va, ha = self._get_text2_transform()
329: (8)                    self.label2.set(
```

```
(12)                        x=0, y=1,
(12)                        verticalalignment=va, horizontalalignment=ha, transform=trans,
(8)                     )
(4)                 def _get_text1_transform(self):
(8)                     return self.axes.get_xaxis_text1_transform(self._pad)
(4)                 def _get_text2_transform(self):
(8)                     return self.axes.get_xaxis_text2_transform(self._pad)
(4)                 def _apply_tickdir(self, tickdir):
(8)                     super()._apply_tickdir(tickdir)
(8)                     mark1, mark2 = {
(12)                        'out': (mlines.TICKDOWN, mlines.TICKUP),
(12)                        'in': (mlines.TICKUP, mlines.TICKDOWN),
(12)                        'inout': ('|', '|'),
(8)                     }[self._tickdir]
(8)                     self.tick1line.set_marker(mark1)
(8)                     self.tick2line.set_marker(mark2)
(4)                 def update_position(self, loc):
(8)                     """Set the location of tick in data coords with scalar *loc*."""
(8)                     self.tick1line.set_xdata((loc,))
(8)                     self.tick2line.set_xdata((loc,))
(8)                     self.gridline.set_xdata((loc,))
(8)                     self.label1.set_x(loc)
(8)                     self.label2.set_x(loc)
(8)                     self._loc = loc
(8)                     self.stale = True
(4)                 def get_view_interval(self):
(8)                     return self.axes.viewLim.intervalx
(0)             class YTick(Tick):
(4)                 """
(4)                 Contains all the Artists needed to make a Y tick - the tick line,
(4)                 the label text and the grid line
(4)                 """
(4)                 __name__ = 'ytick'
(4)                 def __init__(self, *args, **kwargs):
(8)                     super().__init__(*args, **kwargs)
(8)                     ax = self.axes
(8)                     self.tick1line.set(
(12)                        data=([0], [0]), transform=ax.get_yaxis_transform("tick1"))
(8)                     self.tick2line.set(
(12)                        data=([1], [0]), transform=ax.get_yaxis_transform("tick2"))
(8)                     self.gridline.set(
(12)                        data=([0, 1], [0, 0]), transform=ax.get_yaxis_transform("grid"))
(8)                     trans, va, ha = self._get_text1_transform()
(8)                     self.label1.set(
(12)                        x=0, y=0,
(12)                        verticalalignment=va, horizontalalignment=ha, transform=trans,
(8)                     )
(8)                     trans, va, ha = self._get_text2_transform()
(8)                     self.label2.set(
(12)                        x=1, y=0,
(12)                        verticalalignment=va, horizontalalignment=ha, transform=trans,
(8)                     )
(4)                 def _get_text1_transform(self):
(8)                     return self.axes.get_yaxis_text1_transform(self._pad)
(4)                 def _get_text2_transform(self):
(8)                     return self.axes.get_yaxis_text2_transform(self._pad)
(4)                 def _apply_tickdir(self, tickdir):
(8)                     super()._apply_tickdir(tickdir)
(8)                     mark1, mark2 = {
(12)                        'out': (mlines.TICKLEFT, mlines.TICKRIGHT),
(12)                        'in': (mlines.TICKRIGHT, mlines.TICKLEFT),
(12)                        'inout': ('_', '_'),
(8)                     }[self._tickdir]
(8)                     self.tick1line.set_marker(mark1)
(8)                     self.tick2line.set_marker(mark2)
(4)                 def update_position(self, loc):
(8)                     """Set the location of tick in data coords with scalar *loc*."""
(8)                     self.tick1line.set_ydata((loc,))
(8)                     self.tick2line.set_ydata((loc,))
```

```
(8)                 self.gridline.set_ydata((loc,))
(8)                 self.label1.set_y(loc)
(8)                 self.label2.set_y(loc)
(8)                 self._loc = loc
(8)                 self.stale = True
(4)             def get_view_interval(self):
(8)                 return self.axes.viewLim.intervaly
(0)         class Ticker:
(4)             """
(4)             A container for the objects defining tick position and format.
(4)             Attributes
(4)             ----------
(4)             locator : `~matplotlib.ticker.Locator` subclass
(8)                 Determines the positions of the ticks.
(4)             formatter : `~matplotlib.ticker.Formatter` subclass
(8)                 Determines the format of the tick labels.
(4)             """
(4)             def __init__(self):
(8)                 self._locator = None
(8)                 self._formatter = None
(8)                 self._locator_is_default = True
(8)                 self._formatter_is_default = True
(4)             @property
(4)             def locator(self):
(8)                 return self._locator
(4)             @locator.setter
(4)             def locator(self, locator):
(8)                 if not isinstance(locator, mticker.Locator):
(12)                    raise TypeError('locator must be a subclass of '
(28)                                    'matplotlib.ticker.Locator')
(8)                 self._locator = locator
(4)             @property
(4)             def formatter(self):
(8)                 return self._formatter
(4)             @formatter.setter
(4)             def formatter(self, formatter):
(8)                 if not isinstance(formatter, mticker.Formatter):
(12)                    raise TypeError('formatter must be a subclass of '
(28)                                    'matplotlib.ticker.Formatter')
(8)                 self._formatter = formatter
(0)         class _LazyTickList:
(4)             """
(4)             A descriptor for lazy instantiation of tick lists.
(4)             See comment above definition of the ``majorTicks`` and ``minorTicks``
(4)             attributes.
(4)             """
(4)             def __init__(self, major):
(8)                 self._major = major
(4)             def __get__(self, instance, owner):
(8)                 if instance is None:
(12)                    return self
(8)                 else:
(12)                    if self._major:
(16)                        instance.majorTicks = []
(16)                        tick = instance._get_tick(major=True)
(16)                        instance.majorTicks.append(tick)
(16)                        return instance.majorTicks
(12)                    else:
(16)                        instance.minorTicks = []
(16)                        tick = instance._get_tick(major=False)
(16)                        instance.minorTicks.append(tick)
(16)                        return instance.minorTicks
(0)         class Axis(martist.Artist):
(4)             """
(4)             Base class for `.XAxis` and `.YAxis`.
(4)             Attributes
(4)             ----------
(4)             isDefault_label : bool
(4)             axes : `~matplotlib.axes.Axes`
```

```
            The `~.axes.Axes` to which the Axis belongs.
        major : `~matplotlib.axis.Ticker`
            Determines the major tick positions and their label format.
        minor : `~matplotlib.axis.Ticker`
            Determines the minor tick positions and their label format.
        callbacks : `~matplotlib.cbook.CallbackRegistry`
        label : `~matplotlib.text.Text`
            The axis label.
        labelpad : float
            The distance between the axis label and the tick labels.
            Defaults to :rc:`axes.labelpad` = 4.
        offsetText : `~matplotlib.text.Text`
            A `.Text` object containing the data offset of the ticks (if any).
        pickradius : float
            The acceptance radius for containment tests. See also
`.Axis.contains`.
        majorTicks : list of `.Tick`
            The major ticks.
            .. warning::
                Ticks are not guaranteed to be persistent. Various operations
                can create, delete and modify the Tick instances. There is an
                imminent risk that changes to individual ticks will not
                survive if you work on the figure further (including also
                panning/zooming on a displayed figure).
                Working on the individual ticks is a method of last resort.
                Use `.set_tick_params` instead if possible.
        minorTicks : list of `.Tick`
            The minor ticks.
        """
        OFFSETTEXTPAD = 3
        _tick_class = None
        def __str__(self):
            return "{}({},{})".format(
                type(self).__name__, *self.axes.transAxes.transform((0, 0)))
        def __init__(self, axes, *, pickradius=15, clear=True):
            """
            Parameters
            ----------
            axes : `~matplotlib.axes.Axes`
                The `~.axes.Axes` to which the created Axis belongs.
            pickradius : float
                The acceptance radius for containment tests. See also
                `.Axis.contains`.
            clear : bool, default: True
                Whether to clear the Axis on creation. This is not required, e.g.,
when
                creating an Axis as part of an Axes, as ``Axes.clear`` will call
                ``Axis.clear``.
                .. versionadded:: 3.8
            """
            super().__init__()
            self._remove_overlapping_locs = True
            self.set_figure(axes.figure)
            self.isDefault_label = True
            self.axes = axes
            self.major = Ticker()
            self.minor = Ticker()
            self.callbacks = cbook.CallbackRegistry(signals=["units"])
            self._autolabelpos = True
            self.label = mtext.Text(
                np.nan, np.nan,
                fontsize=mpl.rcParams['axes.labelsize'],
                fontweight=mpl.rcParams['axes.labelweight'],
                color=mpl.rcParams['axes.labelcolor'],
            )
            self._set_artist_props(self.label)
            self.offsetText = mtext.Text(np.nan, np.nan)
            self._set_artist_props(self.offsetText)
            self.labelpad = mpl.rcParams['axes.labelpad']
```

```
(8)                 self.pickradius = pickradius
(8)                 self._major_tick_kw = dict()
(8)                 self._minor_tick_kw = dict()
(8)                 if clear:
(12)                    self.clear()
(8)                 else:
(12)                    self.converter = None
(12)                    self.units = None
(8)                 self._autoscale_on = True
(4)             @property
(4)             def isDefault_majloc(self):
(8)                 return self.major._locator_is_default
(4)             @isDefault_majloc.setter
(4)             def isDefault_majloc(self, value):
(8)                 self.major._locator_is_default = value
(4)             @property
(4)             def isDefault_majfmt(self):
(8)                 return self.major._formatter_is_default
(4)             @isDefault_majfmt.setter
(4)             def isDefault_majfmt(self, value):
(8)                 self.major._formatter_is_default = value
(4)             @property
(4)             def isDefault_minloc(self):
(8)                 return self.minor._locator_is_default
(4)             @isDefault_minloc.setter
(4)             def isDefault_minloc(self, value):
(8)                 self.minor._locator_is_default = value
(4)             @property
(4)             def isDefault_minfmt(self):
(8)                 return self.minor._formatter_is_default
(4)             @isDefault_minfmt.setter
(4)             def isDefault_minfmt(self, value):
(8)                 self.minor._formatter_is_default = value
(4)             def _get_shared_axes(self):
(8)                 """Return Grouper of shared Axes for current axis."""
(8)                 return self.axes._shared_axes[
(12)                    self._get_axis_name()].get_siblings(self.axes)
(4)             def _get_shared_axis(self):
(8)                 """Return list of shared axis for current axis."""
(8)                 name = self._get_axis_name()
(8)                 return [ax._axis_map[name] for ax in self._get_shared_axes()]
(4)             def _get_axis_name(self):
(8)                 """Return the axis name."""
(8)                 return [name for name, axis in self.axes._axis_map.items()
(16)                        if axis is self][0]
(4)             majorTicks = _LazyTickList(major=True)
(4)             minorTicks = _LazyTickList(major=False)
(4)             def get_remove_overlapping_locs(self):
(8)                 return self._remove_overlapping_locs
(4)             def set_remove_overlapping_locs(self, val):
(8)                 self._remove_overlapping_locs = bool(val)
(4)             remove_overlapping_locs = property(
(8)                 get_remove_overlapping_locs, set_remove_overlapping_locs,
(8)                 doc=('If minor ticker locations that overlap with major '
(13)                     'ticker locations should be trimmed.'))
(4)             def set_label_coords(self, x, y, transform=None):
(8)                 """
(8)                 Set the coordinates of the label.
(8)                 By default, the x coordinate of the y label and the y coordinate of
the
(8)                 x label are determined by the tick label bounding boxes, but this can
(8)                 lead to poor alignment of multiple labels if there are multiple Axes.
(8)                 You can also specify the coordinate system of the label with the
(8)                 transform.  If None, the default coordinate system will be the axes
(8)                 coordinate system: (0, 0) is bottom left, (0.5, 0.5) is center, etc.
(8)                 """
(8)                 self._autolabelpos = False
(8)                 if transform is None:
(12)                    transform = self.axes.transAxes
```

```
603: (8)                    self.label.set_transform(transform)
604: (8)                    self.label.set_position((x, y))
605: (8)                    self.stale = True
606: (4)                def get_transform(self):
607: (8)                    """Return the transform used in the Axis' scale"""
608: (8)                    return self._scale.get_transform()
609: (4)                def get_scale(self):
610: (8)                    """Return this Axis' scale (as a str)."""
611: (8)                    return self._scale.name
612: (4)                def _set_scale(self, value, **kwargs):
613: (8)                    if not isinstance(value, mscale.ScaleBase):
614: (12)                       self._scale = mscale.scale_factory(value, self, **kwargs)
615: (8)                    else:
616: (12)                       self._scale = value
617: (8)                    self._scale.set_default_locators_and_formatters(self)
618: (8)                    self.isDefault_majloc = True
619: (8)                    self.isDefault_minloc = True
620: (8)                    self.isDefault_majfmt = True
621: (8)                    self.isDefault_minfmt = True
622: (4)                def _set_axes_scale(self, value, **kwargs):
623: (8)                    """
624: (8)                    Set this Axis' scale.
625: (8)                    Parameters
626: (8)                    ----------
627: (8)                    value : {"linear", "log", "symlog", "logit", ...} or `.ScaleBase`
628: (12)                       The axis scale type to apply.
629: (8)                    **kwargs
630: (12)                       Different keyword arguments are accepted, depending on the scale.
631: (12)                       See the respective class keyword arguments:
632: (12)                       - `matplotlib.scale.LinearScale`
633: (12)                       - `matplotlib.scale.LogScale`
634: (12)                       - `matplotlib.scale.SymmetricalLogScale`
635: (12)                       - `matplotlib.scale.LogitScale`
636: (12)                       - `matplotlib.scale.FuncScale`
637: (12)                       - `matplotlib.scale.AsinhScale`
638: (8)                    Notes
639: (8)                    -----
640: (8)                    By default, Matplotlib supports the above-mentioned scales.
641: (8)                    Additionally, custom scales may be registered using
642: (8)                    `matplotlib.scale.register_scale`. These scales can then also
643: (8)                    be used here.
644: (8)                    """
645: (8)                    name = self._get_axis_name()
646: (8)                    old_default_lims = (self.get_major_locator()
647: (28)                                       .nonsingular(-np.inf, np.inf))
648: (8)                    for ax in self._get_shared_axes():
649: (12)                       ax._axis_map[name]._set_scale(value, **kwargs)
650: (12)                       ax._update_transScale()
651: (12)                       ax.stale = True
652: (8)                    new_default_lims = (self.get_major_locator()
653: (28)                                       .nonsingular(-np.inf, np.inf))
654: (8)                    if old_default_lims != new_default_lims:
655: (12)                       self.axes.autoscale_view(
656: (16)                           **{f"scale{k}": k == name for k in self.axes._axis_names})
657: (4)                def limit_range_for_scale(self, vmin, vmax):
658: (8)                    return self._scale.limit_range_for_scale(vmin, vmax,
self.get_minpos())
659: (4)                def _get_autoscale_on(self):
660: (8)                    """Return whether this Axis is autoscaled."""
661: (8)                    return self._autoscale_on
662: (4)                def _set_autoscale_on(self, b):
663: (8)                    """
664: (8)                    Set whether this Axis is autoscaled when drawing or by
665: (8)                    `.Axes.autoscale_view`. If b is None, then the value is not changed.
666: (8)                    Parameters
667: (8)                    ----------
668: (8)                    b : bool
669: (8)                    """
670: (8)                    if b is not None:
```

```
(12)                        self._autoscale_on = b
(4)                 def get_children(self):
(8)                     return [self.label, self.offsetText,
(16)                            *self.get_major_ticks(), *self.get_minor_ticks()]
(4)                 def _reset_major_tick_kw(self):
(8)                     self._major_tick_kw.clear()
(8)                     self._major_tick_kw['gridOn'] = (
(16)                            mpl.rcParams['axes.grid'] and
(16)                            mpl.rcParams['axes.grid.which'] in ('both', 'major'))
(4)                 def _reset_minor_tick_kw(self):
(8)                     self._minor_tick_kw.clear()
(8)                     self._minor_tick_kw['gridOn'] = (
(16)                            mpl.rcParams['axes.grid'] and
(16)                            mpl.rcParams['axes.grid.which'] in ('both', 'minor'))
(4)                 def clear(self):
(8)                     """
(8)                     Clear the axis.
(8)                     This resets axis properties to their default values:
(8)                     - the label
(8)                     - the scale
(8)                     - locators, formatters and ticks
(8)                     - major and minor grid
(8)                     - units
(8)                     - registered callbacks
(8)                     """
(8)                     self.label._reset_visual_defaults()
(8)                     self.label.set_color(mpl.rcParams['axes.labelcolor'])
(8)                     self.label.set_fontsize(mpl.rcParams['axes.labelsize'])
(8)                     self.label.set_fontweight(mpl.rcParams['axes.labelweight'])
(8)                     self.offsetText._reset_visual_defaults()
(8)                     self.labelpad = mpl.rcParams['axes.labelpad']
(8)                     self._init()
(8)                     self._set_scale('linear')
(8)                     self.callbacks = cbook.CallbackRegistry(signals=["units"])
(8)                     self._major_tick_kw['gridOn'] = (
(16)                            mpl.rcParams['axes.grid'] and
(16)                            mpl.rcParams['axes.grid.which'] in ('both', 'major'))
(8)                     self._minor_tick_kw['gridOn'] = (
(16)                            mpl.rcParams['axes.grid'] and
(16)                            mpl.rcParams['axes.grid.which'] in ('both', 'minor'))
(8)                     self.reset_ticks()
(8)                     self.converter = None
(8)                     self.units = None
(8)                     self.stale = True
(4)                 def reset_ticks(self):
(8)                     """
(8)                     Re-initialize the major and minor Tick lists.
(8)                     Each list starts with a single fresh Tick.
(8)                     """
(8)                     try:
(12)                        del self.majorTicks
(8)                     except AttributeError:
(12)                        pass
(8)                     try:
(12)                        del self.minorTicks
(8)                     except AttributeError:
(12)                        pass
(8)                     try:
(12)                        self.set_clip_path(self.axes.patch)
(8)                     except AttributeError:
(12)                        pass
(4)                 def minorticks_on(self):
(8)                     """
(8)                     Display default minor ticks on the Axis, depending on the scale
(8)                     (`~.axis.Axis.get_scale`).
(8)                     Scales use specific minor locators:
(8)                     - log: `~.LogLocator`
(8)                     - symlog: `~.SymmetricalLogLocator`
(8)                     - asinh: `~.AsinhLocator`
```

```
(8)                     - logit: `~.LogitLocator`
(8)                     - default: `~.AutoMinorLocator`
(8)                     Displaying minor ticks may reduce performance; you may turn them off
(8)                     using `minorticks_off()` if drawing speed is a problem.
(8)                     """
(8)                     scale = self.get_scale()
(8)                     if scale == 'log':
(12)                        s = self._scale
(12)                        self.set_minor_locator(mticker.LogLocator(s.base, s.subs))
(8)                     elif scale == 'symlog':
(12)                        s = self._scale
(12)                        self.set_minor_locator(
(16)                            mticker.SymmetricalLogLocator(s._transform, s.subs))
(8)                     elif scale == 'asinh':
(12)                        s = self._scale
(12)                        self.set_minor_locator(
(20)                                mticker.AsinhLocator(s.linear_width, base=s._base,
(41)                                                     subs=s._subs))
(8)                     elif scale == 'logit':
(12)                        self.set_minor_locator(mticker.LogitLocator(minor=True))
(8)                     else:
(12)                        self.set_minor_locator(mticker.AutoMinorLocator())
(4)                 def minorticks_off(self):
(8)                     """Remove minor ticks from the Axis."""
(8)                     self.set_minor_locator(mticker.NullLocator())
(4)                 def set_tick_params(self, which='major', reset=False, **kwargs):
(8)                     """
(8)                     Set appearance parameters for ticks, ticklabels, and gridlines.
(8)                     For documentation of keyword arguments, see
(8)                     :meth:`matplotlib.axes.Axes.tick_params`.
(8)                     See Also
(8)                     --------
(8)                     .Axis.get_tick_params
(12)                        View the current style settings for ticks, ticklabels, and
(12)                        gridlines.
(8)                     """
(8)                     _api.check_in_list(['major', 'minor', 'both'], which=which)
(8)                     kwtrans = self._translate_tick_params(kwargs)
(8)                     if reset:
(12)                        if which in ['major', 'both']:
(16)                            self._reset_major_tick_kw()
(16)                            self._major_tick_kw.update(kwtrans)
(12)                        if which in ['minor', 'both']:
(16)                            self._reset_minor_tick_kw()
(16)                            self._minor_tick_kw.update(kwtrans)
(12)                        self.reset_ticks()
(8)                     else:
(12)                        if which in ['major', 'both']:
(16)                            self._major_tick_kw.update(kwtrans)
(16)                            for tick in self.majorTicks:
(20)                                tick._apply_params(**kwtrans)
(12)                        if which in ['minor', 'both']:
(16)                            self._minor_tick_kw.update(kwtrans)
(16)                            for tick in self.minorTicks:
(20)                                tick._apply_params(**kwtrans)
(12)                        if 'label1On' in kwtrans or 'label2On' in kwtrans:
(16)                            self.offsetText.set_visible(
(20)                                self._major_tick_kw.get('label1On', False)
(20)                                or self._major_tick_kw.get('label2On', False))
(12)                        if 'labelcolor' in kwtrans:
(16)                            self.offsetText.set_color(kwtrans['labelcolor'])
(8)                     self.stale = True
(4)                 def get_tick_params(self, which='major'):
(8)                     """
(8)                     Get appearance parameters for ticks, ticklabels, and gridlines.
(8)                     .. versionadded:: 3.7
(8)                     Parameters
(8)                     ----------
(8)                     which : {'major', 'minor'}, default: 'major'
```

```
(12)                        The group of ticks for which the parameters are retrieved.
(8)                     Returns
(8)                     -------
(8)                     dict
(12)                        Properties for styling tick elements added to the axis.
(8)                     Notes
(8)                     -----
(8)                     This method returns the appearance parameters for styling *new*
(8)                     elements added to this axis and may be different from the values
(8)                     on current elements if they were modified directly by the user
(8)                     (e.g., via ``set_*`` methods on individual tick objects).
(8)                     Examples
(8)                     --------
(8)                     ::
(12)                        >>> ax.yaxis.set_tick_params(labelsize=30, labelcolor='red',
(12)                        ...                           direction='out', which='major')
(12)                        >>> ax.yaxis.get_tick_params(which='major')
(12)                        {'direction': 'out',
(12)                        'left': True,
(12)                        'right': False,
(12)                        'labelleft': True,
(12)                        'labelright': False,
(12)                        'gridOn': False,
(12)                        'labelsize': 30,
(12)                        'labelcolor': 'red'}
(12)                        >>> ax.yaxis.get_tick_params(which='minor')
(12)                        {'left': True,
(12)                        'right': False,
(12)                        'labelleft': True,
(12)                        'labelright': False,
(12)                        'gridOn': False}
(8)                     """
(8)                     _api.check_in_list(['major', 'minor'], which=which)
(8)                     if which == 'major':
(12)                        return self._translate_tick_params(
(16)                            self._major_tick_kw, reverse=True
(12)                        )
(8)                     return self._translate_tick_params(self._minor_tick_kw, reverse=True)
(4)                 @staticmethod
(4)                 def _translate_tick_params(kw, reverse=False):
(8)                     """
(8)                     Translate the kwargs supported by `.Axis.set_tick_params` to kwargs
(8)                     supported by `.Tick._apply_params`.
(8)                     In particular, this maps axis specific names like 'top', 'left'
(8)                     to the generic tick1, tick2 logic of the axis. Additionally, there
(8)                     are some other name translations.
(8)                     Returns a new dict of translated kwargs.
(8)                     Note: Use reverse=True to translate from those supported by
(8)                     `.Tick._apply_params` back to those supported by
(8)                     `.Axis.set_tick_params`.
(8)                     """
(8)                     kw_ = {**kw}
(8)                     allowed_keys = [
(12)                        'size', 'width', 'color', 'tickdir', 'pad',
(12)                        'labelsize', 'labelcolor', 'labelfontfamily', 'zorder', 'gridOn',
(12)                        'tick1On', 'tick2On', 'label1On', 'label2On',
(12)                        'length', 'direction', 'left', 'bottom', 'right', 'top',
(12)                        'labelleft', 'labelbottom', 'labelright', 'labeltop',
(12)                        'labelrotation',
(12)                        *_gridline_param_names]
(8)                     keymap = {
(12)                        'length': 'size',
(12)                        'direction': 'tickdir',
(12)                        'rotation': 'labelrotation',
(12)                        'left': 'tick1On',
(12)                        'bottom': 'tick1On',
(12)                        'right': 'tick2On',
(12)                        'top': 'tick2On',
(12)                        'labelleft': 'label1On',
```

```
878: (12)                        'labelbottom': 'label1On',
879: (12)                        'labelright': 'label2On',
880: (12)                        'labeltop': 'label2On',
881: (8)                    }
882: (8)                    if reverse:
883: (12)                        kwtrans = {
884: (16)                            oldkey: kw_.pop(newkey)
885: (16)                            for oldkey, newkey in keymap.items() if newkey in kw_
886: (12)                        }
887: (8)                    else:
888: (12)                        kwtrans = {
889: (16)                            newkey: kw_.pop(oldkey)
890: (16)                            for oldkey, newkey in keymap.items() if oldkey in kw_
891: (12)                        }
892: (8)                    if 'colors' in kw_:
893: (12)                        c = kw_.pop('colors')
894: (12)                        kwtrans['color'] = c
895: (12)                        kwtrans['labelcolor'] = c
896: (8)                    for key in kw_:
897: (12)                        if key not in allowed_keys:
898: (16)                            raise ValueError(
899: (20)                                "keyword %s is not recognized; valid keywords are %s"
900: (20)                                % (key, allowed_keys))
901: (8)                    kwtrans.update(kw_)
902: (8)                    return kwtrans
903: (4)                @_api.rename_parameter("3.8", "clippath", "path")
904: (4)                def set_clip_path(self, path, transform=None):
905: (8)                    super().set_clip_path(path, transform)
906: (8)                    for child in self.majorTicks + self.minorTicks:
907: (12)                        child.set_clip_path(path, transform)
908: (8)                    self.stale = True
909: (4)                def get_view_interval(self):
910: (8)                    """Return the ``(min, max)`` view limits of this axis."""
911: (8)                    raise NotImplementedError('Derived must override')
912: (4)                def set_view_interval(self, vmin, vmax, ignore=False):
913: (8)                    """
914: (8)                    Set the axis view limits.  This method is for internal use; Matplotlib
915: (8)                    users should typically use e.g. `~.Axes.set_xlim` or
`~.Axes.set_ylim`.
916: (8)                    If *ignore* is False (the default), this method will never reduce the
917: (8)                    preexisting view limits, only expand them if *vmin* or *vmax* are not
918: (8)                    within them.  Moreover, the order of *vmin* and *vmax* does not
matter;
919: (8)                    the orientation of the axis will not change.
920: (8)                    If *ignore* is True, the view limits will be set exactly to ``(vmin,
921: (8)                    vmax)`` in that order.
922: (8)                    """
923: (8)                    raise NotImplementedError('Derived must override')
924: (4)                def get_data_interval(self):
925: (8)                    """Return the ``(min, max)`` data limits of this axis."""
926: (8)                    raise NotImplementedError('Derived must override')
927: (4)                def set_data_interval(self, vmin, vmax, ignore=False):
928: (8)                    """
929: (8)                    Set the axis data limits.  This method is for internal use.
930: (8)                    If *ignore* is False (the default), this method will never reduce the
931: (8)                    preexisting data limits, only expand them if *vmin* or *vmax* are not
932: (8)                    within them.  Moreover, the order of *vmin* and *vmax* does not
matter;
933: (8)                    the orientation of the axis will not change.
934: (8)                    If *ignore* is True, the data limits will be set exactly to ``(vmin,
935: (8)                    vmax)`` in that order.
936: (8)                    """
937: (8)                    raise NotImplementedError('Derived must override')
938: (4)                def get_inverted(self):
939: (8)                    """
940: (8)                    Return whether this Axis is oriented in the "inverse" direction.
941: (8)                    The "normal" direction is increasing to the right for the x-axis and
to
942: (8)                    the top for the y-axis; the "inverse" direction is increasing to the
```

```
943: (8)                    left for the x-axis and to the bottom for the y-axis.
944: (8)                    """
945: (8)                    low, high = self.get_view_interval()
946: (8)                    return high < low
947: (4)                def set_inverted(self, inverted):
948: (8)                    """
949: (8)                    Set whether this Axis is oriented in the "inverse" direction.
950: (8)                    The "normal" direction is increasing to the right for the x-axis and
to
951: (8)                    the top for the y-axis; the "inverse" direction is increasing to the
952: (8)                    left for the x-axis and to the bottom for the y-axis.
953: (8)                    """
954: (8)                    a, b = self.get_view_interval()
955: (8)                    self._set_lim(*sorted((a, b), reverse=bool(inverted)), auto=None)
956: (4)                def set_default_intervals(self):
957: (8)                    """
958: (8)                    Set the default limits for the axis data and view interval if they
959: (8)                    have not been not mutated yet.
960: (8)                    """
961: (4)                def _set_lim(self, v0, v1, *, emit=True, auto):
962: (8)                    """
963: (8)                    Set view limits.
964: (8)                    This method is a helper for the Axes ``set_xlim``, ``set_ylim``, and
965: (8)                    ``set_zlim`` methods.
966: (8)                    Parameters
967: (8)                    ----------
968: (8)                    v0, v1 : float
969: (12)                       The view limits.  (Passing *v0* as a (low, high) pair is not
970: (12)                       supported; normalization must occur in the Axes setters.)
971: (8)                    emit : bool, default: True
972: (12)                       Whether to notify observers of limit change.
973: (8)                    auto : bool or None, default: False
974: (12)                       Whether to turn on autoscaling of the x-axis. True turns on, False
975: (12)                       turns off, None leaves unchanged.
976: (8)                    """
977: (8)                    name = self._get_axis_name()
978: (8)                    self.axes._process_unit_info([(name, (v0, v1))], convert=False)
979: (8)                    v0 = self.axes._validate_converted_limits(v0, self.convert_units)
980: (8)                    v1 = self.axes._validate_converted_limits(v1, self.convert_units)
981: (8)                    if v0 is None or v1 is None:
982: (12)                       old0, old1 = self.get_view_interval()
983: (12)                       if v0 is None:
984: (16)                           v0 = old0
985: (12)                       if v1 is None:
986: (16)                           v1 = old1
987: (8)                    if self.get_scale() == 'log' and (v0 <= 0 or v1 <= 0):
988: (12)                       old0, old1 = self.get_view_interval()
989: (12)                       if v0 <= 0:
990: (16)                           _api.warn_external(f"Attempt to set non-positive {name}lim on
"
991: (35)                                              f"a log-scaled axis will be ignored.")
992: (16)                           v0 = old0
993: (12)                       if v1 <= 0:
994: (16)                           _api.warn_external(f"Attempt to set non-positive {name}lim on
"
995: (35)                                              f"a log-scaled axis will be ignored.")
996: (16)                           v1 = old1
997: (8)                    if v0 == v1:
998: (12)                       _api.warn_external(
999: (16)                           f"Attempting to set identical low and high {name}lims "
1000: (16)                          f"makes transformation singular; automatically expanding.")
1001: (8)                   reverse = bool(v0 > v1)  # explicit cast needed for
python3.8+np.bool_.
1002: (8)                   v0, v1 = self.get_major_locator().nonsingular(v0, v1)
1003: (8)                   v0, v1 = self.limit_range_for_scale(v0, v1)
1004: (8)                   v0, v1 = sorted([v0, v1], reverse=bool(reverse))
1005: (8)                   self.set_view_interval(v0, v1, ignore=True)
1006: (8)                   for ax in self._get_shared_axes():
1007: (12)                      ax._stale_viewlims[name] = False
```

```
(8)                 self._set_autoscale_on(auto)
(8)                 if emit:
(12)                    self.axes.callbacks.process(f"{name}lim_changed", self.axes)
(12)                    for other in self._get_shared_axes():
(16)                        if other is self.axes:
(20)                            continue
(16)                        other._axis_map[name]._set_lim(v0, v1, emit=False, auto=auto)
(16)                        if emit:
(20)                            other.callbacks.process(f"{name}lim_changed", other)
(16)                        if other.figure != self.figure:
(20)                            other.figure.canvas.draw_idle()
(8)                 self.stale = True
(8)                 return v0, v1
(4)             def _set_artist_props(self, a):
(8)                 if a is None:
(12)                    return
(8)                 a.set_figure(self.figure)
(4)             def _update_ticks(self):
(8)                 """
(8)                 Update ticks (position and labels) using the current data interval of
(8)                 the axes.  Return the list of ticks that will be drawn.
(8)                 """
(8)                 major_locs = self.get_majorticklocs()
(8)                 major_labels = self.major.formatter.format_ticks(major_locs)
(8)                 major_ticks = self.get_major_ticks(len(major_locs))
(8)                 for tick, loc, label in zip(major_ticks, major_locs, major_labels):
(12)                    tick.update_position(loc)
(12)                    tick.label1.set_text(label)
(12)                    tick.label2.set_text(label)
(8)                 minor_locs = self.get_minorticklocs()
(8)                 minor_labels = self.minor.formatter.format_ticks(minor_locs)
(8)                 minor_ticks = self.get_minor_ticks(len(minor_locs))
(8)                 for tick, loc, label in zip(minor_ticks, minor_locs, minor_labels):
(12)                    tick.update_position(loc)
(12)                    tick.label1.set_text(label)
(12)                    tick.label2.set_text(label)
(8)                 ticks = [*major_ticks, *minor_ticks]
(8)                 view_low, view_high = self.get_view_interval()
(8)                 if view_low > view_high:
(12)                    view_low, view_high = view_high, view_low
(8)                 if (hasattr(self, "axes") and self.axes.name == '3d'
(16)                        and mpl.rcParams['axes3d.automargin']):
(12)                    margin = 0.019965277777777776
(12)                    delta = view_high - view_low
(12)                    view_high = view_high - delta * margin
(12)                    view_low = view_low + delta * margin
(8)                 interval_t = self.get_transform().transform([view_low, view_high])
(8)                 ticks_to_draw = []
(8)                 for tick in ticks:
(12)                    try:
(16)                        loc_t = self.get_transform().transform(tick.get_loc())
(12)                    except AssertionError:
(16)                        pass
(12)                    else:
(16)                        if mtransforms._interval_contains_close(interval_t, loc_t):
(20)                            ticks_to_draw.append(tick)
(8)                 return ticks_to_draw
(4)             def _get_ticklabel_bboxes(self, ticks, renderer=None):
(8)                 """Return lists of bboxes for ticks' label1's and label2's."""
(8)                 if renderer is None:
(12)                    renderer = self.figure._get_renderer()
(8)                 return ([tick.label1.get_window_extent(renderer)
(17)                         for tick in ticks if tick.label1.get_visible()],
(16)                        [tick.label2.get_window_extent(renderer)
(17)                         for tick in ticks if tick.label2.get_visible()])
(4)             def get_tightbbox(self, renderer=None, *, for_layout_only=False):
(8)                 """
(8)                 Return a bounding box that encloses the axis. It only accounts
(8)                 tick labels, axis label, and offsetText.
```

```
(8)                 If *for_layout_only* is True, then the width of the label (if this
(8)                 is an x-axis) or the height of the label (if this is a y-axis) is
(8)                 collapsed to near zero.  This allows tight/constrained_layout to
ignore
(8)                 too-long labels when doing their layout.
(8)                 """
(8)                 if not self.get_visible():
(12)                    return
(8)                 if renderer is None:
(12)                    renderer = self.figure._get_renderer()
(8)                 ticks_to_draw = self._update_ticks()
(8)                 self._update_label_position(renderer)
(8)                 tlb1, tlb2 = self._get_ticklabel_bboxes(ticks_to_draw, renderer)
(8)                 self._update_offset_text_position(tlb1, tlb2)
(8)                 self.offsetText.set_text(self.major.formatter.get_offset())
(8)                 bboxes = [
(12)                    *(a.get_window_extent(renderer)
(14)                      for a in [self.offsetText]
(14)                      if a.get_visible()),
(12)                    *tlb1, *tlb2,
(8)                 ]
(8)                 if self.label.get_visible():
(12)                    bb = self.label.get_window_extent(renderer)
(12)                    if for_layout_only:
(16)                        if self.axis_name == "x" and bb.width > 0:
(20)                            bb.x0 = (bb.x0 + bb.x1) / 2 - 0.5
(20)                            bb.x1 = bb.x0 + 1.0
(16)                        if self.axis_name == "y" and bb.height > 0:
(20)                            bb.y0 = (bb.y0 + bb.y1) / 2 - 0.5
(20)                            bb.y1 = bb.y0 + 1.0
(12)                    bboxes.append(bb)
(8)                 bboxes = [b for b in bboxes
(18)                          if 0 < b.width < np.inf and 0 < b.height < np.inf]
(8)                 if bboxes:
(12)                    return mtransforms.Bbox.union(bboxes)
(8)                 else:
(12)                    return None
(4)             def get_tick_padding(self):
(8)                 values = []
(8)                 if len(self.majorTicks):
(12)                    values.append(self.majorTicks[0].get_tick_padding())
(8)                 if len(self.minorTicks):
(12)                    values.append(self.minorTicks[0].get_tick_padding())
(8)                 return max(values, default=0)
(4)             @martist.allow_rasterization
(4)             def draw(self, renderer):
(8)                 if not self.get_visible():
(12)                    return
(8)                 renderer.open_group(__name__, gid=self.get_gid())
(8)                 ticks_to_draw = self._update_ticks()
(8)                 tlb1, tlb2 = self._get_ticklabel_bboxes(ticks_to_draw, renderer)
(8)                 for tick in ticks_to_draw:
(12)                    tick.draw(renderer)
(8)                 self._update_label_position(renderer)
(8)                 self.label.draw(renderer)
(8)                 self._update_offset_text_position(tlb1, tlb2)
(8)                 self.offsetText.set_text(self.major.formatter.get_offset())
(8)                 self.offsetText.draw(renderer)
(8)                 renderer.close_group(__name__)
(8)                 self.stale = False
(4)             def get_gridlines(self):
(8)                 r"""Return this Axis' grid lines as a list of `.Line2D`\s."""
(8)                 ticks = self.get_major_ticks()
(8)                 return cbook.silent_list('Line2D gridline',
(33)                                         [tick.gridline for tick in ticks])
(4)             def get_label(self):
(8)                 """Return the axis label as a Text instance."""
(8)                 return self.label
(4)             def get_offset_text(self):
```

```
(8)                 """Return the axis offsetText as a Text instance."""
(8)                 return self.offsetText
(4)             def get_pickradius(self):
(8)                 """Return the depth of the axis used by the picker."""
(8)                 return self._pickradius
(4)             def get_majorticklabels(self):
(8)                 """Return this Axis' major tick labels, as a list of `~.text.Text`."""
(8)                 self._update_ticks()
(8)                 ticks = self.get_major_ticks()
(8)                 labels1 = [tick.label1 for tick in ticks if tick.label1.get_visible()]
(8)                 labels2 = [tick.label2 for tick in ticks if tick.label2.get_visible()]
(8)                 return labels1 + labels2
(4)             def get_minorticklabels(self):
(8)                 """Return this Axis' minor tick labels, as a list of `~.text.Text`."""
(8)                 self._update_ticks()
(8)                 ticks = self.get_minor_ticks()
(8)                 labels1 = [tick.label1 for tick in ticks if tick.label1.get_visible()]
(8)                 labels2 = [tick.label2 for tick in ticks if tick.label2.get_visible()]
(8)                 return labels1 + labels2
(4)             def get_ticklabels(self, minor=False, which=None):
(8)                 """
(8)                 Get this Axis' tick labels.
(8)                 Parameters
(8)                 ----------
(8)                 minor : bool
(11)                   Whether to return the minor or the major ticklabels.
(8)                 which : None, ('minor', 'major', 'both')
(11)                   Overrides *minor*.
(11)                   Selects which ticklabels to return
(8)                 Returns
(8)                 -------
(8)                 list of `~matplotlib.text.Text`
(8)                 """
(8)                 if which is not None:
(12)                    if which == 'minor':
(16)                        return self.get_minorticklabels()
(12)                    elif which == 'major':
(16)                        return self.get_majorticklabels()
(12)                    elif which == 'both':
(16)                        return self.get_majorticklabels() + self.get_minorticklabels()
(12)                    else:
(16)                        _api.check_in_list(['major', 'minor', 'both'], which=which)
(8)                 if minor:
(12)                    return self.get_minorticklabels()
(8)                 return self.get_majorticklabels()
(4)             def get_majorticklines(self):
(8)                 r"""Return this Axis' major tick lines as a list of `.Line2D`\s."""
(8)                 lines = []
(8)                 ticks = self.get_major_ticks()
(8)                 for tick in ticks:
(12)                    lines.append(tick.tick1line)
(12)                    lines.append(tick.tick2line)
(8)                 return cbook.silent_list('Line2D ticklines', lines)
(4)             def get_minorticklines(self):
(8)                 r"""Return this Axis' minor tick lines as a list of `.Line2D`\s."""
(8)                 lines = []
(8)                 ticks = self.get_minor_ticks()
(8)                 for tick in ticks:
(12)                    lines.append(tick.tick1line)
(12)                    lines.append(tick.tick2line)
(8)                 return cbook.silent_list('Line2D ticklines', lines)
(4)             def get_ticklines(self, minor=False):
(8)                 r"""Return this Axis' tick lines as a list of `.Line2D`\s."""
(8)                 if minor:
(12)                    return self.get_minorticklines()
(8)                 return self.get_majorticklines()
(4)             def get_majorticklocs(self):
(8)                 """Return this Axis' major tick locations in data coordinates."""
(8)                 return self.major.locator()
```

```
(4)     def get_minorticklocs(self):
(8)         """Return this Axis' minor tick locations in data coordinates."""
(8)         minor_locs = np.asarray(self.minor.locator())
(8)         if self.remove_overlapping_locs:
(12)            major_locs = self.major.locator()
(12)            transform = self._scale.get_transform()
(12)            tr_minor_locs = transform.transform(minor_locs)
(12)            tr_major_locs = transform.transform(major_locs)
(12)            lo, hi = sorted(transform.transform(self.get_view_interval()))
(12)            tol = (hi - lo) * 1e-5
(12)            mask = np.isclose(tr_minor_locs[:, None], tr_major_locs[None, :],
(30)                              atol=tol, rtol=0).any(axis=1)
(12)            minor_locs = minor_locs[~mask]
(8)         return minor_locs
(4)     def get_ticklocs(self, *, minor=False):
(8)         """
(8)         Return this Axis' tick locations in data coordinates.
(8)         The locations are not clipped to the current axis limits and hence
(8)         may contain locations that are not visible in the output.
(8)         Parameters
(8)         ----------
(8)         minor : bool, default: False
(12)            True to return the minor tick directions,
(12)            False to return the major tick directions.
(8)         Returns
(8)         -------
(8)         array of tick locations
(8)         """
(8)         return self.get_minorticklocs() if minor else self.get_majorticklocs()
(4)     def get_ticks_direction(self, minor=False):
(8)         """
(8)         Return an array of this Axis' tick directions.
(8)         Parameters
(8)         ----------
(8)         minor : bool, default: False
(12)            True to return the minor tick directions,
(12)            False to return the major tick directions.
(8)         Returns
(8)         -------
(8)         array of tick directions
(8)         """
(8)         if minor:
(12)            return np.array(
(16)                [tick._tickdir for tick in self.get_minor_ticks()])
(8)         else:
(12)            return np.array(
(16)                [tick._tickdir for tick in self.get_major_ticks()])
(4)     def _get_tick(self, major):
(8)         """Return the default tick instance."""
(8)         if self._tick_class is None:
(12)            raise NotImplementedError(
(16)                f"The Axis subclass {self.__class__.__name__} must define "
(16)                "_tick_class or reimplement _get_tick()")
(8)         tick_kw = self._major_tick_kw if major else self._minor_tick_kw
(8)         return self._tick_class(self.axes, 0, major=major, **tick_kw)
(4)     def _get_tick_label_size(self, axis_name):
(8)         """
(8)         Return the text size of tick labels for this Axis.
(8)         This is a convenience function to avoid having to create a `Tick` in
(8)         `.get_tick_space`, since it is expensive.
(8)         """
(8)         tick_kw = self._major_tick_kw
(8)         size = tick_kw.get('labelsize',
(27)                           mpl.rcParams[f'{axis_name}tick.labelsize'])
(8)         return mtext.FontProperties(size=size).get_size_in_points()
(4)     def _copy_tick_props(self, src, dest):
(8)         """Copy the properties from *src* tick to *dest* tick."""
(8)         if src is None or dest is None:
(12)            return
```

```
(8)                 dest.label1.update_from(src.label1)
(8)                 dest.label2.update_from(src.label2)
(8)                 dest.tick1line.update_from(src.tick1line)
(8)                 dest.tick2line.update_from(src.tick2line)
(8)                 dest.gridline.update_from(src.gridline)
(8)                 dest.update_from(src)
(8)                 dest._loc = src._loc
(8)                 dest._size = src._size
(8)                 dest._width = src._width
(8)                 dest._base_pad = src._base_pad
(8)                 dest._labelrotation = src._labelrotation
(8)                 dest._zorder = src._zorder
(8)                 dest._tickdir = src._tickdir
(4)             def get_label_text(self):
(8)                 """Get the text of the label."""
(8)                 return self.label.get_text()
(4)             def get_major_locator(self):
(8)                 """Get the locator of the major ticker."""
(8)                 return self.major.locator
(4)             def get_minor_locator(self):
(8)                 """Get the locator of the minor ticker."""
(8)                 return self.minor.locator
(4)             def get_major_formatter(self):
(8)                 """Get the formatter of the major ticker."""
(8)                 return self.major.formatter
(4)             def get_minor_formatter(self):
(8)                 """Get the formatter of the minor ticker."""
(8)                 return self.minor.formatter
(4)             def get_major_ticks(self, numticks=None):
(8)                 r"""
(8)                 Return the list of major `.Tick`\s.
(8)                 .. warning::
(12)                    Ticks are not guaranteed to be persistent. Various operations
(12)                    can create, delete and modify the Tick instances. There is an
(12)                    imminent risk that changes to individual ticks will not
(12)                    survive if you work on the figure further (including also
(12)                    panning/zooming on a displayed figure).
(12)                    Working on the individual ticks is a method of last resort.
(12)                    Use `.set_tick_params` instead if possible.
(8)                 """
(8)                 if numticks is None:
(12)                    numticks = len(self.get_majorticklocs())
(8)                 while len(self.majorTicks) < numticks:
(12)                    tick = self._get_tick(major=True)
(12)                    self.majorTicks.append(tick)
(12)                    self._copy_tick_props(self.majorTicks[0], tick)
(8)                 return self.majorTicks[:numticks]
(4)             def get_minor_ticks(self, numticks=None):
(8)                 r"""
(8)                 Return the list of minor `.Tick`\s.
(8)                 .. warning::
(12)                    Ticks are not guaranteed to be persistent. Various operations
(12)                    can create, delete and modify the Tick instances. There is an
(12)                    imminent risk that changes to individual ticks will not
(12)                    survive if you work on the figure further (including also
(12)                    panning/zooming on a displayed figure).
(12)                    Working on the individual ticks is a method of last resort.
(12)                    Use `.set_tick_params` instead if possible.
(8)                 """
(8)                 if numticks is None:
(12)                    numticks = len(self.get_minorticklocs())
(8)                 while len(self.minorTicks) < numticks:
(12)                    tick = self._get_tick(major=False)
(12)                    self.minorTicks.append(tick)
(12)                    self._copy_tick_props(self.minorTicks[0], tick)
(8)                 return self.minorTicks[:numticks]
(4)             def grid(self, visible=None, which='major', **kwargs):
(8)                 """
(8)                 Configure the grid lines.
```

```
(8)                 Parameters
(8)                 ----------
(8)                 visible : bool or None
(12)                    Whether to show the grid lines.  If any *kwargs* are supplied, it
(12)                    is assumed you want the grid on and *visible* will be set to True.
(12)                    If *visible* is *None* and there are no *kwargs*, this toggles the
(12)                    visibility of the lines.
(8)                 which : {'major', 'minor', 'both'}
(12)                    The grid lines to apply the changes on.
(8)                 **kwargs : `~matplotlib.lines.Line2D` properties
(12)                    Define the line properties of the grid, e.g.::
(16)                        grid(color='r', linestyle='-', linewidth=2)
(8)                 """
(8)                 if kwargs:
(12)                    if visible is None:
(16)                        visible = True
(12)                    elif not visible:  # something false-like but not None
(16)                        _api.warn_external('First parameter to grid() is false, '
(35)                                           'but line properties are supplied. The '
(35)                                           'grid will be enabled.')
(16)                        visible = True
(8)                 which = which.lower()
(8)                 _api.check_in_list(['major', 'minor', 'both'], which=which)
(8)                 gridkw = {f'grid_{name}': value for name, value in kwargs.items()}
(8)                 if which in ['minor', 'both']:
(12)                    gridkw['gridOn'] = (not self._minor_tick_kw['gridOn']
(32)                                        if visible is None else visible)
(12)                    self.set_tick_params(which='minor', **gridkw)
(8)                 if which in ['major', 'both']:
(12)                    gridkw['gridOn'] = (not self._major_tick_kw['gridOn']
(32)                                        if visible is None else visible)
(12)                    self.set_tick_params(which='major', **gridkw)
(8)                 self.stale = True
(4)             def update_units(self, data):
(8)                 """
(8)                 Introspect *data* for units converter and update the
(8)                 ``axis.converter`` instance if necessary. Return *True*
(8)                 if *data* is registered for unit conversion.
(8)                 """
(8)                 converter = munits.registry.get_converter(data)
(8)                 if converter is None:
(12)                    return False
(8)                 neednew = self.converter != converter
(8)                 self.converter = converter
(8)                 default = self.converter.default_units(data, self)
(8)                 if default is not None and self.units is None:
(12)                    self.set_units(default)
(8)                 elif neednew:
(12)                    self._update_axisinfo()
(8)                 self.stale = True
(8)                 return True
(4)             def _update_axisinfo(self):
(8)                 """
(8)                 Check the axis converter for the stored units to see if the
(8)                 axis info needs to be updated.
(8)                 """
(8)                 if self.converter is None:
(12)                    return
(8)                 info = self.converter.axisinfo(self.units, self)
(8)                 if info is None:
(12)                    return
(8)                 if info.majloc is not None and \
(11)                   self.major.locator != info.majloc and self.isDefault_majloc:
(12)                    self.set_major_locator(info.majloc)
(12)                    self.isDefault_majloc = True
(8)                 if info.minloc is not None and \
(11)                   self.minor.locator != info.minloc and self.isDefault_minloc:
(12)                    self.set_minor_locator(info.minloc)
(12)                    self.isDefault_minloc = True
```

```
1421: (8)                    if info.majfmt is not None and \
1422: (11)                      self.major.formatter != info.majfmt and self.isDefault_majfmt:
1423: (12)                       self.set_major_formatter(info.majfmt)
1424: (12)                       self.isDefault_majfmt = True
1425: (8)                    if info.minfmt is not None and \
1426: (11)                      self.minor.formatter != info.minfmt and self.isDefault_minfmt:
1427: (12)                       self.set_minor_formatter(info.minfmt)
1428: (12)                       self.isDefault_minfmt = True
1429: (8)                    if info.label is not None and self.isDefault_label:
1430: (12)                       self.set_label_text(info.label)
1431: (12)                       self.isDefault_label = True
1432: (8)                    self.set_default_intervals()
1433: (4)                def have_units(self):
1434: (8)                    return self.converter is not None or self.units is not None
1435: (4)                def convert_units(self, x):
1436: (8)                    if munits._is_natively_supported(x):
1437: (12)                       return x
1438: (8)                    if self.converter is None:
1439: (12)                       self.converter = munits.registry.get_converter(x)
1440: (8)                    if self.converter is None:
1441: (12)                       return x
1442: (8)                    try:
1443: (12)                       ret = self.converter.convert(x, self.units, self)
1444: (8)                    except Exception as e:
1445: (12)                       raise munits.ConversionError('Failed to convert value(s) to axis '
1446: (41)                                                    f'units: {x!r}') from e
1447: (8)                    return ret
1448: (4)                def set_units(self, u):
1449: (8)                    """
1450: (8)                    Set the units for axis.
1451: (8)                    Parameters
1452: (8)                    ----------
1453: (8)                    u : units tag
1454: (8)                    Notes
1455: (8)                    -----
1456: (8)                    The units of any shared axis will also be updated.
1457: (8)                    """
1458: (8)                    if u == self.units:
1459: (12)                       return
1460: (8)                    for axis in self._get_shared_axis():
1461: (12)                       axis.units = u
1462: (12)                       axis._update_axisinfo()
1463: (12)                       axis.callbacks.process('units')
1464: (12)                       axis.stale = True
1465: (4)                def get_units(self):
1466: (8)                    """Return the units for axis."""
1467: (8)                    return self.units
1468: (4)                def set_label_text(self, label, fontdict=None, **kwargs):
1469: (8)                    """
1470: (8)                    Set the text value of the axis label.
1471: (8)                    Parameters
1472: (8)                    ----------
1473: (8)                    label : str
1474: (12)                       Text string.
1475: (8)                    fontdict : dict
1476: (12)                       Text properties.
1477: (12)                       .. admonition:: Discouraged
1478: (15)                          The use of *fontdict* is discouraged. Parameters should be
passed as
1479: (15)                          individual keyword arguments or using dictionary-unpacking
1480: (15)                          ``set_label_text(..., **fontdict)``.
1481: (8)                    **kwargs
1482: (12)                       Merged into fontdict.
1483: (8)                    """
1484: (8)                    self.isDefault_label = False
1485: (8)                    self.label.set_text(label)
1486: (8)                    if fontdict is not None:
1487: (12)                       self.label.update(fontdict)
1488: (8)                    self.label.update(kwargs)
```

```
1489: (8)                    self.stale = True
1490: (8)                    return self.label
1491: (4)                def set_major_formatter(self, formatter):
1492: (8)                    """
1493: (8)                    Set the formatter of the major ticker.
1494: (8)                    In addition to a `~matplotlib.ticker.Formatter` instance,
1495: (8)                    this also accepts a ``str`` or function.
1496: (8)                    For a ``str`` a `~matplotlib.ticker.StrMethodFormatter` is used.
1497: (8)                    The field used for the value must be labeled ``'x'`` and the field
used
1498: (8)                    for the position must be labeled ``'pos'``.
1499: (8)                    See the  `~matplotlib.ticker.StrMethodFormatter` documentation for
1500: (8)                    more information.
1501: (8)                    For a function, a `~matplotlib.ticker.FuncFormatter` is used.
1502: (8)                    The function must take two inputs (a tick value ``x`` and a
1503: (8)                    position ``pos``), and return a string containing the corresponding
1504: (8)                    tick label.
1505: (8)                    See the  `~matplotlib.ticker.FuncFormatter` documentation for
1506: (8)                    more information.
1507: (8)                    Parameters
1508: (8)                    ----------
1509: (8)                    formatter : `~matplotlib.ticker.Formatter`, ``str``, or function
1510: (8)                    """
1511: (8)                    self._set_formatter(formatter, self.major)
1512: (4)                def set_minor_formatter(self, formatter):
1513: (8)                    """
1514: (8)                    Set the formatter of the minor ticker.
1515: (8)                    In addition to a `~matplotlib.ticker.Formatter` instance,
1516: (8)                    this also accepts a ``str`` or function.
1517: (8)                    See `.Axis.set_major_formatter` for more information.
1518: (8)                    Parameters
1519: (8)                    ----------
1520: (8)                    formatter : `~matplotlib.ticker.Formatter`, ``str``, or function
1521: (8)                    """
1522: (8)                    self._set_formatter(formatter, self.minor)
1523: (4)                def _set_formatter(self, formatter, level):
1524: (8)                    if isinstance(formatter, str):
1525: (12)                       formatter = mticker.StrMethodFormatter(formatter)
1526: (8)                    elif (callable(formatter) and
1527: (14)                         not isinstance(formatter, mticker.TickHelper)):
1528: (12)                       formatter = mticker.FuncFormatter(formatter)
1529: (8)                    else:
1530: (12)                       _api.check_isinstance(mticker.Formatter, formatter=formatter)
1531: (8)                    if (isinstance(formatter, mticker.FixedFormatter)
1532: (16)                          and len(formatter.seq) > 0
1533: (16)                          and not isinstance(level.locator, mticker.FixedLocator)):
1534: (12)                       _api.warn_external('FixedFormatter should only be used together '
1535: (31)                                          'with FixedLocator')
1536: (8)                    if level == self.major:
1537: (12)                       self.isDefault_majfmt = False
1538: (8)                    else:
1539: (12)                       self.isDefault_minfmt = False
1540: (8)                    level.formatter = formatter
1541: (8)                    formatter.set_axis(self)
1542: (8)                    self.stale = True
1543: (4)                def set_major_locator(self, locator):
1544: (8)                    """
1545: (8)                    Set the locator of the major ticker.
1546: (8)                    Parameters
1547: (8)                    ----------
1548: (8)                    locator : `~matplotlib.ticker.Locator`
1549: (8)                    """
1550: (8)                    _api.check_isinstance(mticker.Locator, locator=locator)
1551: (8)                    self.isDefault_majloc = False
1552: (8)                    self.major.locator = locator
1553: (8)                    if self.major.formatter:
1554: (12)                       self.major.formatter._set_locator(locator)
1555: (8)                    locator.set_axis(self)
1556: (8)                    self.stale = True
```

```
(4)     def set_minor_locator(self, locator):
(8)         """
(8)         Set the locator of the minor ticker.
(8)         Parameters
(8)         ----------
(8)         locator : `~matplotlib.ticker.Locator`
(8)         """
(8)         _api.check_isinstance(mticker.Locator, locator=locator)
(8)         self.isDefault_minloc = False
(8)         self.minor.locator = locator
(8)         if self.minor.formatter:
(12)            self.minor.formatter._set_locator(locator)
(8)         locator.set_axis(self)
(8)         self.stale = True
(4)     def set_pickradius(self, pickradius):
(8)         """
(8)         Set the depth of the axis used by the picker.
(8)         Parameters
(8)         ----------
(8)         pickradius : float
(12)            The acceptance radius for containment tests.
(12)            See also `.Axis.contains`.
(8)         """
(8)         if not isinstance(pickradius, Real) or pickradius < 0:
(12)            raise ValueError("pick radius should be a distance")
(8)         self._pickradius = pickradius
(4)     pickradius = property(
(8)         get_pickradius, set_pickradius, doc="The acceptance radius for "
(8)         "containment tests. See also `.Axis.contains`.")
(4)     @staticmethod
(4)     def _format_with_dict(tickd, x, pos):
(8)         return tickd.get(x, "")
(4)     def set_ticklabels(self, labels, *, minor=False, fontdict=None, **kwargs):
(8)         r"""
(8)         [*Discouraged*] Set this Axis' tick labels with list of string labels.
(8)         .. admonition:: Discouraged
(12)            The use of this method is discouraged, because of the dependency
on
(12)            tick positions. In most cases, you'll want to use
(12)            ``Axes.set_[x/y/z]ticks(positions, labels)`` or ``Axis.set_ticks``
(12)            instead.
(12)            If you are using this method, you should always fix the tick
(12)            positions before, e.g. by using `.Axis.set_ticks` or by explicitly
(12)            setting a `~.ticker.FixedLocator`. Otherwise, ticks are free to
(12)            move and the labels may end up in unexpected positions.
(8)         Parameters
(8)         ----------
(8)         labels : sequence of str or of `.Text`\s
(12)            Texts for labeling each tick location in the sequence set by
(12)            `.Axis.set_ticks`; the number of labels must match the number of
locations.
(12)            The labels are used as is, via a `.FixedFormatter` (without
further
(12)            formatting).
(8)         minor : bool
(12)            If True, set minor ticks instead of major ticks.
(8)         fontdict : dict, optional
(12)            .. admonition:: Discouraged
(15)               The use of *fontdict* is discouraged. Parameters should be
passed as
(15)               individual keyword arguments or using dictionary-unpacking
(15)               ``set_ticklabels(..., **fontdict)``.
(12)            A dictionary controlling the appearance of the ticklabels.
(12)            The default *fontdict* is::
(15)               {'fontsize': rcParams['axes.titlesize'],
(16)                'fontweight': rcParams['axes.titleweight'],
(16)                'verticalalignment': 'baseline',
(16)                'horizontalalignment': loc}
(8)         **kwargs
```

```
(12)                        Text properties.
(12)                        .. warning::
(16)                            This only sets the properties of the current ticks, which is
(16)                            only sufficient for static plots.
(16)                            Ticks are not guaranteed to be persistent. Various operations
(16)                            can create, delete and modify the Tick instances. There is an
(16)                            imminent risk that these settings can get lost if you work on
(16)                            the figure further (including also panning/zooming on a
(16)                            displayed figure).
(16)                            Use `.set_tick_params` instead if possible.
(8)                     Returns
(8)                     -------
(8)                     list of `.Text`\s
(12)                        For each tick, includes ``tick.label1`` if it is visible, then
(12)                        ``tick.label2`` if it is visible, in that order.
(8)                     """
(8)                     try:
(12)                        labels = [t.get_text() if hasattr(t, 'get_text') else t
(22)                                  for t in labels]
(8)                     except TypeError:
(12)                        raise TypeError(f"{labels:=} must be a sequence") from None
(8)                     locator = (self.get_minor_locator() if minor
(19)                               else self.get_major_locator())
(8)                     if not labels:
(12)                        formatter = mticker.NullFormatter()
(8)                     elif isinstance(locator, mticker.FixedLocator):
(12)                        if len(locator.locs) != len(labels) and len(labels) != 0:
(16)                            raise ValueError(
(20)                                "The number of FixedLocator locations"
(20)                                f" ({len(locator.locs)}), usually from a call to"
(20)                                " set_ticks, does not match"
(20)                                f" the number of labels ({len(labels)}).")
(12)                        tickd = {loc: lab for loc, lab in zip(locator.locs, labels)}
(12)                        func = functools.partial(self._format_with_dict, tickd)
(12)                        formatter = mticker.FuncFormatter(func)
(8)                     else:
(12)                        _api.warn_external(
(17)                             "set_ticklabels() should only be used with a fixed number of
"
(17)                             "ticks, i.e. after set_ticks() or using a FixedLocator.")
(12)                        formatter = mticker.FixedFormatter(labels)
(8)                     with warnings.catch_warnings():
(12)                        warnings.filterwarnings(
(16)                            "ignore",
(16)                            message="FixedFormatter should only be used together with
FixedLocator")
(12)                        if minor:
(16)                            self.set_minor_formatter(formatter)
(16)                            locs = self.get_minorticklocs()
(16)                            ticks = self.get_minor_ticks(len(locs))
(12)                        else:
(16)                            self.set_major_formatter(formatter)
(16)                            locs = self.get_majorticklocs()
(16)                            ticks = self.get_major_ticks(len(locs))
(8)                     ret = []
(8)                     if fontdict is not None:
(12)                        kwargs.update(fontdict)
(8)                     for pos, (loc, tick) in enumerate(zip(locs, ticks)):
(12)                        tick.update_position(loc)
(12)                        tick_label = formatter(loc, pos)
(12)                        tick.label1.set_text(tick_label)
(12)                        tick.label1._internal_update(kwargs)
(12)                        tick.label2.set_text(tick_label)
(12)                        tick.label2._internal_update(kwargs)
(12)                        if tick.label1.get_visible():
(16)                            ret.append(tick.label1)
(12)                        if tick.label2.get_visible():
(16)                            ret.append(tick.label2)
(8)                     self.stale = True
```

```
(8)                 return ret
(4)             def _set_tick_locations(self, ticks, *, minor=False):
(8)                 ticks = self.convert_units(ticks)
(8)                 locator = mticker.FixedLocator(ticks)  # validate ticks early.
(8)                 if len(ticks):
(12)                    for axis in self._get_shared_axis():
(16)                        axis.set_view_interval(min(ticks), max(ticks))
(8)                 self.axes.stale = True
(8)                 if minor:
(12)                    self.set_minor_locator(locator)
(12)                    return self.get_minor_ticks(len(ticks))
(8)                 else:
(12)                    self.set_major_locator(locator)
(12)                    return self.get_major_ticks(len(ticks))
(4)             def set_ticks(self, ticks, labels=None, *, minor=False, **kwargs):
(8)                 """
(8)                 Set this Axis' tick locations and optionally tick labels.
(8)                 If necessary, the view limits of the Axis are expanded so that all
(8)                 given ticks are visible.
(8)                 Parameters
(8)                 ----------
(8)                 ticks : 1D array-like
(12)                    Array of tick locations (either floats or in axis units). The axis
(12)                    `.Locator` is replaced by a `~.ticker.FixedLocator`.
(12)                    Pass an empty list (``set_ticks([])``) to remove all ticks.
(12)                    Some tick formatters will not label arbitrary tick positions;
(12)                    e.g. log formatters only label decade ticks by default. In
(12)                    such a case you can set a formatter explicitly on the axis
(12)                    using `.Axis.set_major_formatter` or provide formatted
(12)                    *labels* yourself.
(8)                 labels : list of str, optional
(12)                    Tick labels for each location in *ticks*; must have the same
length as
(12)                    *ticks*. If set, the labels are used as is, via a
`.FixedFormatter`.
(12)                    If not set, the labels are generated using the axis tick
`.Formatter`.
(8)                 minor : bool, default: False
(12)                    If ``False``, set only the major ticks; if ``True``, only the
minor ticks.
(8)                 **kwargs
(12)                    `.Text` properties for the labels. Using these is only allowed if
(12)                    you pass *labels*. In other cases, please use
`~.Axes.tick_params`.
(8)                 Notes
(8)                 -----
(8)                 The mandatory expansion of the view limits is an intentional design
(8)                 choice to prevent the surprise of a non-visible tick. If you need
(8)                 other limits, you should set the limits explicitly after setting the
(8)                 ticks.
(8)                 """
(8)                 if labels is None and kwargs:
(12)                    first_key = next(iter(kwargs))
(12)                    raise ValueError(
(16)                        f"Incorrect use of keyword argument {first_key!r}. Keyword
arguments "
(16)                        "other than 'minor' modify the text labels and can only be
used if "
(16)                        "'labels' are passed as well.")
(8)                 result = self._set_tick_locations(ticks, minor=minor)
(8)                 if labels is not None:
(12)                    self.set_ticklabels(labels, minor=minor, **kwargs)
(8)                 return result
(4)             def _get_tick_boxes_siblings(self, renderer):
(8)                 """
(8)                 Get the bounding boxes for this `.axis` and its siblings
(8)                 as set by `.Figure.align_xlabels` or  `.Figure.align_ylabels`.
(8)                 By default, it just gets bboxes for *self*.
(8)                 """
```

```
(8)                 name = self._get_axis_name()
(8)                 if name not in self.figure._align_label_groups:
(12)                    return [], []
(8)                 grouper = self.figure._align_label_groups[name]
(8)                 bboxes = []
(8)                 bboxes2 = []
(8)                 for ax in grouper.get_siblings(self.axes):
(12)                    axis = ax._axis_map[name]
(12)                    ticks_to_draw = axis._update_ticks()
(12)                    tlb, tlb2 = axis._get_ticklabel_bboxes(ticks_to_draw, renderer)
(12)                    bboxes.extend(tlb)
(12)                    bboxes2.extend(tlb2)
(8)                 return bboxes, bboxes2
(4)             def _update_label_position(self, renderer):
(8)                 """
(8)                 Update the label position based on the bounding box enclosing
(8)                 all the ticklabels and axis spine.
(8)                 """
(8)                 raise NotImplementedError('Derived must override')
(4)             def _update_offset_text_position(self, bboxes, bboxes2):
(8)                 """
(8)                 Update the offset text position based on the sequence of bounding
(8)                 boxes of all the ticklabels.
(8)                 """
(8)                 raise NotImplementedError('Derived must override')
(4)             def axis_date(self, tz=None):
(8)                 """
(8)                 Set up axis ticks and labels to treat data along this Axis as dates.
(8)                 Parameters
(8)                 ----------
(8)                 tz : str or `datetime.tzinfo`, default: :rc:`timezone`
(12)                    The timezone used to create date labels.
(8)                 """
(8)                 if isinstance(tz, str):
(12)                    import dateutil.tz
(12)                    tz = dateutil.tz.gettz(tz)
(8)                 self.update_units(datetime.datetime(2009, 1, 1, 0, 0, 0, 0, tz))
(4)             def get_tick_space(self):
(8)                 """Return the estimated number of ticks that can fit on the axis."""
(8)                 raise NotImplementedError()
(4)             def _get_ticks_position(self):
(8)                 """
(8)                 Helper for `XAxis.get_ticks_position` and `YAxis.get_ticks_position`.
(8)                 Check the visibility of tick1line, label1, tick2line, and label2 on
(8)                 the first major and the first minor ticks, and return
(8)                 - 1 if only tick1line and label1 are visible (which corresponds to
(10)                  "bottom" for the x-axis and "left" for the y-axis);
(8)                 - 2 if only tick2line and label2 are visible (which corresponds to
(10)                  "top" for the x-axis and "right" for the y-axis);
(8)                 - "default" if only tick1line, tick2line and label1 are visible;
(8)                 - "unknown" otherwise.
(8)                 """
(8)                 major = self.majorTicks[0]
(8)                 minor = self.minorTicks[0]
(8)                 if all(tick.tick1line.get_visible()
(15)                       and not tick.tick2line.get_visible()
(15)                       and tick.label1.get_visible()
(15)                       and not tick.label2.get_visible()
(15)                       for tick in [major, minor]):
(12)                    return 1
(8)                 elif all(tick.tick2line.get_visible()
(17)                         and not tick.tick1line.get_visible()
(17)                         and tick.label2.get_visible()
(17)                         and not tick.label1.get_visible()
(17)                         for tick in [major, minor]):
(12)                    return 2
(8)                 elif all(tick.tick1line.get_visible()
(17)                         and tick.tick2line.get_visible()
(17)                         and tick.label1.get_visible()
```

```
(17)                         and not tick.label2.get_visible()
(17)                         for tick in [major, minor]):
(12)                     return "default"
(8)                 else:
(12)                     return "unknown"
(4)             def get_label_position(self):
(8)                 """
(8)                 Return the label position (top or bottom)
(8)                 """
(8)                 return self.label_position
(4)             def set_label_position(self, position):
(8)                 """
(8)                 Set the label position (top or bottom)
(8)                 Parameters
(8)                 ----------
(8)                 position : {'top', 'bottom'}
(8)                 """
(8)                 raise NotImplementedError()
(4)             def get_minpos(self):
(8)                 raise NotImplementedError()
(0)         def _make_getset_interval(method_name, lim_name, attr_name):
(4)             """
(4)             Helper to generate ``get_{data,view}_interval`` and
(4)             ``set_{data,view}_interval`` implementations.
(4)             """
(4)             def getter(self):
(8)                 return getattr(getattr(self.axes, lim_name), attr_name)
(4)             def setter(self, vmin, vmax, ignore=False):
(8)                 if ignore:
(12)                    setattr(getattr(self.axes, lim_name), attr_name, (vmin, vmax))
(8)                 else:
(12)                    oldmin, oldmax = getter(self)
(12)                    if oldmin < oldmax:
(16)                        setter(self, min(vmin, vmax, oldmin), max(vmin, vmax, oldmax),
(23)                               ignore=True)
(12)                    else:
(16)                        setter(self, max(vmin, vmax, oldmin), min(vmin, vmax, oldmax),
(23)                               ignore=True)
(8)                 self.stale = True
(4)             getter.__name__ = f"get_{method_name}_interval"
(4)             setter.__name__ = f"set_{method_name}_interval"
(4)             return getter, setter
(0)         class XAxis(Axis):
(4)             __name__ = 'xaxis'
(4)             axis_name = 'x'  #: Read-only name identifying the axis.
(4)             _tick_class = XTick
(4)             def __init__(self, *args, **kwargs):
(8)                 super().__init__(*args, **kwargs)
(8)                 self._init()
(4)             def _init(self):
(8)                 """
(8)                 Initialize the label and offsetText instance values and
(8)                 `label_position` / `offset_text_position`.
(8)                 """
(8)                 self.label.set(
(12)                    x=0.5, y=0,
(12)                    verticalalignment='top', horizontalalignment='center',
(12)                    transform=mtransforms.blended_transform_factory(
(16)                        self.axes.transAxes, mtransforms.IdentityTransform()),
(8)                 )
(8)                 self.label_position = 'bottom'
(8)                 if mpl.rcParams['xtick.labelcolor'] == 'inherit':
(12)                    tick_color = mpl.rcParams['xtick.color']
(8)                 else:
(12)                    tick_color = mpl.rcParams['xtick.labelcolor']
(8)                 self.offsetText.set(
(12)                    x=1, y=0,
(12)                    verticalalignment='top', horizontalalignment='right',
(12)                    transform=mtransforms.blended_transform_factory(
```

```
(16)                        self.axes.transAxes, mtransforms.IdentityTransform()),
(12)                    fontsize=mpl.rcParams['xtick.labelsize'],
(12)                    color=tick_color
(8)                 )
(8)                 self.offset_text_position = 'bottom'
(4)             def contains(self, mouseevent):
(8)                 """Test whether the mouse event occurred in the x-axis."""
(8)                 if self._different_canvas(mouseevent):
(12)                    return False, {}
(8)                 x, y = mouseevent.x, mouseevent.y
(8)                 try:
(12)                    trans = self.axes.transAxes.inverted()
(12)                    xaxes, yaxes = trans.transform((x, y))
(8)                 except ValueError:
(12)                    return False, {}
(8)                 (l, b), (r, t) = self.axes.transAxes.transform([(0, 0), (1, 1)])
(8)                 inaxis = 0 <= xaxes <= 1 and (
(12)                    b - self._pickradius < y < b or
(12)                    t < y < t + self._pickradius)
(8)                 return inaxis, {}
(4)             def set_label_position(self, position):
(8)                 """
(8)                 Set the label position (top or bottom)
(8)                 Parameters
(8)                 ----------
(8)                 position : {'top', 'bottom'}
(8)                 """
(8)                 self.label.set_verticalalignment(_api.check_getitem({
(12)                    'top': 'baseline', 'bottom': 'top',
(8)                 }, position=position))
(8)                 self.label_position = position
(8)                 self.stale = True
(4)             def _update_label_position(self, renderer):
(8)                 """
(8)                 Update the label position based on the bounding box enclosing
(8)                 all the ticklabels and axis spine
(8)                 """
(8)                 if not self._autolabelpos:
(12)                    return
(8)                 bboxes, bboxes2 = self._get_tick_boxes_siblings(renderer=renderer)
(8)                 x, y = self.label.get_position()
(8)                 if self.label_position == 'bottom':
(12)                    try:
(16)                        spine = self.axes.spines['bottom']
(16)                        spinebbox = spine.get_window_extent()
(12)                    except KeyError:
(16)                        spinebbox = self.axes.bbox
(12)                    bbox = mtransforms.Bbox.union(bboxes + [spinebbox])
(12)                    bottom = bbox.y0
(12)                    self.label.set_position(
(16)                        (x, bottom - self.labelpad * self.figure.dpi / 72)
(12)                    )
(8)                 else:
(12)                    try:
(16)                        spine = self.axes.spines['top']
(16)                        spinebbox = spine.get_window_extent()
(12)                    except KeyError:
(16)                        spinebbox = self.axes.bbox
(12)                    bbox = mtransforms.Bbox.union(bboxes2 + [spinebbox])
(12)                    top = bbox.y1
(12)                    self.label.set_position(
(16)                        (x, top + self.labelpad * self.figure.dpi / 72)
(12)                    )
(4)             def _update_offset_text_position(self, bboxes, bboxes2):
(8)                 """
(8)                 Update the offset_text position based on the sequence of bounding
(8)                 boxes of all the ticklabels
(8)                 """
(8)                 x, y = self.offsetText.get_position()
```

```
(8)                 if not hasattr(self, '_tick_position'):
(12)                    self._tick_position = 'bottom'
(8)                 if self._tick_position == 'bottom':
(12)                    if not len(bboxes):
(16)                        bottom = self.axes.bbox.ymin
(12)                    else:
(16)                        bbox = mtransforms.Bbox.union(bboxes)
(16)                        bottom = bbox.y0
(12)                    y = bottom - self.OFFSETTEXTPAD * self.figure.dpi / 72
(8)                 else:
(12)                    if not len(bboxes2):
(16)                        top = self.axes.bbox.ymax
(12)                    else:
(16)                        bbox = mtransforms.Bbox.union(bboxes2)
(16)                        top = bbox.y1
(12)                    y = top + self.OFFSETTEXTPAD * self.figure.dpi / 72
(8)                 self.offsetText.set_position((x, y))
(4)             def set_ticks_position(self, position):
(8)                 """
(8)                 Set the ticks position.
(8)                 Parameters
(8)                 ----------
(8)                 position : {'top', 'bottom', 'both', 'default', 'none'}
(12)                    'both' sets the ticks to appear on both positions, but does not
(12)                    change the tick labels.  'default' resets the tick positions to
(12)                    the default: ticks on both positions, labels at bottom.  'none'
(12)                    can be used if you don't want any ticks. 'none' and 'both'
(12)                    affect only the ticks, not the labels.
(8)                 """
(8)                 if position == 'top':
(12)                    self.set_tick_params(which='both', top=True, labeltop=True,
(33)                                         bottom=False, labelbottom=False)
(12)                    self._tick_position = 'top'
(12)                    self.offsetText.set_verticalalignment('bottom')
(8)                 elif position == 'bottom':
(12)                    self.set_tick_params(which='both', top=False, labeltop=False,
(33)                                         bottom=True, labelbottom=True)
(12)                    self._tick_position = 'bottom'
(12)                    self.offsetText.set_verticalalignment('top')
(8)                 elif position == 'both':
(12)                    self.set_tick_params(which='both', top=True,
(33)                                         bottom=True)
(8)                 elif position == 'none':
(12)                    self.set_tick_params(which='both', top=False,
(33)                                         bottom=False)
(8)                 elif position == 'default':
(12)                    self.set_tick_params(which='both', top=True, labeltop=False,
(33)                                         bottom=True, labelbottom=True)
(12)                    self._tick_position = 'bottom'
(12)                    self.offsetText.set_verticalalignment('top')
(8)                 else:
(12)                    _api.check_in_list(['top', 'bottom', 'both', 'default', 'none'],
(31)                                       position=position)
(8)                 self.stale = True
(4)             def tick_top(self):
(8)                 """
(8)                 Move ticks and ticklabels (if present) to the top of the Axes.
(8)                 """
(8)                 label = True
(8)                 if 'label1On' in self._major_tick_kw:
(12)                    label = (self._major_tick_kw['label1On']
(21)                             or self._major_tick_kw['label2On'])
(8)                 self.set_ticks_position('top')
(8)                 self.set_tick_params(which='both', labeltop=label)
(4)             def tick_bottom(self):
(8)                 """
(8)                 Move ticks and ticklabels (if present) to the bottom of the Axes.
(8)                 """
(8)                 label = True
```

```
(8)                 if 'label1On' in self._major_tick_kw:
(12)                    label = (self._major_tick_kw['label1On']
(21)                             or self._major_tick_kw['label2On'])
(8)                 self.set_ticks_position('bottom')
(8)                 self.set_tick_params(which='both', labelbottom=label)
(4)             def get_ticks_position(self):
(8)                 """
(8)                 Return the ticks position ("top", "bottom", "default", or "unknown").
(8)                 """
(8)                 return {1: "bottom", 2: "top",
(16)                        "default": "default", "unknown": "unknown"}[
(20)                            self._get_ticks_position()]
(4)             get_view_interval, set_view_interval = _make_getset_interval(
(8)                 "view", "viewLim", "intervalx")
(4)             get_data_interval, set_data_interval = _make_getset_interval(
(8)                 "data", "dataLim", "intervalx")
(4)             def get_minpos(self):
(8)                 return self.axes.dataLim.minposx
(4)             def set_default_intervals(self):
(8)                 if (not self.axes.dataLim.mutatedx() and
(16)                        not self.axes.viewLim.mutatedx()):
(12)                    if self.converter is not None:
(16)                        info = self.converter.axisinfo(self.units, self)
(16)                        if info.default_limits is not None:
(20)                            xmin, xmax = self.convert_units(info.default_limits)
(20)                            self.axes.viewLim.intervalx = xmin, xmax
(8)                 self.stale = True
(4)             def get_tick_space(self):
(8)                 ends = mtransforms.Bbox.unit().transformed(
(12)                    self.axes.transAxes - self.figure.dpi_scale_trans)
(8)                 length = ends.width * 72
(8)                 size = self._get_tick_label_size('x') * 3
(8)                 if size > 0:
(12)                    return int(np.floor(length / size))
(8)                 else:
(12)                    return 2**31 - 1
(0)         class YAxis(Axis):
(4)             __name__ = 'yaxis'
(4)             axis_name = 'y'  #: Read-only name identifying the axis.
(4)             _tick_class = YTick
(4)             def __init__(self, *args, **kwargs):
(8)                 super().__init__(*args, **kwargs)
(8)                 self._init()
(4)             def _init(self):
(8)                 """
(8)                 Initialize the label and offsetText instance values and
(8)                 `label_position` / `offset_text_position`.
(8)                 """
(8)                 self.label.set(
(12)                    x=0, y=0.5,
(12)                    verticalalignment='bottom', horizontalalignment='center',
(12)                    rotation='vertical', rotation_mode='anchor',
(12)                    transform=mtransforms.blended_transform_factory(
(16)                        mtransforms.IdentityTransform(), self.axes.transAxes),
(8)                 )
(8)                 self.label_position = 'left'
(8)                 if mpl.rcParams['ytick.labelcolor'] == 'inherit':
(12)                    tick_color = mpl.rcParams['ytick.color']
(8)                 else:
(12)                    tick_color = mpl.rcParams['ytick.labelcolor']
(8)                 self.offsetText.set(
(12)                    x=0, y=0.5,
(12)                    verticalalignment='baseline', horizontalalignment='left',
(12)                    transform=mtransforms.blended_transform_factory(
(16)                        self.axes.transAxes, mtransforms.IdentityTransform()),
(12)                    fontsize=mpl.rcParams['ytick.labelsize'],
(12)                    color=tick_color
(8)                 )
(8)                 self.offset_text_position = 'left'
```

```
(4)     def contains(self, mouseevent):
(8)         if self._different_canvas(mouseevent):
(12)            return False, {}
(8)         x, y = mouseevent.x, mouseevent.y
(8)         try:
(12)            trans = self.axes.transAxes.inverted()
(12)            xaxes, yaxes = trans.transform((x, y))
(8)         except ValueError:
(12)            return False, {}
(8)         (l, b), (r, t) = self.axes.transAxes.transform([(0, 0), (1, 1)])
(8)         inaxis = 0 <= yaxes <= 1 and (
(12)            l - self._pickradius < x < l or
(12)            r < x < r + self._pickradius)
(8)         return inaxis, {}
(4)     def set_label_position(self, position):
(8)         """
(8)         Set the label position (left or right)
(8)         Parameters
(8)         ----------
(8)         position : {'left', 'right'}
(8)         """
(8)         self.label.set_rotation_mode('anchor')
(8)         self.label.set_verticalalignment(_api.check_getitem({
(12)            'left': 'bottom', 'right': 'top',
(8)         }, position=position))
(8)         self.label_position = position
(8)         self.stale = True
(4)     def _update_label_position(self, renderer):
(8)         """
(8)         Update the label position based on the bounding box enclosing
(8)         all the ticklabels and axis spine
(8)         """
(8)         if not self._autolabelpos:
(12)            return
(8)         bboxes, bboxes2 = self._get_tick_boxes_siblings(renderer=renderer)
(8)         x, y = self.label.get_position()
(8)         if self.label_position == 'left':
(12)            try:
(16)                spine = self.axes.spines['left']
(16)                spinebbox = spine.get_window_extent()
(12)            except KeyError:
(16)                spinebbox = self.axes.bbox
(12)            bbox = mtransforms.Bbox.union(bboxes + [spinebbox])
(12)            left = bbox.x0
(12)            self.label.set_position(
(16)                (left - self.labelpad * self.figure.dpi / 72, y)
(12)            )
(8)         else:
(12)            try:
(16)                spine = self.axes.spines['right']
(16)                spinebbox = spine.get_window_extent()
(12)            except KeyError:
(16)                spinebbox = self.axes.bbox
(12)            bbox = mtransforms.Bbox.union(bboxes2 + [spinebbox])
(12)            right = bbox.x1
(12)            self.label.set_position(
(16)                (right + self.labelpad * self.figure.dpi / 72, y)
(12)            )
(4)     def _update_offset_text_position(self, bboxes, bboxes2):
(8)         """
(8)         Update the offset_text position based on the sequence of bounding
(8)         boxes of all the ticklabels
(8)         """
(8)         x, _ = self.offsetText.get_position()
(8)         if 'outline' in self.axes.spines:
(12)            bbox = self.axes.spines['outline'].get_window_extent()
(8)         else:
(12)            bbox = self.axes.bbox
(8)         top = bbox.ymax
```

```
(8)                 self.offsetText.set_position(
(12)                    (x, top + self.OFFSETTEXTPAD * self.figure.dpi / 72)
(8)                 )
(4)             def set_offset_position(self, position):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 position : {'left', 'right'}
(8)                 """
(8)                 x, y = self.offsetText.get_position()
(8)                 x = _api.check_getitem({'left': 0, 'right': 1}, position=position)
(8)                 self.offsetText.set_ha(position)
(8)                 self.offsetText.set_position((x, y))
(8)                 self.stale = True
(4)             def set_ticks_position(self, position):
(8)                 """
(8)                 Set the ticks position.
(8)                 Parameters
(8)                 ----------
(8)                 position : {'left', 'right', 'both', 'default', 'none'}
(12)                    'both' sets the ticks to appear on both positions, but does not
(12)                    change the tick labels.  'default' resets the tick positions to
(12)                    the default: ticks on both positions, labels at left.  'none'
(12)                    can be used if you don't want any ticks. 'none' and 'both'
(12)                    affect only the ticks, not the labels.
(8)                 """
(8)                 if position == 'right':
(12)                    self.set_tick_params(which='both', right=True, labelright=True,
(33)                                         left=False, labelleft=False)
(12)                    self.set_offset_position(position)
(8)                 elif position == 'left':
(12)                    self.set_tick_params(which='both', right=False, labelright=False,
(33)                                         left=True, labelleft=True)
(12)                    self.set_offset_position(position)
(8)                 elif position == 'both':
(12)                    self.set_tick_params(which='both', right=True,
(33)                                         left=True)
(8)                 elif position == 'none':
(12)                    self.set_tick_params(which='both', right=False,
(33)                                         left=False)
(8)                 elif position == 'default':
(12)                    self.set_tick_params(which='both', right=True, labelright=False,
(33)                                         left=True, labelleft=True)
(8)                 else:
(12)                    _api.check_in_list(['left', 'right', 'both', 'default', 'none'],
(31)                                       position=position)
(8)                 self.stale = True
(4)             def tick_right(self):
(8)                 """
(8)                 Move ticks and ticklabels (if present) to the right of the Axes.
(8)                 """
(8)                 label = True
(8)                 if 'label1On' in self._major_tick_kw:
(12)                    label = (self._major_tick_kw['label1On']
(21)                             or self._major_tick_kw['label2On'])
(8)                 self.set_ticks_position('right')
(8)                 self.set_tick_params(which='both', labelright=label)
(4)             def tick_left(self):
(8)                 """
(8)                 Move ticks and ticklabels (if present) to the left of the Axes.
(8)                 """
(8)                 label = True
(8)                 if 'label1On' in self._major_tick_kw:
(12)                    label = (self._major_tick_kw['label1On']
(21)                             or self._major_tick_kw['label2On'])
(8)                 self.set_ticks_position('left')
(8)                 self.set_tick_params(which='both', labelleft=label)
(4)             def get_ticks_position(self):
(8)                 """
```

```
2234: (8)                   Return the ticks position ("left", "right", "default", or "unknown").
2235: (8)                   """
2236: (8)                   return {1: "left", 2: "right",
2237: (16)                          "default": "default", "unknown": "unknown"}[
2238: (20)                              self._get_ticks_position()]
2239: (4)               get_view_interval, set_view_interval = _make_getset_interval(
2240: (8)                   "view", "viewLim", "intervaly")
2241: (4)               get_data_interval, set_data_interval = _make_getset_interval(
2242: (8)                   "data", "dataLim", "intervaly")
2243: (4)               def get_minpos(self):
2244: (8)                   return self.axes.dataLim.minposy
2245: (4)               def set_default_intervals(self):
2246: (8)                   if (not self.axes.dataLim.mutatedy() and
2247: (16)                          not self.axes.viewLim.mutatedy()):
2248: (12)                      if self.converter is not None:
2249: (16)                          info = self.converter.axisinfo(self.units, self)
2250: (16)                          if info.default_limits is not None:
2251: (20)                              ymin, ymax = self.convert_units(info.default_limits)
2252: (20)                              self.axes.viewLim.intervaly = ymin, ymax
2253: (8)                   self.stale = True
2254: (4)               def get_tick_space(self):
2255: (8)                   ends = mtransforms.Bbox.unit().transformed(
2256: (12)                      self.axes.transAxes - self.figure.dpi_scale_trans)
2257: (8)                   length = ends.height * 72
2258: (8)                   size = self._get_tick_label_size('y') * 2
2259: (8)                   if size > 0:
2260: (12)                      return int(np.floor(length / size))
2261: (8)                   else:
2262: (12)                      return 2**31 - 1
```

File 24 - backend_bases.py:

```
1: (0)              """
2: (0)              Abstract base classes define the primitives that renderers and
3: (0)              graphics contexts must implement to serve as a Matplotlib backend.
4: (0)              `RendererBase`
5: (4)                  An abstract base class to handle drawing/rendering operations.
6: (0)              `FigureCanvasBase`
7: (4)                  The abstraction layer that separates the `.Figure` from the backend
8: (4)                  specific details like a user interface drawing area.
9: (0)              `GraphicsContextBase`
10: (4)                 An abstract base class that provides color, line styles, etc.
11: (0)             `Event`
12: (4)                 The base class for all of the Matplotlib event handling.  Derived classes
13: (4)                 such as `KeyEvent` and `MouseEvent` store the meta data like keys and
14: (4)                 buttons pressed, x and y locations in pixel and `~.axes.Axes` coordinates.
15: (0)             `ShowBase`
16: (4)                 The base class for the ``Show`` class of each interactive backend; the
17: (4)                 'show' callable is then set to ``Show.__call__``.
18: (0)             `ToolContainerBase`
19: (4)                 The base class for the Toolbar class of each interactive backend.
20: (0)             """
21: (0)             from collections import namedtuple
22: (0)             from contextlib import ExitStack, contextmanager, nullcontext
23: (0)             from enum import Enum, IntEnum
24: (0)             import functools
25: (0)             import importlib
26: (0)             import inspect
27: (0)             import io
28: (0)             import itertools
29: (0)             import logging
30: (0)             import os
31: (0)             import pathlib
32: (0)             import signal
33: (0)             import socket
34: (0)             import sys
35: (0)             import time
```

```
36: (0)             import weakref
37: (0)             from weakref import WeakKeyDictionary
38: (0)             import numpy as np
39: (0)             import matplotlib as mpl
40: (0)             from matplotlib import (
41: (4)                 _api, backend_tools as tools, cbook, colors, _docstring, text,
42: (4)                 _tight_bbox, transforms, widgets, is_interactive, rcParams)
43: (0)             from matplotlib._pylab_helpers import Gcf
44: (0)             from matplotlib.backend_managers import ToolManager
45: (0)             from matplotlib.cbook import _setattr_cm
46: (0)             from matplotlib.layout_engine import ConstrainedLayoutEngine
47: (0)             from matplotlib.path import Path
48: (0)             from matplotlib.texmanager import TexManager
49: (0)             from matplotlib.transforms import Affine2D
50: (0)             from matplotlib._enums import JoinStyle, CapStyle
51: (0)             _log = logging.getLogger(__name__)
52: (0)             _default_filetypes = {
53: (4)                 'eps': 'Encapsulated Postscript',
54: (4)                 'jpg': 'Joint Photographic Experts Group',
55: (4)                 'jpeg': 'Joint Photographic Experts Group',
56: (4)                 'pdf': 'Portable Document Format',
57: (4)                 'pgf': 'PGF code for LaTeX',
58: (4)                 'png': 'Portable Network Graphics',
59: (4)                 'ps': 'Postscript',
60: (4)                 'raw': 'Raw RGBA bitmap',
61: (4)                 'rgba': 'Raw RGBA bitmap',
62: (4)                 'svg': 'Scalable Vector Graphics',
63: (4)                 'svgz': 'Scalable Vector Graphics',
64: (4)                 'tif': 'Tagged Image File Format',
65: (4)                 'tiff': 'Tagged Image File Format',
66: (4)                 'webp': 'WebP Image Format',
67: (0)             }
68: (0)             _default_backends = {
69: (4)                 'eps': 'matplotlib.backends.backend_ps',
70: (4)                 'jpg': 'matplotlib.backends.backend_agg',
71: (4)                 'jpeg': 'matplotlib.backends.backend_agg',
72: (4)                 'pdf': 'matplotlib.backends.backend_pdf',
73: (4)                 'pgf': 'matplotlib.backends.backend_pgf',
74: (4)                 'png': 'matplotlib.backends.backend_agg',
75: (4)                 'ps': 'matplotlib.backends.backend_ps',
76: (4)                 'raw': 'matplotlib.backends.backend_agg',
77: (4)                 'rgba': 'matplotlib.backends.backend_agg',
78: (4)                 'svg': 'matplotlib.backends.backend_svg',
79: (4)                 'svgz': 'matplotlib.backends.backend_svg',
80: (4)                 'tif': 'matplotlib.backends.backend_agg',
81: (4)                 'tiff': 'matplotlib.backends.backend_agg',
82: (4)                 'webp': 'matplotlib.backends.backend_agg',
83: (0)             }
84: (0)             def register_backend(format, backend, description=None):
85: (4)                 """
86: (4)                 Register a backend for saving to a given file format.
87: (4)                 Parameters
88: (4)                 ----------
89: (4)                 format : str
90: (8)                     File extension
91: (4)                 backend : module string or canvas class
92: (8)                     Backend for handling file output
93: (4)                 description : str, default: ""
94: (8)                     Description of the file type.
95: (4)                 """
96: (4)                 if description is None:
97: (8)                     description = ''
98: (4)                 _default_backends[format] = backend
99: (4)                 _default_filetypes[format] = description
100: (0)            def get_registered_canvas_class(format):
101: (4)                """
102: (4)                Return the registered default canvas for given file format.
103: (4)                Handles deferred import of required backend.
104: (4)                """
```

```
105: (4)                if format not in _default_backends:
106: (8)                    return None
107: (4)                backend_class = _default_backends[format]
108: (4)                if isinstance(backend_class, str):
109: (8)                    backend_class = importlib.import_module(backend_class).FigureCanvas
110: (8)                    _default_backends[format] = backend_class
111: (4)                return backend_class
112: (0)            class RendererBase:
113: (4)                """
114: (4)                An abstract base class to handle drawing/rendering operations.
115: (4)                The following methods must be implemented in the backend for full
116: (4)                functionality (though just implementing `draw_path` alone would give a
117: (4)                highly capable backend):
118: (4)                * `draw_path`
119: (4)                * `draw_image`
120: (4)                * `draw_gouraud_triangles`
121: (4)                The following methods *should* be implemented in the backend for
122: (4)                optimization reasons:
123: (4)                * `draw_text`
124: (4)                * `draw_markers`
125: (4)                * `draw_path_collection`
126: (4)                * `draw_quad_mesh`
127: (4)                """
128: (4)                def __init__(self):
129: (8)                    super().__init__()
130: (8)                    self._texmanager = None
131: (8)                    self._text2path = text.TextToPath()
132: (8)                    self._raster_depth = 0
133: (8)                    self._rasterizing = False
134: (4)                def open_group(self, s, gid=None):
135: (8)                    """
136: (8)                    Open a grouping element with label *s* and *gid* (if set) as id.
137: (8)                    Only used by the SVG renderer.
138: (8)                    """
139: (4)                def close_group(self, s):
140: (8)                    """
141: (8)                    Close a grouping element with label *s*.
142: (8)                    Only used by the SVG renderer.
143: (8)                    """
144: (4)                def draw_path(self, gc, path, transform, rgbFace=None):
145: (8)                    """Draw a `~.path.Path` instance using the given affine transform."""
146: (8)                    raise NotImplementedError
147: (4)                def draw_markers(self, gc, marker_path, marker_trans, path,
148: (21)                                 trans, rgbFace=None):
149: (8)                    """
150: (8)                    Draw a marker at each of *path*'s vertices (excluding control points).
151: (8)                    The base (fallback) implementation makes multiple calls to
`draw_path`.
152: (8)                    Backends may want to override this method in order to draw the marker
153: (8)                    only once and reuse it multiple times.
154: (8)                    Parameters
155: (8)                    ----------
156: (8)                    gc : `.GraphicsContextBase`
157: (12)                       The graphics context.
158: (8)                    marker_path : `~matplotlib.path.Path`
159: (12)                       The path for the marker.
160: (8)                    marker_trans : `~matplotlib.transforms.Transform`
161: (12)                       An affine transform applied to the marker.
162: (8)                    path : `~matplotlib.path.Path`
163: (12)                       The locations to draw the markers.
164: (8)                    trans : `~matplotlib.transforms.Transform`
165: (12)                       An affine transform applied to the path.
166: (8)                    rgbFace : :mpltype:`color`, optional
167: (8)                    """
168: (8)                    for vertices, codes in path.iter_segments(trans, simplify=False):
169: (12)                       if len(vertices):
170: (16)                           x, y = vertices[-2:]
171: (16)                           self.draw_path(gc, marker_path,
172: (31)                                          marker_trans +
```

```
173: (31)                                        transforms.Affine2D().translate(x, y),
174: (31)                                        rgbFace)
175: (4)                 def draw_path_collection(self, gc, master_transform, paths,
all_transforms,
176: (29)                                       offsets, offset_trans, facecolors, edgecolors,
177: (29)                                       linewidths, linestyles, antialiaseds, urls,
178: (29)                                       offset_position):
179: (8)                     """
180: (8)                     Draw a collection of *paths*.
181: (8)                     Each path is first transformed by the corresponding entry
182: (8)                     in *all_transforms* (a list of (3, 3) matrices) and then by
183: (8)                     *master_transform*.  They are then translated by the corresponding
184: (8)                     entry in *offsets*, which has been first transformed by
*offset_trans*.
185: (8)                     *facecolors*, *edgecolors*, *linewidths*, *linestyles*, and
186: (8)                     *antialiased* are lists that set the corresponding properties.
187: (8)                     *offset_position* is unused now, but the argument is kept for
188: (8)                     backwards compatibility.
189: (8)                     The base (fallback) implementation makes multiple calls to
`draw_path`.
190: (8)                     Backends may want to override this in order to render each set of
191: (8)                     path data only once, and then reference that path multiple times with
192: (8)                     the different offsets, colors, styles etc.  The generator methods
193: (8)                     `_iter_collection_raw_paths` and `_iter_collection` are provided to
194: (8)                     help with (and standardize) the implementation across backends.  It
195: (8)                     is highly recommended to use those generators, so that changes to the
196: (8)                     behavior of `draw_path_collection` can be made globally.
197: (8)                     """
198: (8)                     path_ids = self._iter_collection_raw_paths(master_transform,
199: (51)                                                               paths, all_transforms)
200: (8)                     for xo, yo, path_id, gc0, rgbFace in self._iter_collection(
201: (16)                            gc, list(path_ids), offsets, offset_trans,
202: (16)                            facecolors, edgecolors, linewidths, linestyles,
203: (16)                            antialiaseds, urls, offset_position):
204: (12)                        path, transform = path_id
205: (12)                        if xo != 0 or yo != 0:
206: (16)                            transform = transform.frozen()
207: (16)                            transform.translate(xo, yo)
208: (12)                        self.draw_path(gc0, path, transform, rgbFace)
209: (4)                 def draw_quad_mesh(self, gc, master_transform, meshWidth, meshHeight,
210: (23)                                   coordinates, offsets, offsetTrans, facecolors,
211: (23)                                   antialiased, edgecolors):
212: (8)                     """
213: (8)                     Draw a quadmesh.
214: (8)                     The base (fallback) implementation converts the quadmesh to paths and
215: (8)                     then calls `draw_path_collection`.
216: (8)                     """
217: (8)                     from matplotlib.collections import QuadMesh
218: (8)                     paths = QuadMesh._convert_mesh_to_paths(coordinates)
219: (8)                     if edgecolors is None:
220: (12)                        edgecolors = facecolors
221: (8)                     linewidths = np.array([gc.get_linewidth()], float)
222: (8)                     return self.draw_path_collection(
223: (12)                        gc, master_transform, paths, [], offsets, offsetTrans, facecolors,
224: (12)                        edgecolors, linewidths, [], [antialiased], [None], 'screen')
225: (4)                 def draw_gouraud_triangles(self, gc, triangles_array, colors_array,
226: (31)                                           transform):
227: (8)                     """
228: (8)                     Draw a series of Gouraud triangles.
229: (8)                     Parameters
230: (8)                     ----------
231: (8)                     gc : `.GraphicsContextBase`
232: (12)                        The graphics context.
233: (8)                     triangles_array : (N, 3, 2) array-like
234: (12)                        Array of *N* (x, y) points for the triangles.
235: (8)                     colors_array : (N, 3, 4) array-like
236: (12)                        Array of *N* RGBA colors for each point of the triangles.
237: (8)                     transform : `~matplotlib.transforms.Transform`
238: (12)                        An affine transform to apply to the points.
```

```
(8)                 """
(8)                 raise NotImplementedError
(4)             def _iter_collection_raw_paths(self, master_transform, paths,
(35)                                           all_transforms):
(8)                 """
(8)                 Helper method (along with `_iter_collection`) to implement
(8)                 `draw_path_collection` in a memory-efficient manner.
(8)                 This method yields all of the base path/transform combinations, given
a
(8)                 master transform, a list of paths and list of transforms.
(8)                 The arguments should be exactly what is passed in to
(8)                 `draw_path_collection`.
(8)                 The backend should take each yielded path and transform and create an
(8)                 object that can be referenced (reused) later.
(8)                 """
(8)                 Npaths = len(paths)
(8)                 Ntransforms = len(all_transforms)
(8)                 N = max(Npaths, Ntransforms)
(8)                 if Npaths == 0:
(12)                    return
(8)                 transform = transforms.IdentityTransform()
(8)                 for i in range(N):
(12)                    path = paths[i % Npaths]
(12)                    if Ntransforms:
(16)                        transform = Affine2D(all_transforms[i % Ntransforms])
(12)                    yield path, transform + master_transform
(4)             def _iter_collection_uses_per_path(self, paths, all_transforms,
(39)                                               offsets, facecolors, edgecolors):
(8)                 """
(8)                 Compute how many times each raw path object returned by
(8)                 `_iter_collection_raw_paths` would be used when calling
(8)                 `_iter_collection`. This is intended for the backend to decide
(8)                 on the tradeoff between using the paths in-line and storing
(8)                 them once and reusing. Rounds up in case the number of uses
(8)                 is not the same for every path.
(8)                 """
(8)                 Npaths = len(paths)
(8)                 if Npaths == 0 or len(facecolors) == len(edgecolors) == 0:
(12)                    return 0
(8)                 Npath_ids = max(Npaths, len(all_transforms))
(8)                 N = max(Npath_ids, len(offsets))
(8)                 return (N + Npath_ids - 1) // Npath_ids
(4)             def _iter_collection(self, gc, path_ids, offsets, offset_trans,
facecolors,
(25)                                 edgecolors, linewidths, linestyles,
(25)                                 antialiaseds, urls, offset_position):
(8)                 """
(8)                 Helper method (along with `_iter_collection_raw_paths`) to implement
(8)                 `draw_path_collection` in a memory-efficient manner.
(8)                 This method yields all of the path, offset and graphics context
(8)                 combinations to draw the path collection.  The caller should already
(8)                 have looped over the results of `_iter_collection_raw_paths` to draw
(8)                 this collection.
(8)                 The arguments should be the same as that passed into
(8)                 `draw_path_collection`, with the exception of *path_ids*, which is a
(8)                 list of arbitrary objects that the backend will use to reference one
of
(8)                 the paths created in the `_iter_collection_raw_paths` stage.
(8)                 Each yielded result is of the form::
(11)                   xo, yo, path_id, gc, rgbFace
(8)                 where *xo*, *yo* is an offset; *path_id* is one of the elements of
(8)                 *path_ids*; *gc* is a graphics context and *rgbFace* is a color to
(8)                 use for filling the path.
(8)                 """
(8)                 Npaths = len(path_ids)
(8)                 Noffsets = len(offsets)
(8)                 N = max(Npaths, Noffsets)
(8)                 Nfacecolors = len(facecolors)
(8)                 Nedgecolors = len(edgecolors)
```

```
(8)                 Nlinewidths = len(linewidths)
(8)                 Nlinestyles = len(linestyles)
(8)                 Nurls = len(urls)
(8)                 if (Nfacecolors == 0 and Nedgecolors == 0) or Npaths == 0:
(12)                    return
(8)                 gc0 = self.new_gc()
(8)                 gc0.copy_properties(gc)
(8)                 def cycle_or_default(seq, default=None):
(12)                    return (itertools.cycle(seq) if len(seq)
(20)                            else itertools.repeat(default))
(8)                 pathids = cycle_or_default(path_ids)
(8)                 toffsets = cycle_or_default(offset_trans.transform(offsets), (0, 0))
(8)                 fcs = cycle_or_default(facecolors)
(8)                 ecs = cycle_or_default(edgecolors)
(8)                 lws = cycle_or_default(linewidths)
(8)                 lss = cycle_or_default(linestyles)
(8)                 aas = cycle_or_default(antialiaseds)
(8)                 urls = cycle_or_default(urls)
(8)                 if Nedgecolors == 0:
(12)                    gc0.set_linewidth(0.0)
(8)                 for pathid, (xo, yo), fc, ec, lw, ls, aa, url in itertools.islice(
(16)                        zip(pathids, toffsets, fcs, ecs, lws, lss, aas, urls), N):
(12)                    if not (np.isfinite(xo) and np.isfinite(yo)):
(16)                        continue
(12)                    if Nedgecolors:
(16)                        if Nlinewidths:
(20)                            gc0.set_linewidth(lw)
(16)                        if Nlinestyles:
(20)                            gc0.set_dashes(*ls)
(16)                        if len(ec) == 4 and ec[3] == 0.0:
(20)                            gc0.set_linewidth(0)
(16)                        else:
(20)                            gc0.set_foreground(ec)
(12)                    if fc is not None and len(fc) == 4 and fc[3] == 0:
(16)                        fc = None
(12)                    gc0.set_antialiased(aa)
(12)                    if Nurls:
(16)                        gc0.set_url(url)
(12)                    yield xo, yo, pathid, gc0, fc
(8)                 gc0.restore()
(4)             def get_image_magnification(self):
(8)                 """
(8)                 Get the factor by which to magnify images passed to `draw_image`.
(8)                 Allows a backend to have images at a different resolution to other
(8)                 artists.
(8)                 """
(8)                 return 1.0
(4)             def draw_image(self, gc, x, y, im, transform=None):
(8)                 """
(8)                 Draw an RGBA image.
(8)                 Parameters
(8)                 ----------
(8)                 gc : `.GraphicsContextBase`
(12)                    A graphics context with clipping information.
(8)                 x : scalar
(12)                    The distance in physical units (i.e., dots or pixels) from the
left
(12)                    hand side of the canvas.
(8)                 y : scalar
(12)                    The distance in physical units (i.e., dots or pixels) from the
(12)                    bottom side of the canvas.
(8)                 im : (N, M, 4) array of `numpy.uint8`
(12)                    An array of RGBA pixels.
(8)                 transform : `~matplotlib.transforms.Affine2DBase`
(12)                    If and only if the concrete backend is written such that
(12)                    `option_scale_image` returns ``True``, an affine transformation
(12)                    (i.e., an `.Affine2DBase`) *may* be passed to `draw_image`.  The
(12)                    translation vector of the transformation is given in physical
units
```

```
(12)                        (i.e., dots or pixels). Note that the transformation does not
(12)                        override *x* and *y*, and has to be applied *before* translating
(12)                        the result by *x* and *y* (this can be accomplished by adding *x*
(12)                        and *y* to the translation vector defined by *transform*).
(8)                     """
(8)                     raise NotImplementedError
(4)                 def option_image_nocomposite(self):
(8)                     """
(8)                     Return whether image composition by Matplotlib should be skipped.
(8)                     Raster backends should usually return False (letting the C-level
(8)                     rasterizer take care of image composition); vector backends should
(8)                     usually return ``not rcParams["image.composite_image"]``.
(8)                     """
(8)                     return False
(4)                 def option_scale_image(self):
(8)                     """
(8)                     Return whether arbitrary affine transformations in `draw_image` are
(8)                     supported (True for most vector backends).
(8)                     """
(8)                     return False
(4)                 def draw_tex(self, gc, x, y, s, prop, angle, *, mtext=None):
(8)                     """
(8)                     Draw a TeX instance.
(8)                     Parameters
(8)                     ----------
(8)                     gc : `.GraphicsContextBase`
(12)                        The graphics context.
(8)                     x : float
(12)                        The x location of the text in display coords.
(8)                     y : float
(12)                        The y location of the text baseline in display coords.
(8)                     s : str
(12)                        The TeX text string.
(8)                     prop : `~matplotlib.font_manager.FontProperties`
(12)                        The font properties.
(8)                     angle : float
(12)                        The rotation angle in degrees anti-clockwise.
(8)                     mtext : `~matplotlib.text.Text`
(12)                        The original text object to be rendered.
(8)                     """
(8)                     self._draw_text_as_path(gc, x, y, s, prop, angle, ismath="TeX")
(4)                 def draw_text(self, gc, x, y, s, prop, angle, ismath=False, mtext=None):
(8)                     """
(8)                     Draw a text instance.
(8)                     Parameters
(8)                     ----------
(8)                     gc : `.GraphicsContextBase`
(12)                        The graphics context.
(8)                     x : float
(12)                        The x location of the text in display coords.
(8)                     y : float
(12)                        The y location of the text baseline in display coords.
(8)                     s : str
(12)                        The text string.
(8)                     prop : `~matplotlib.font_manager.FontProperties`
(12)                        The font properties.
(8)                     angle : float
(12)                        The rotation angle in degrees anti-clockwise.
(8)                     ismath : bool or "TeX"
(12)                        If True, use mathtext parser. If "TeX", use tex for rendering.
(8)                     mtext : `~matplotlib.text.Text`
(12)                        The original text object to be rendered.
(8)                     """
(8)                     self._draw_text_as_path(gc, x, y, s, prop, angle, ismath)
(4)                 def _get_text_path_transform(self, x, y, s, prop, angle, ismath):
(8)                     """
(8)                     Return the text path and transform.
(8)                     Parameters
(8)                     ----------
```

```
        x : float
            The x location of the text in display coords.
        y : float
            The y location of the text baseline in display coords.
        s : str
            The text to be converted.
        prop : `~matplotlib.font_manager.FontProperties`
            The font property.
        angle : float
            Angle in degrees to render the text at.
        ismath : bool or "TeX"
            If True, use mathtext parser. If "TeX", use tex for rendering.
        """
        text2path = self._text2path
        fontsize = self.points_to_pixels(prop.get_size_in_points())
        verts, codes = text2path.get_text_path(prop, s, ismath=ismath)
        path = Path(verts, codes)
        angle = np.deg2rad(angle)
        if self.flipy():
            width, height = self.get_canvas_width_height()
            transform = (Affine2D()
                         .scale(fontsize / text2path.FONT_SCALE)
                         .rotate(angle)
                         .translate(x, height - y))
        else:
            transform = (Affine2D()
                         .scale(fontsize / text2path.FONT_SCALE)
                         .rotate(angle)
                         .translate(x, y))
        return path, transform
    def _draw_text_as_path(self, gc, x, y, s, prop, angle, ismath):
        """
        Draw the text by converting them to paths using `.TextToPath`.
        Parameters
        ----------
        gc : `.GraphicsContextBase`
            The graphics context.
        x : float
            The x location of the text in display coords.
        y : float
            The y location of the text baseline in display coords.
        s : str
            The text to be converted.
        prop : `~matplotlib.font_manager.FontProperties`
            The font property.
        angle : float
            Angle in degrees to render the text at.
        ismath : bool or "TeX"
            If True, use mathtext parser. If "TeX", use tex for rendering.
        """
        path, transform = self._get_text_path_transform(
            x, y, s, prop, angle, ismath)
        color = gc.get_rgb()
        gc.set_linewidth(0.0)
        self.draw_path(gc, path, transform, rgbFace=color)
    def get_text_width_height_descent(self, s, prop, ismath):
        """
        Get the width, height, and descent (offset from the bottom to the
baseline), in
        display coords, of the string *s* with `.FontProperties` *prop*.
        Whitespace at the start and the end of *s* is included in the reported
width.
        """
        fontsize = prop.get_size_in_points()
        if ismath == 'TeX':
            return self.get_texmanager().get_text_width_height_descent(
                s, fontsize, renderer=self)
        dpi = self.points_to_pixels(72)
        if ismath:
```

```
(12)                        dims = self._text2path.mathtext_parser.parse(s, dpi, prop)
(12)                        return dims[0:3]  # return width, height, descent
(8)                     flags = self._text2path._get_hinting_flag()
(8)                     font = self._text2path._get_font(prop)
(8)                     font.set_size(fontsize, dpi)
(8)                     font.set_text(s, 0.0, flags=flags)
(8)                     w, h = font.get_width_height()
(8)                     d = font.get_descent()
(8)                     w /= 64.0  # convert from subpixels
(8)                     h /= 64.0
(8)                     d /= 64.0
(8)                     return w, h, d
(4)                 def flipy(self):
(8)                     """
(8)                     Return whether y values increase from top to bottom.
(8)                     Note that this only affects drawing of texts.
(8)                     """
(8)                     return True
(4)                 def get_canvas_width_height(self):
(8)                     """Return the canvas width and height in display coords."""
(8)                     return 1, 1
(4)                 def get_texmanager(self):
(8)                     """Return the `.TexManager` instance."""
(8)                     if self._texmanager is None:
(12)                        self._texmanager = TexManager()
(8)                     return self._texmanager
(4)                 def new_gc(self):
(8)                     """Return an instance of a `.GraphicsContextBase`."""
(8)                     return GraphicsContextBase()
(4)                 def points_to_pixels(self, points):
(8)                     """
(8)                     Convert points to display units.
(8)                     You need to override this function (unless your backend
(8)                     doesn't have a dpi, e.g., postscript or svg).  Some imaging
(8)                     systems assume some value for pixels per inch::
(12)                        points to pixels = points * pixels_per_inch/72 * dpi/72
(8)                     Parameters
(8)                     ----------
(8)                     points : float or array-like
(8)                     Returns
(8)                     -------
(8)                     Points converted to pixels
(8)                     """
(8)                     return points
(4)                 def start_rasterizing(self):
(8)                     """
(8)                     Switch to the raster renderer.
(8)                     Used by `.MixedModeRenderer`.
(8)                     """
(4)                 def stop_rasterizing(self):
(8)                     """
(8)                     Switch back to the vector renderer and draw the contents of the raster
(8)                     renderer as an image on the vector renderer.
(8)                     Used by `.MixedModeRenderer`.
(8)                     """
(4)                 def start_filter(self):
(8)                     """
(8)                     Switch to a temporary renderer for image filtering effects.
(8)                     Currently only supported by the agg renderer.
(8)                     """
(4)                 def stop_filter(self, filter_func):
(8)                     """
(8)                     Switch back to the original renderer.  The contents of the temporary
(8)                     renderer is processed with the *filter_func* and is drawn on the
(8)                     original renderer as an image.
(8)                     Currently only supported by the agg renderer.
(8)                     """
(4)                 def _draw_disabled(self):
(8)                     """
```

```
(8)                     Context manager to temporary disable drawing.
(8)                     This is used for getting the drawn size of Artists.  This lets us
(8)                     run the draw process to update any Python state but does not pay the
(8)                     cost of the draw_XYZ calls on the canvas.
(8)                     """
(8)                     no_ops = {
(12)                        meth_name: lambda *args, **kwargs: None
(12)                        for meth_name in dir(RendererBase)
(12)                        if (meth_name.startswith("draw_")
(16)                            or meth_name in ["open_group", "close_group"])
(8)                     }
(8)                     return _setattr_cm(self, **no_ops)
(0)             class GraphicsContextBase:
(4)                 """An abstract base class that provides color, line styles, etc."""
(4)                 def __init__(self):
(8)                     self._alpha = 1.0
(8)                     self._forced_alpha = False  # if True, _alpha overrides A from RGBA
(8)                     self._antialiased = 1  # use 0, 1 not True, False for extension code
(8)                     self._capstyle = CapStyle('butt')
(8)                     self._cliprect = None
(8)                     self._clippath = None
(8)                     self._dashes = 0, None
(8)                     self._joinstyle = JoinStyle('round')
(8)                     self._linestyle = 'solid'
(8)                     self._linewidth = 1
(8)                     self._rgb = (0.0, 0.0, 0.0, 1.0)
(8)                     self._hatch = None
(8)                     self._hatch_color = colors.to_rgba(rcParams['hatch.color'])
(8)                     self._hatch_linewidth = rcParams['hatch.linewidth']
(8)                     self._url = None
(8)                     self._gid = None
(8)                     self._snap = None
(8)                     self._sketch = None
(4)                 def copy_properties(self, gc):
(8)                     """Copy properties from *gc* to self."""
(8)                     self._alpha = gc._alpha
(8)                     self._forced_alpha = gc._forced_alpha
(8)                     self._antialiased = gc._antialiased
(8)                     self._capstyle = gc._capstyle
(8)                     self._cliprect = gc._cliprect
(8)                     self._clippath = gc._clippath
(8)                     self._dashes = gc._dashes
(8)                     self._joinstyle = gc._joinstyle
(8)                     self._linestyle = gc._linestyle
(8)                     self._linewidth = gc._linewidth
(8)                     self._rgb = gc._rgb
(8)                     self._hatch = gc._hatch
(8)                     self._hatch_color = gc._hatch_color
(8)                     self._hatch_linewidth = gc._hatch_linewidth
(8)                     self._url = gc._url
(8)                     self._gid = gc._gid
(8)                     self._snap = gc._snap
(8)                     self._sketch = gc._sketch
(4)                 def restore(self):
(8)                     """
(8)                     Restore the graphics context from the stack - needed only
(8)                     for backends that save graphics contexts on a stack.
(8)                     """
(4)                 def get_alpha(self):
(8)                     """
(8)                     Return the alpha value used for blending - not supported on all
(8)                     backends.
(8)                     """
(8)                     return self._alpha
(4)                 def get_antialiased(self):
(8)                     """Return whether the object should try to do antialiased
rendering."""
(8)                     return self._antialiased
(4)                 def get_capstyle(self):
```

```
(8)                 """Return the `.CapStyle`."""
(8)                 return self._capstyle.name
(4)             def get_clip_rectangle(self):
(8)                 """
(8)                 Return the clip rectangle as a `~matplotlib.transforms.Bbox` instance.
(8)                 """
(8)                 return self._cliprect
(4)             def get_clip_path(self):
(8)                 """
(8)                 Return the clip path in the form (path, transform), where path
(8)                 is a `~.path.Path` instance, and transform is
(8)                 an affine transform to apply to the path before clipping.
(8)                 """
(8)                 if self._clippath is not None:
(12)                    tpath, tr = self._clippath.get_transformed_path_and_affine()
(12)                    if np.all(np.isfinite(tpath.vertices)):
(16)                        return tpath, tr
(12)                    else:
(16)                        _log.warning("Ill-defined clip_path detected. Returning
None.")
(16)                        return None, None
(8)                 return None, None
(4)             def get_dashes(self):
(8)                 """
(8)                 Return the dash style as an (offset, dash-list) pair.
(8)                 See `.set_dashes` for details.
(8)                 Default value is (None, None).
(8)                 """
(8)                 return self._dashes
(4)             def get_forced_alpha(self):
(8)                 """
(8)                 Return whether the value given by get_alpha() should be used to
(8)                 override any other alpha-channel values.
(8)                 """
(8)                 return self._forced_alpha
(4)             def get_joinstyle(self):
(8)                 """Return the `.JoinStyle`."""
(8)                 return self._joinstyle.name
(4)             def get_linewidth(self):
(8)                 """Return the line width in points."""
(8)                 return self._linewidth
(4)             def get_rgb(self):
(8)                 """Return a tuple of three or four floats from 0-1."""
(8)                 return self._rgb
(4)             def get_url(self):
(8)                 """Return a url if one is set, None otherwise."""
(8)                 return self._url
(4)             def get_gid(self):
(8)                 """Return the object identifier if one is set, None otherwise."""
(8)                 return self._gid
(4)             def get_snap(self):
(8)                 """
(8)                 Return the snap setting, which can be:
(8)                 * True: snap vertices to the nearest pixel center
(8)                 * False: leave vertices as-is
(8)                 * None: (auto) If the path contains only rectilinear line segments,
(10)                  round to the nearest pixel center
(8)                 """
(8)                 return self._snap
(4)             def set_alpha(self, alpha):
(8)                 """
(8)                 Set the alpha value used for blending - not supported on all backends.
(8)                 If ``alpha=None`` (the default), the alpha components of the
(8)                 foreground and fill colors will be used to set their respective
(8)                 transparencies (where applicable); otherwise, ``alpha`` will override
(8)                 them.
(8)                 """
(8)                 if alpha is not None:
(12)                    self._alpha = alpha
```

```
(12)                        self._forced_alpha = True
(8)                     else:
(12)                        self._alpha = 1.0
(12)                        self._forced_alpha = False
(8)                     self.set_foreground(self._rgb, isRGBA=True)
(4)                 def set_antialiased(self, b):
(8)                     """Set whether object should be drawn with antialiased rendering."""
(8)                     self._antialiased = int(bool(b))
(4)                 @_docstring.interpd
(4)                 def set_capstyle(self, cs):
(8)                     """
(8)                     Set how to draw endpoints of lines.
(8)                     Parameters
(8)                     ----------
(8)                     cs : `.CapStyle` or %(CapStyle)s
(8)                     """
(8)                     self._capstyle = CapStyle(cs)
(4)                 def set_clip_rectangle(self, rectangle):
(8)                     """Set the clip rectangle to a `.Bbox` or None."""
(8)                     self._cliprect = rectangle
(4)                 def set_clip_path(self, path):
(8)                     """Set the clip path to a `.TransformedPath` or None."""
(8)                     _api.check_isinstance((transforms.TransformedPath, None), path=path)
(8)                     self._clippath = path
(4)                 def set_dashes(self, dash_offset, dash_list):
(8)                     """
(8)                     Set the dash style for the gc.
(8)                     Parameters
(8)                     ----------
(8)                     dash_offset : float
(12)                        Distance, in points, into the dash pattern at which to
(12)                        start the pattern. It is usually set to 0.
(8)                     dash_list : array-like or None
(12)                        The on-off sequence as points.  None specifies a solid line. All
(12)                        values must otherwise be non-negative (:math:`\\ge 0`).
(8)                     Notes
(8)                     -----
(8)                     See p. 666 of the PostScript
(8)                     `Language Reference
(8)                     <https://www.adobe.com/jp/print/postscript/pdfs/PLRM.pdf>`_
(8)                     for more info.
(8)                     """
(8)                     if dash_list is not None:
(12)                        dl = np.asarray(dash_list)
(12)                        if np.any(dl < 0.0):
(16)                            raise ValueError(
(20)                                "All values in the dash list must be non-negative")
(12)                        if dl.size and not np.any(dl > 0.0):
(16)                            raise ValueError(
(20)                                'At least one value in the dash list must be positive')
(8)                     self._dashes = dash_offset, dash_list
(4)                 def set_foreground(self, fg, isRGBA=False):
(8)                     """
(8)                     Set the foreground color.
(8)                     Parameters
(8)                     ----------
(8)                     fg : :mpltype:`color`
(8)                     isRGBA : bool
(12)                        If *fg* is known to be an ``(r, g, b, a)`` tuple, *isRGBA* can be
(12)                        set to True to improve performance.
(8)                     """
(8)                     if self._forced_alpha and isRGBA:
(12)                        self._rgb = fg[:3] + (self._alpha,)
(8)                     elif self._forced_alpha:
(12)                        self._rgb = colors.to_rgba(fg, self._alpha)
(8)                     elif isRGBA:
(12)                        self._rgb = fg
(8)                     else:
(12)                        self._rgb = colors.to_rgba(fg)
```

```
    @_docstring.interpd
    def set_joinstyle(self, js):
        """
        Set how to draw connections between line segments.
        Parameters
        ----------
        js : `.JoinStyle` or %(JoinStyle)s
        """
        self._joinstyle = JoinStyle(js)
    def set_linewidth(self, w):
        """Set the linewidth in points."""
        self._linewidth = float(w)
    def set_url(self, url):
        """Set the url for links in compatible backends."""
        self._url = url
    def set_gid(self, id):
        """Set the id."""
        self._gid = id
    def set_snap(self, snap):
        """
        Set the snap setting which may be:
        * True: snap vertices to the nearest pixel center
        * False: leave vertices as-is
        * None: (auto) If the path contains only rectilinear line segments,
          round to the nearest pixel center
        """
        self._snap = snap
    def set_hatch(self, hatch):
        """Set the hatch style (for fills)."""
        self._hatch = hatch
    def get_hatch(self):
        """Get the current hatch style."""
        return self._hatch
    def get_hatch_path(self, density=6.0):
        """Return a `.Path` for the current hatch."""
        hatch = self.get_hatch()
        if hatch is None:
            return None
        return Path.hatch(hatch, density)
    def get_hatch_color(self):
        """Get the hatch color."""
        return self._hatch_color
    def set_hatch_color(self, hatch_color):
        """Set the hatch color."""
        self._hatch_color = hatch_color
    def get_hatch_linewidth(self):
        """Get the hatch linewidth."""
        return self._hatch_linewidth
    def get_sketch_params(self):
        """
        Return the sketch parameters for the artist.
        Returns
        -------
        tuple or `None`
            A 3-tuple with the following elements:
            * ``scale``: The amplitude of the wiggle perpendicular to the
              source line.
            * ``length``: The length of the wiggle along the line.
            * ``randomness``: The scale factor by which the length is
              shrunken or expanded.
            May return `None` if no sketch parameters were set.
        """
        return self._sketch
    def set_sketch_params(self, scale=None, length=None, randomness=None):
        """
        Set the sketch parameters.
        Parameters
        ----------
        scale : float, optional
```

```
(12)                        The amplitude of the wiggle perpendicular to the source line, in
(12)                        pixels.  If scale is `None`, or not provided, no sketch filter
will
(12)                        be provided.
(8)                     length : float, default: 128
(12)                        The length of the wiggle along the line, in pixels.
(8)                     randomness : float, default: 16
(12)                        The scale factor by which the length is shrunken or expanded.
(8)                     """
(8)                     self._sketch = (
(12)                        None if scale is None
(12)                        else (scale, length or 128., randomness or 16.))
(0)             class TimerBase:
(4)                 """
(4)                 A base class for providing timer events, useful for things animations.
(4)                 Backends need to implement a few specific methods in order to use their
(4)                 own timing mechanisms so that the timer events are integrated into their
(4)                 event loops.
(4)                 Subclasses must override the following methods:
(4)                 - ``_timer_start``: Backend-specific code for starting the timer.
(4)                 - ``_timer_stop``: Backend-specific code for stopping the timer.
(4)                 Subclasses may additionally override the following methods:
(4)                 - ``_timer_set_single_shot``: Code for setting the timer to single shot
(6)                   operating mode, if supported by the timer object.  If not, the `Timer`
(6)                   class itself will store the flag and the ``_on_timer`` method should be
(6)                   overridden to support such behavior.
(4)                 - ``_timer_set_interval``: Code for setting the interval on the timer, if
(6)                   there is a method for doing so on the timer object.
(4)                 - ``_on_timer``: The internal function that any timer object should call,
(6)                   which will handle the task of running all callbacks that have been set.
(4)                 """
(4)                 def __init__(self, interval=None, callbacks=None):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     interval : int, default: 1000ms
(12)                        The time between timer events in milliseconds.  Will be stored as
(12)                        ``timer.interval``.
(8)                     callbacks : list[tuple[callable, tuple, dict]]
(12)                        List of (func, args, kwargs) tuples that will be called upon timer
(12)                        events.  This list is accessible as ``timer.callbacks`` and can be
(12)                        manipulated directly, or the functions `~.TimerBase.add_callback`
(12)                        and `~.TimerBase.remove_callback` can be used.
(8)                     """
(8)                     self.callbacks = [] if callbacks is None else callbacks.copy()
(8)                     self.interval = 1000 if interval is None else interval
(8)                     self.single_shot = False
(4)                 def __del__(self):
(8)                     """Need to stop timer and possibly disconnect timer."""
(8)                     self._timer_stop()
(4)                 @_api.delete_parameter("3.9", "interval", alternative="timer.interval")
(4)                 def start(self, interval=None):
(8)                     """
(8)                     Start the timer object.
(8)                     Parameters
(8)                     ----------
(8)                     interval : int, optional
(12)                        Timer interval in milliseconds; overrides a previously set
interval
(12)                        if provided.
(8)                     """
(8)                     if interval is not None:
(12)                        self.interval = interval
(8)                     self._timer_start()
(4)                 def stop(self):
(8)                     """Stop the timer."""
(8)                     self._timer_stop()
(4)                 def _timer_start(self):
(8)                     pass
```

```
(4)     def _timer_stop(self):
(8)         pass
(4)     @property
(4)     def interval(self):
(8)         """The time between timer events, in milliseconds."""
(8)         return self._interval
(4)     @interval.setter
(4)     def interval(self, interval):
(8)         interval = max(int(interval), 1)
(8)         self._interval = interval
(8)         self._timer_set_interval()
(4)     @property
(4)     def single_shot(self):
(8)         """Whether this timer should stop after a single run."""
(8)         return self._single
(4)     @single_shot.setter
(4)     def single_shot(self, ss):
(8)         self._single = ss
(8)         self._timer_set_single_shot()
(4)     def add_callback(self, func, *args, **kwargs):
(8)         """
(8)         Register *func* to be called by timer when the event fires. Any
(8)         additional arguments provided will be passed to *func*.
(8)         This function returns *func*, which makes it possible to use it as a
(8)         decorator.
(8)         """
(8)         self.callbacks.append((func, args, kwargs))
(8)         return func
(4)     def remove_callback(self, func, *args, **kwargs):
(8)         """
(8)         Remove *func* from list of callbacks.
(8)         *args* and *kwargs* are optional and used to distinguish between
copies
(8)         of the same function registered to be called with different arguments.
(8)         This behavior is deprecated.  In the future, ``*args, **kwargs`` won't
(8)         be considered anymore; to keep a specific callback removable by
itself,
(8)         pass it to `add_callback` as a `functools.partial` object.
(8)         """
(8)         if args or kwargs:
(12)            _api.warn_deprecated(
(16)                "3.1", message="In a future version, Timer.remove_callback "
(16)                "will not take *args, **kwargs anymore, but remove all "
(16)                "callbacks where the callable matches; to keep a specific "
(16)                "callback removable by itself, pass it to add_callback as a "
(16)                "functools.partial object.")
(12)            self.callbacks.remove((func, args, kwargs))
(8)         else:
(12)            funcs = [c[0] for c in self.callbacks]
(12)            if func in funcs:
(16)                self.callbacks.pop(funcs.index(func))
(4)     def _timer_set_interval(self):
(8)         """Used to set interval on underlying timer object."""
(4)     def _timer_set_single_shot(self):
(8)         """Used to set single shot on underlying timer object."""
(4)     def _on_timer(self):
(8)         """
(8)         Runs all function that have been registered as callbacks. Functions
(8)         can return False (or 0) if they should not be called any more. If
there
(8)         are no callbacks, the timer is automatically stopped.
(8)         """
(8)         for func, args, kwargs in self.callbacks:
(12)            ret = func(*args, **kwargs)
(12)            if ret == 0:
(16)                self.callbacks.remove((func, args, kwargs))
(8)         if len(self.callbacks) == 0:
(12)            self.stop()
(0)     class Event:
```

```
(4)                 """
(4)                 A Matplotlib event.
(4)                 The following attributes are defined and shown with their default values.
(4)                 Subclasses may define additional attributes.
(4)                 Attributes
(4)                 ----------
(4)                 name : str
(8)                     The event name.
(4)                 canvas : `FigureCanvasBase`
(8)                     The backend-specific canvas instance generating the event.
(4)                 guiEvent
(8)                     The GUI event that triggered the Matplotlib event.
(4)                 """
(4)                 def __init__(self, name, canvas, guiEvent=None):
(8)                     self.name = name
(8)                     self.canvas = canvas
(8)                     self._guiEvent = guiEvent
(8)                     self._guiEvent_deleted = False
(4)                 def _process(self):
(8)                     """Process this event on ``self.canvas``, then unset ``guiEvent``."""
(8)                     self.canvas.callbacks.process(self.name, self)
(8)                     self._guiEvent_deleted = True
(4)                 @property
(4)                 def guiEvent(self):
(8)                     if self._guiEvent_deleted:
(12)                        _api.warn_deprecated(
(16)                            "3.8", message="Accessing guiEvent outside of the original GUI
event "
(16)                            "handler is unsafe and deprecated since %(since)s; in the
future, the "
(16)                            "attribute will be set to None after quitting the event
handler.  You "
(16)                            "may separately record the value of the guiEvent attribute at
your own "
(16)                            "risk.")
(8)                     return self._guiEvent
(0)             class DrawEvent(Event):
(4)                 """
(4)                 An event triggered by a draw operation on the canvas.
(4)                 In most backends, callbacks subscribed to this event will be fired after
(4)                 the rendering is complete but before the screen is updated. Any extra
(4)                 artists drawn to the canvas's renderer will be reflected without an
(4)                 explicit call to ``blit``.
(4)                 .. warning::
(7)                    Calling ``canvas.draw`` and ``canvas.blit`` in these callbacks may
(7)                    not be safe with all backends and may cause infinite recursion.
(4)                 A DrawEvent has a number of special attributes in addition to those
defined
(4)                 by the parent `Event` class.
(4)                 Attributes
(4)                 ----------
(4)                 renderer : `RendererBase`
(8)                     The renderer for the draw event.
(4)                 """
(4)                 def __init__(self, name, canvas, renderer):
(8)                     super().__init__(name, canvas)
(8)                     self.renderer = renderer
(0)             class ResizeEvent(Event):
(4)                 """
(4)                 An event triggered by a canvas resize.
(4)                 A ResizeEvent has a number of special attributes in addition to those
(4)                 defined by the parent `Event` class.
(4)                 Attributes
(4)                 ----------
(4)                 width : int
(8)                     Width of the canvas in pixels.
(4)                 height : int
(8)                     Height of the canvas in pixels.
(4)                 """
```

```
(4)                 def __init__(self, name, canvas):
(8)                     super().__init__(name, canvas)
(8)                     self.width, self.height = canvas.get_width_height()
(0)             class CloseEvent(Event):
(4)                 """An event triggered by a figure being closed."""
(0)             class LocationEvent(Event):
(4)                 """
(4)                 An event that has a screen location.
(4)                 A LocationEvent has a number of special attributes in addition to those
(4)                 defined by the parent `Event` class.
(4)                 Attributes
(4)                 ----------
(4)                 x, y : int or None
(8)                     Event location in pixels from bottom left of canvas.
(4)                 inaxes : `~matplotlib.axes.Axes` or None
(8)                     The `~.axes.Axes` instance over which the mouse is, if any.
(4)                 xdata, ydata : float or None
(8)                     Data coordinates of the mouse within *inaxes*, or *None* if the mouse
(8)                     is not over an Axes.
(4)                 modifiers : frozenset
(8)                     The keyboard modifiers currently being pressed (except for KeyEvent).
(4)                 """
(4)                 _lastevent = None
(4)                 lastevent = _api.deprecated("3.8")(
(8)                     _api.classproperty(lambda cls: cls._lastevent))
(4)                 _last_axes_ref = None
(4)                 def __init__(self, name, canvas, x, y, guiEvent=None, *, modifiers=None):
(8)                     super().__init__(name, canvas, guiEvent=guiEvent)
(8)                     self.x = int(x) if x is not None else x
(8)                     self.y = int(y) if y is not None else y
(8)                     self.inaxes = None  # the Axes instance the mouse is over
(8)                     self.xdata = None   # x coord of mouse in data coords
(8)                     self.ydata = None   # y coord of mouse in data coords
(8)                     self.modifiers = frozenset(modifiers if modifiers is not None else [])
(8)                     if x is None or y is None:
(12)                        return
(8)                     self._set_inaxes(self.canvas.inaxes((x, y))
(25)                                     if self.canvas.mouse_grabber is None else
(25)                                     self.canvas.mouse_grabber,
(25)                                     (x, y))
(4)                 def _set_inaxes(self, inaxes, xy=None):
(8)                     self.inaxes = inaxes
(8)                     if inaxes is not None:
(12)                        try:
(16)                            self.xdata, self.ydata =
inaxes.transData.inverted().transform(
(20)                                xy if xy is not None else (self.x, self.y))
(12)                        except ValueError:
(16)                            pass
(0)             class MouseButton(IntEnum):
(4)                 LEFT = 1
(4)                 MIDDLE = 2
(4)                 RIGHT = 3
(4)                 BACK = 8
(4)                 FORWARD = 9
(0)             class MouseEvent(LocationEvent):
(4)                 """
(4)                 A mouse event ('button_press_event', 'button_release_event', \
(0)             'scroll_event', 'motion_notify_event').
(4)                 A MouseEvent has a number of special attributes in addition to those
(4)                 defined by the parent `Event` and `LocationEvent` classes.
(4)                 Attributes
(4)                 ----------
(4)                 button : None or `MouseButton` or {'up', 'down'}
(8)                     The button pressed. 'up' and 'down' are used for scroll events.
(8)                     Note that LEFT and RIGHT actually refer to the "primary" and
(8)                     "secondary" buttons, i.e. if the user inverts their left and right
(8)                     buttons ("left-handed setting") then the LEFT button will be the one
(8)                     physically on the right.
```

```
(8)                 If this is unset, *name* is "scroll_event", and *step* is nonzero, then
(8)                 this will be set to "up" or "down" depending on the sign of *step*.
(4)             key : None or str
(8)                 The key pressed when the mouse event triggered, e.g. 'shift'.
(8)                 See `KeyEvent`.
(8)                 .. warning::
(11)                   This key is currently obtained from the last 'key_press_event' or
(11)                   'key_release_event' that occurred within the canvas.  Thus, if the
(11)                   last change of keyboard state occurred while the canvas did not have
(11)                   focus, this attribute will be wrong.  On the other hand, the
(11)                   ``modifiers`` attribute should always be correct, but it can only
(11)                   report on modifier keys.
(4)             step : float
(8)                 The number of scroll steps (positive for 'up', negative for 'down').
(8)                 This applies only to 'scroll_event' and defaults to 0 otherwise.
(4)             dblclick : bool
(8)                 Whether the event is a double-click. This applies only to
(8)                 'button_press_event' and is False otherwise. In particular, it's
(8)                 not used in 'button_release_event'.
(4)             Examples
(4)             --------
(4)             ::
(8)                 def on_press(event):
(12)                    print('you pressed', event.button, event.xdata, event.ydata)
(8)                 cid = fig.canvas.mpl_connect('button_press_event', on_press)
(4)             """
(4)             def __init__(self, name, canvas, x, y, button=None, key=None,
(17)                         step=0, dblclick=False, guiEvent=None, *, modifiers=None):
(8)                 super().__init__(
(12)                    name, canvas, x, y, guiEvent=guiEvent, modifiers=modifiers)
(8)                 if button in MouseButton.__members__.values():
(12)                    button = MouseButton(button)
(8)                 if name == "scroll_event" and button is None:
(12)                    if step > 0:
(16)                        button = "up"
(12)                    elif step < 0:
(16)                        button = "down"
(8)                 self.button = button
(8)                 self.key = key
(8)                 self.step = step
(8)                 self.dblclick = dblclick
(4)             def __str__(self):
(8)                 return (f"{self.name}: "
(16)                        f"xy=({self.x}, {self.y}) xydata=({self.xdata}, {self.ydata})"
(16)                        f"button={self.button} dblclick={self.dblclick} "
(16)                        f"inaxes={self.inaxes}")
(0)         class PickEvent(Event):
(4)             """
(4)             A pick event.
(4)             This event is fired when the user picks a location on the canvas
(4)             sufficiently close to an artist that has been made pickable with
(4)             `.Artist.set_picker`.
(4)             A PickEvent has a number of special attributes in addition to those defined
(4)             by the parent `Event` class.
(4)             Attributes
(4)             ----------
(4)             mouseevent : `MouseEvent`
(8)                 The mouse event that generated the pick.
(4)             artist : `~matplotlib.artist.Artist`
(8)                 The picked artist.  Note that artists are not pickable by default
(8)                 (see `.Artist.set_picker`).
(4)             other
(8)                 Additional attributes may be present depending on the type of the
(8)                 picked object; e.g., a `.Line2D` pick may define different extra
(8)                 attributes than a `.PatchCollection` pick.
```

```
(4)             Examples
(4)             --------
(4)             Bind a function ``on_pick()`` to pick events, that prints the coordinates
(4)             of the picked data point::
(8)                 ax.plot(np.rand(100), 'o', picker=5)  # 5 points tolerance
(8)                 def on_pick(event):
(12)                    line = event.artist
(12)                    xdata, ydata = line.get_data()
(12)                    ind = event.ind
(12)                    print(f'on pick line: {xdata[ind]:.3f}, {ydata[ind]:.3f}')
(8)                 cid = fig.canvas.mpl_connect('pick_event', on_pick)
(4)             """
(4)             def __init__(self, name, canvas, mouseevent, artist,
(17)                         guiEvent=None, **kwargs):
(8)                 if guiEvent is None:
(12)                    guiEvent = mouseevent.guiEvent
(8)                 super().__init__(name, canvas, guiEvent)
(8)                 self.mouseevent = mouseevent
(8)                 self.artist = artist
(8)                 self.__dict__.update(kwargs)
(0)         class KeyEvent(LocationEvent):
(4)             """
(4)             A key event (key press, key release).
(4)             A KeyEvent has a number of special attributes in addition to those defined
(4)             by the parent `Event` and `LocationEvent` classes.
(4)             Attributes
(4)             ----------
(4)             key : None or str
(8)                 The key(s) pressed. Could be *None*, a single case sensitive Unicode
(8)                 character ("g", "G", "#", etc.), a special key ("control", "shift",
(8)                 "f1", "up", etc.) or a combination of the above (e.g., "ctrl+alt+g",
(8)                 "ctrl+alt+G").
(4)             Notes
(4)             -----
(4)             Modifier keys will be prefixed to the pressed key and will be in the order
(4)             "ctrl", "alt", "super". The exception to this rule is when the pressed key
(4)             is itself a modifier key, therefore "ctrl+alt" and "alt+control" can both
(4)             be valid key values.
(4)             Examples
(4)             --------
(4)             ::
(8)                 def on_key(event):
(12)                    print('you pressed', event.key, event.xdata, event.ydata)
(8)                 cid = fig.canvas.mpl_connect('key_press_event', on_key)
(4)             """
(4)             def __init__(self, name, canvas, key, x=0, y=0, guiEvent=None):
(8)                 super().__init__(name, canvas, x, y, guiEvent=guiEvent)
(8)                 self.key = key
(0)         def _key_handler(event):
(4)             if event.name == "key_press_event":
(8)                 event.canvas._key = event.key
(4)             elif event.name == "key_release_event":
(8)                 event.canvas._key = None
(0)         def _mouse_handler(event):
(4)             if event.name == "button_press_event":
(8)                 event.canvas._button = event.button
(4)             elif event.name == "button_release_event":
(8)                 event.canvas._button = None
(4)             elif event.name == "motion_notify_event" and event.button is None:
(8)                 event.button = event.canvas._button
(4)             if event.key is None:
(8)                 event.key = event.canvas._key
(4)             if event.name == "motion_notify_event":
(8)                 last_ref = LocationEvent._last_axes_ref
(8)                 last_axes = last_ref() if last_ref else None
(8)                 if last_axes != event.inaxes:
(12)                    if last_axes is not None:
(16)                        try:
(20)                            leave_event = LocationEvent(
```

```
1250: (24)                                "axes_leave_event", last_axes.figure.canvas,
1251: (24)                                event.x, event.y, event.guiEvent,
1252: (24)                                modifiers=event.modifiers)
1253: (20)                            leave_event._set_inaxes(last_axes)
1254: (20)                            last_axes.figure.canvas.callbacks.process(
1255: (24)                                "axes_leave_event", leave_event)
1256: (16)                        except Exception:
1257: (20)                            pass  # The last canvas may already have been torn down.
1258: (12)                    if event.inaxes is not None:
1259: (16)                        event.canvas.callbacks.process("axes_enter_event", event)
1260: (8)                LocationEvent._last_axes_ref = (
1261: (12)                    weakref.ref(event.inaxes) if event.inaxes else None)
1262: (8)                LocationEvent._lastevent = (
1263: (12)                    None if event.name == "figure_leave_event" else event)
1264: (0)            def _get_renderer(figure, print_method=None):
1265: (4)                """
1266: (4)                Get the renderer that would be used to save a `.Figure`.
1267: (4)                If you need a renderer without any active draw methods use
1268: (4)                renderer._draw_disabled to temporary patch them out at your call site.
1269: (4)                """
1270: (4)                class Done(Exception):
1271: (8)                    pass
1272: (4)                def _draw(renderer): raise Done(renderer)
1273: (4)                with cbook._setattr_cm(figure, draw=_draw), ExitStack() as stack:
1274: (8)                    if print_method is None:
1275: (12)                       fmt = figure.canvas.get_default_filetype()
1276: (12)                       print_method = stack.enter_context(
1277: (16)                           figure.canvas._switch_canvas_and_return_print_method(fmt))
1278: (8)                    try:
1279: (12)                       print_method(io.BytesIO())
1280: (8)                    except Done as exc:
1281: (12)                       renderer, = exc.args
1282: (12)                       return renderer
1283: (8)                    else:
1284: (12)                       raise RuntimeError(f"{print_method} did not call Figure.draw, so "
1285: (31)                                          f"no renderer is available")
1286: (0)            def _no_output_draw(figure):
1287: (4)                figure.draw_without_rendering()
1288: (0)            def _is_non_interactive_terminal_ipython(ip):
1289: (4)                """
1290: (4)                Return whether we are in a terminal IPython, but non interactive.
1291: (4)                When in _terminal_ IPython, ip.parent will have and `interact` attribute,
1292: (4)                if this attribute is False we do not setup eventloop integration as the
1293: (4)                user will _not_ interact with IPython. In all other case (ZMQKernel, or is
1294: (4)                interactive), we do.
1295: (4)                """
1296: (4)                return (hasattr(ip, 'parent')
1297: (12)                       and (ip.parent is not None)
1298: (12)                       and getattr(ip.parent, 'interact', None) is False)
1299: (0)            @contextmanager
1300: (0)            def _allow_interrupt(prepare_notifier, handle_sigint):
1301: (4)                """
1302: (4)                A context manager that allows terminating a plot by sending a SIGINT.  It
1303: (4)                is necessary because the running backend prevents the Python interpreter
1304: (4)                from running and processing signals (i.e., to raise a KeyboardInterrupt).
1305: (4)                To solve this, one needs to somehow wake up the interpreter and make it
1306: (4)                close the plot window.  We do this by using the signal.set_wakeup_fd()
1307: (4)                function which organizes a write of the signal number into a socketpair.
1308: (4)                A backend-specific function, *prepare_notifier*, arranges to listen to
1309: (4)                the pair's read socket while the event loop is running.  (If it returns a
1310: (4)                notifier object, that object is kept alive while the context manager
runs.)
1311: (4)                If SIGINT was indeed caught, after exiting the on_signal() function the
1312: (4)                interpreter reacts to the signal according to the handler function which
1313: (4)                had been set up by a signal.signal() call; here, we arrange to call the
1314: (4)                backend-specific *handle_sigint* function.  Finally, we call the old
SIGINT
1315: (4)                handler with the same arguments that were given to our custom handler.
1316: (4)                We do this only if the old handler for SIGINT was not None, which means
```

```
(4)                 that a non-python handler was installed, i.e. in Julia, and not SIG_IGN
(4)                 which means we should ignore the interrupts.
(4)                 Parameters
(4)                 ----------
(4)                 prepare_notifier : Callable[[socket.socket], object]
(4)                 handle_sigint : Callable[[], object]
(4)                 """
(4)                 old_sigint_handler = signal.getsignal(signal.SIGINT)
(4)                 if old_sigint_handler in (None, signal.SIG_IGN, signal.SIG_DFL):
(8)                     yield
(8)                     return
(4)                 handler_args = None
(4)                 wsock, rsock = socket.socketpair()
(4)                 wsock.setblocking(False)
(4)                 rsock.setblocking(False)
(4)                 old_wakeup_fd = signal.set_wakeup_fd(wsock.fileno())
(4)                 notifier = prepare_notifier(rsock)
(4)                 def save_args_and_handle_sigint(*args):
(8)                     nonlocal handler_args
(8)                     handler_args = args
(8)                     handle_sigint()
(4)                 signal.signal(signal.SIGINT, save_args_and_handle_sigint)
(4)                 try:
(8)                     yield
(4)                 finally:
(8)                     wsock.close()
(8)                     rsock.close()
(8)                     signal.set_wakeup_fd(old_wakeup_fd)
(8)                     signal.signal(signal.SIGINT, old_sigint_handler)
(8)                     if handler_args is not None:
(12)                        old_sigint_handler(*handler_args)
(0)             class FigureCanvasBase:
(4)                 """
(4)                 The canvas the figure renders into.
(4)                 Attributes
(4)                 ----------
(4)                 figure : `~matplotlib.figure.Figure`
(8)                     A high-level figure instance.
(4)                 """
(4)                 required_interactive_framework = None
(4)                 manager_class = _api.classproperty(lambda cls: FigureManagerBase)
(4)                 events = [
(8)                     'resize_event',
(8)                     'draw_event',
(8)                     'key_press_event',
(8)                     'key_release_event',
(8)                     'button_press_event',
(8)                     'button_release_event',
(8)                     'scroll_event',
(8)                     'motion_notify_event',
(8)                     'pick_event',
(8)                     'figure_enter_event',
(8)                     'figure_leave_event',
(8)                     'axes_enter_event',
(8)                     'axes_leave_event',
(8)                     'close_event'
(4)                 ]
(4)                 fixed_dpi = None
(4)                 filetypes = _default_filetypes
(4)                 @_api.classproperty
(4)                 def supports_blit(cls):
(8)                     """If this Canvas sub-class supports blitting."""
(8)                     return (hasattr(cls, "copy_from_bbox")
(16)                            and hasattr(cls, "restore_region"))
(4)                 def __init__(self, figure=None):
(8)                     from matplotlib.figure import Figure
(8)                     self._fix_ipython_backend2gui()
(8)                     self._is_idle_drawing = True
(8)                     self._is_saving = False
```

```
(8)                 if figure is None:
(12)                    figure = Figure()
(8)                 figure.set_canvas(self)
(8)                 self.figure = figure
(8)                 self.manager = None
(8)                 self.widgetlock = widgets.LockDraw()
(8)                 self._button = None  # the button pressed
(8)                 self._key = None  # the key pressed
(8)                 self.mouse_grabber = None  # the Axes currently grabbing mouse
(8)                 self.toolbar = None  # NavigationToolbar2 will set me
(8)                 self._is_idle_drawing = False
(8)                 figure._original_dpi = figure.dpi
(8)                 self._device_pixel_ratio = 1
(8)                 super().__init__()  # Typically the GUI widget init (if any).
(4)             callbacks = property(lambda self: self.figure._canvas_callbacks)
(4)             button_pick_id = property(lambda self: self.figure._button_pick_id)
(4)             scroll_pick_id = property(lambda self: self.figure._scroll_pick_id)
(4)             @classmethod
(4)             @functools.cache
(4)             def _fix_ipython_backend2gui(cls):
(8)                 mod_ipython = sys.modules.get("IPython")
(8)                 if mod_ipython is None or mod_ipython.version_info[:2] >= (8, 24):
(12)                    return
(8)                 import IPython
(8)                 ip = IPython.get_ipython()
(8)                 if not ip:
(12)                    return
(8)                 from IPython.core import pylabtools as pt
(8)                 if (not hasattr(pt, "backend2gui")
(16)                        or not hasattr(ip, "enable_matplotlib")):
(12)                    return
(8)                 backend2gui_rif = {
(12)                    "qt": "qt",
(12)                    "gtk3": "gtk3",
(12)                    "gtk4": "gtk4",
(12)                    "wx": "wx",
(12)                    "macosx": "osx",
(8)                 }.get(cls.required_interactive_framework)
(8)                 if backend2gui_rif:
(12)                    if _is_non_interactive_terminal_ipython(ip):
(16)                        ip.enable_gui(backend2gui_rif)
(4)             @classmethod
(4)             def new_manager(cls, figure, num):
(8)                 """
(8)                 Create a new figure manager for *figure*, using this canvas class.
(8)                 Notes
(8)                 -----
(8)                 This method should not be reimplemented in subclasses.  If
(8)                 custom manager creation logic is needed, please reimplement
(8)                 ``FigureManager.create_with_canvas``.
(8)                 """
(8)                 return cls.manager_class.create_with_canvas(cls, figure, num)
(4)             @contextmanager
(4)             def _idle_draw_cntx(self):
(8)                 self._is_idle_drawing = True
(8)                 try:
(12)                    yield
(8)                 finally:
(12)                    self._is_idle_drawing = False
(4)             def is_saving(self):
(8)                 """
(8)                 Return whether the renderer is in the process of saving
(8)                 to a file, rather than rendering for an on-screen buffer.
(8)                 """
(8)                 return self._is_saving
(4)             def blit(self, bbox=None):
(8)                 """Blit the canvas in bbox (default entire canvas)."""
(4)             def inaxes(self, xy):
(8)                 """
```

```
(8)                 Return the topmost visible `~.axes.Axes` containing the point *xy*.
(8)                 Parameters
(8)                 ----------
(8)                 xy : (float, float)
(12)                    (x, y) pixel positions from left/bottom of the canvas.
(8)                 Returns
(8)                 -------
(8)                 `~matplotlib.axes.Axes` or None
(12)                    The topmost visible Axes containing the point, or None if there
(12)                    is no Axes at the point.
(8)                 """
(8)                 axes_list = [a for a in self.figure.get_axes()
(21)                             if a.patch.contains_point(xy) and a.get_visible()]
(8)                 if axes_list:
(12)                    axes = cbook._topmost_artist(axes_list)
(8)                 else:
(12)                    axes = None
(8)                 return axes
(4)             def grab_mouse(self, ax):
(8)                 """
(8)                 Set the child `~.axes.Axes` which is grabbing the mouse events.
(8)                 Usually called by the widgets themselves. It is an error to call this
(8)                 if the mouse is already grabbed by another Axes.
(8)                 """
(8)                 if self.mouse_grabber not in (None, ax):
(12)                    raise RuntimeError("Another Axes already grabs mouse input")
(8)                 self.mouse_grabber = ax
(4)             def release_mouse(self, ax):
(8)                 """
(8)                 Release the mouse grab held by the `~.axes.Axes` *ax*.
(8)                 Usually called by the widgets. It is ok to call this even if *ax*
(8)                 doesn't have the mouse grab currently.
(8)                 """
(8)                 if self.mouse_grabber is ax:
(12)                    self.mouse_grabber = None
(4)             def set_cursor(self, cursor):
(8)                 """
(8)                 Set the current cursor.
(8)                 This may have no effect if the backend does not display anything.
(8)                 If required by the backend, this method should trigger an update in
(8)                 the backend event loop after the cursor is set, as this method may be
(8)                 called e.g. before a long-running task during which the GUI is not
(8)                 updated.
(8)                 Parameters
(8)                 ----------
(8)                 cursor : `.Cursors`
(12)                    The cursor to display over the canvas. Note: some backends may
(12)                    change the cursor for the entire window.
(8)                 """
(4)             def draw(self, *args, **kwargs):
(8)                 """
(8)                 Render the `.Figure`.
(8)                 This method must walk the artist tree, even if no output is produced,
(8)                 because it triggers deferred work that users may want to access
(8)                 before saving output to disk. For example computing limits,
(8)                 auto-limits, and tick values.
(8)                 """
(4)             def draw_idle(self, *args, **kwargs):
(8)                 """
(8)                 Request a widget redraw once control returns to the GUI event loop.
(8)                 Even if multiple calls to `draw_idle` occur before control returns
(8)                 to the GUI event loop, the figure will only be rendered once.
(8)                 Notes
(8)                 -----
(8)                 Backends may choose to override the method and implement their own
(8)                 strategy to prevent multiple renderings.
(8)                 """
(8)                 if not self._is_idle_drawing:
(12)                    with self._idle_draw_cntx():
```

```
(16)                        self.draw(*args, **kwargs)
(4)                 @property
(4)                 def device_pixel_ratio(self):
(8)                     """
(8)                     The ratio of physical to logical pixels used for the canvas on screen.
(8)                     By default, this is 1, meaning physical and logical pixels are the
same
(8)                     size. Subclasses that support High DPI screens may set this property
to
(8)                     indicate that said ratio is different. All Matplotlib interaction,
(8)                     unless working directly with the canvas, remains in logical pixels.
(8)                     """
(8)                     return self._device_pixel_ratio
(4)                 def _set_device_pixel_ratio(self, ratio):
(8)                     """
(8)                     Set the ratio of physical to logical pixels used for the canvas.
(8)                     Subclasses that support High DPI screens can set this property to
(8)                     indicate that said ratio is different. The canvas itself will be
(8)                     created at the physical size, while the client side will use the
(8)                     logical size. Thus the DPI of the Figure will change to be scaled by
(8)                     this ratio. Implementations that support High DPI screens should use
(8)                     physical pixels for events so that transforms back to Axes space are
(8)                     correct.
(8)                     By default, this is 1, meaning physical and logical pixels are the
same
(8)                     size.
(8)                     Parameters
(8)                     ----------
(8)                     ratio : float
(12)                        The ratio of logical to physical pixels used for the canvas.
(8)                     Returns
(8)                     -------
(8)                     bool
(12)                        Whether the ratio has changed. Backends may interpret this as a
(12)                        signal to resize the window, repaint the canvas, or change any
(12)                        other relevant properties.
(8)                     """
(8)                     if self._device_pixel_ratio == ratio:
(12)                        return False
(8)                     dpi = ratio * self.figure._original_dpi
(8)                     self.figure._set_dpi(dpi, forward=False)
(8)                     self._device_pixel_ratio = ratio
(8)                     return True
(4)                 def get_width_height(self, *, physical=False):
(8)                     """
(8)                     Return the figure width and height in integral points or pixels.
(8)                     When the figure is used on High DPI screens (and the backend supports
(8)                     it), the truncation to integers occurs after scaling by the device
(8)                     pixel ratio.
(8)                     Parameters
(8)                     ----------
(8)                     physical : bool, default: False
(12)                        Whether to return true physical pixels or logical pixels. Physical
(12)                        pixels may be used by backends that support HiDPI, but still
(12)                        configure the canvas using its actual size.
(8)                     Returns
(8)                     -------
(8)                     width, height : int
(12)                        The size of the figure, in points or pixels, depending on the
(12)                        backend.
(8)                     """
(8)                     return tuple(int(size / (1 if physical else self.device_pixel_ratio))
(21)                                 for size in self.figure.bbox.max)
(4)                 @classmethod
(4)                 def get_supported_filetypes(cls):
(8)                     """Return dict of savefig file formats supported by this backend."""
(8)                     return cls.filetypes
(4)                 @classmethod
(4)                 def get_supported_filetypes_grouped(cls):
```

```
1590: (8)                    """
1591: (8)                    Return a dict of savefig file formats supported by this backend,
1592: (8)                    where the keys are a file type name, such as 'Joint Photographic
1593: (8)                    Experts Group', and the values are a list of filename extensions used
1594: (8)                    for that filetype, such as ['jpg', 'jpeg'].
1595: (8)                    """
1596: (8)                    groupings = {}
1597: (8)                    for ext, name in cls.filetypes.items():
1598: (12)                       groupings.setdefault(name, []).append(ext)
1599: (12)                       groupings[name].sort()
1600: (8)                    return groupings
1601: (4)                @contextmanager
1602: (4)                def _switch_canvas_and_return_print_method(self, fmt, backend=None):
1603: (8)                    """
1604: (8)                    Context manager temporarily setting the canvas for saving the figure::
1605: (12)                       with canvas._switch_canvas_and_return_print_method(fmt, backend)
\\
1606: (20)                               as print_method:
1607: (8)                    Parameters
1608: (8)                    ----------
1609: (8)                    fmt : str
1610: (12)                       If *backend* is None, then determine a suitable canvas class for
1611: (12)                       saving to format *fmt* -- either the current canvas class, if it
1612: (12)                       supports *fmt*, or whatever `get_registered_canvas_class` returns;
1613: (12)                       switch the figure canvas to that canvas class.
1614: (8)                    backend : str or None, default: None
1615: (12)                       If not None, switch the figure canvas to the ``FigureCanvas``
class
1616: (12)                       of the given backend.
1617: (8)                    """
1618: (8)                    canvas = None
1619: (8)                    if backend is not None:
1620: (12)                       from .backends.registry import backend_registry
1621: (12)                       canvas_class =
backend_registry.load_backend_module(backend).FigureCanvas
1622: (12)                       if not hasattr(canvas_class, f"print_{fmt}"):
1623: (16)                           raise ValueError(
1624: (20)                               f"The {backend!r} backend does not support {fmt} output")
1625: (12)                       canvas = canvas_class(self.figure)
1626: (8)                    elif hasattr(self, f"print_{fmt}"):
1627: (12)                       canvas = self
1628: (8)                    else:
1629: (12)                       canvas_class = get_registered_canvas_class(fmt)
1630: (12)                       if canvas_class is None:
1631: (16)                           raise ValueError(
1632: (20)                               "Format {!r} is not supported (supported formats:
{})".format(
1633: (24)                                   fmt, ",
".join(sorted(self.get_supported_filetypes()))))
1634: (12)                       canvas = canvas_class(self.figure)
1635: (8)                    canvas._is_saving = self._is_saving
1636: (8)                    meth = getattr(canvas, f"print_{fmt}")
1637: (8)                    mod = (meth.func.__module__
1638: (15)                          if hasattr(meth, "func")  # partialmethod, e.g. backend_wx.
1639: (15)                          else meth.__module__)
1640: (8)                    if mod.startswith(("matplotlib.", "mpl_toolkits.")):
1641: (12)                       optional_kws = {  # Passed by print_figure for other renderers.
1642: (16)                           "dpi", "facecolor", "edgecolor", "orientation",
1643: (16)                           "bbox_inches_restore"}
1644: (12)                       skip = optional_kws - {*inspect.signature(meth).parameters}
1645: (12)                       print_method = functools.wraps(meth)(lambda *args, **kwargs: meth(
1646: (16)                           *args, **{k: v for k, v in kwargs.items() if k not in skip}))
1647: (8)                    else:  # Let third-parties do as they see fit.
1648: (12)                       print_method = meth
1649: (8)                    try:
1650: (12)                       yield print_method
1651: (8)                    finally:
1652: (12)                       self.figure.canvas = self
1653: (4)                def print_figure(
```

```
1654: (12)                      self, filename, dpi=None, facecolor=None, edgecolor=None,
1655: (12)                      orientation='portrait', format=None, *,
1656: (12)                      bbox_inches=None, pad_inches=None, bbox_extra_artists=None,
1657: (12)                      backend=None, **kwargs):
1658: (8)                   """
1659: (8)                   Render the figure to hardcopy. Set the figure patch face and edge
1660: (8)                   colors.  This is useful because some of the GUIs have a gray figure
1661: (8)                   face color background and you'll probably want to override this on
1662: (8)                   hardcopy.
1663: (8)                   Parameters
1664: (8)                   ----------
1665: (8)                   filename : str or path-like or file-like
1666: (12)                      The file where the figure is saved.
1667: (8)                   dpi : float, default: :rc:`savefig.dpi`
1668: (12)                      The dots per inch to save the figure in.
1669: (8)                   facecolor : :mpltype:`color` or 'auto', default:
:rc:`savefig.facecolor`
1670: (12)                      The facecolor of the figure.  If 'auto', use the current figure
1671: (12)                      facecolor.
1672: (8)                   edgecolor : :mpltype:`color` or 'auto', default:
:rc:`savefig.edgecolor`
1673: (12)                      The edgecolor of the figure.  If 'auto', use the current figure
1674: (12)                      edgecolor.
1675: (8)                   orientation : {'landscape', 'portrait'}, default: 'portrait'
1676: (12)                      Only currently applies to PostScript printing.
1677: (8)                   format : str, optional
1678: (12)                      Force a specific file format. If not given, the format is inferred
1679: (12)                      from the *filename* extension, and if that fails from
1680: (12)                      :rc:`savefig.format`.
1681: (8)                   bbox_inches : 'tight' or `.Bbox`, default: :rc:`savefig.bbox`
1682: (12)                      Bounding box in inches: only the given portion of the figure is
1683: (12)                      saved.  If 'tight', try to figure out the tight bbox of the
figure.
1684: (8)                   pad_inches : float or 'layout', default: :rc:`savefig.pad_inches`
1685: (12)                      Amount of padding in inches around the figure when bbox_inches is
1686: (12)                      'tight'. If 'layout' use the padding from the constrained or
1687: (12)                      compressed layout engine; ignored if one of those engines is not
in
1688: (12)                      use.
1689: (8)                   bbox_extra_artists : list of `~matplotlib.artist.Artist`, optional
1690: (12)                      A list of extra artists that will be considered when the
1691: (12)                      tight bbox is calculated.
1692: (8)                   backend : str, optional
1693: (12)                      Use a non-default backend to render the file, e.g. to render a
1694: (12)                      png file with the "cairo" backend rather than the default "agg",
1695: (12)                      or a pdf file with the "pgf" backend rather than the default
1696: (12)                      "pdf".  Note that the default backend is normally sufficient.  See
1697: (12)                      :ref:`the-builtin-backends` for a list of valid backends for each
1698: (12)                      file format.  Custom backends can be referenced as "module://...".
1699: (8)                   """
1700: (8)                   if format is None:
1701: (12)                      if isinstance(filename, os.PathLike):
1702: (16)                          filename = os.fspath(filename)
1703: (12)                      if isinstance(filename, str):
1704: (16)                          format = os.path.splitext(filename)[1][1:]
1705: (12)                      if format is None or format == '':
1706: (16)                          format = self.get_default_filetype()
1707: (16)                          if isinstance(filename, str):
1708: (20)                              filename = filename.rstrip('.') + '.' + format
1709: (8)                   format = format.lower()
1710: (8)                   if dpi is None:
1711: (12)                      dpi = rcParams['savefig.dpi']
1712: (8)                   if dpi == 'figure':
1713: (12)                      dpi = getattr(self.figure, '_original_dpi', self.figure.dpi)
1714: (8)                   if kwargs.get("papertype") == 'auto':
1715: (12)                      _api.warn_deprecated(
1716: (16)                          "3.8", name="papertype='auto'", addendum="Pass an explicit
paper type, "
1717: (16)                          "'figure', or omit the *papertype* argument entirely.")
```

```
(8)                     with cbook._setattr_cm(self, manager=None), \
(13)                         self._switch_canvas_and_return_print_method(format, backend) \
(17)                             as print_method, \
(13)                         cbook._setattr_cm(self.figure, dpi=dpi), \
(13)                         cbook._setattr_cm(self.figure.canvas, _device_pixel_ratio=1), \
(13)                         cbook._setattr_cm(self.figure.canvas, _is_saving=True), \
(13)                         ExitStack() as stack:
(12)                         for prop in ["facecolor", "edgecolor"]:
(16)                             color = locals()[prop]
(16)                             if color is None:
(20)                                 color = rcParams[f"savefig.{prop}"]
(16)                             if not cbook._str_equal(color, "auto"):
(20)                                 stack.enter_context(self.figure._cm_set(**{prop: color}))
(12)                         if bbox_inches is None:
(16)                             bbox_inches = rcParams['savefig.bbox']
(12)                         layout_engine = self.figure.get_layout_engine()
(12)                         if layout_engine is not None or bbox_inches == "tight":
(16)                             renderer = _get_renderer(
(20)                                 self.figure,
(20)                                 functools.partial(
(24)                                     print_method, orientation=orientation)
(16)                             )
(16)                             with getattr(renderer, "_draw_disabled", nullcontext)():
(20)                                 self.figure.draw(renderer)
(12)                         if bbox_inches:
(16)                             if bbox_inches == "tight":
(20)                                 bbox_inches = self.figure.get_tightbbox(
(24)                                     renderer, bbox_extra_artists=bbox_extra_artists)
(20)                                 if (isinstance(layout_engine, ConstrainedLayoutEngine) and
(28)                                         pad_inches == "layout"):
(24)                                     h_pad = layout_engine.get()["h_pad"]
(24)                                     w_pad = layout_engine.get()["w_pad"]
(20)                                 else:
(24)                                     if pad_inches in [None, "layout"]:
(28)                                         pad_inches = rcParams['savefig.pad_inches']
(24)                                     h_pad = w_pad = pad_inches
(20)                                 bbox_inches = bbox_inches.padded(w_pad, h_pad)
(16)                             restore_bbox = _tight_bbox.adjust_bbox(
(20)                                 self.figure, bbox_inches, self.figure.canvas.fixed_dpi)
(16)                             _bbox_inches_restore = (bbox_inches, restore_bbox)
(12)                         else:
(16)                             _bbox_inches_restore = None
(12)                         stack.enter_context(self.figure._cm_set(layout_engine='none'))
(12)                         try:
(16)                             with cbook._setattr_cm(self.figure, dpi=dpi):
(20)                                 result = print_method(
(24)                                     filename,
(24)                                     facecolor=facecolor,
(24)                                     edgecolor=edgecolor,
(24)                                     orientation=orientation,
(24)                                     bbox_inches_restore=_bbox_inches_restore,
(24)                                     **kwargs)
(12)                         finally:
(16)                             if bbox_inches and restore_bbox:
(20)                                 restore_bbox()
(12)                         return result
(4)                 @classmethod
(4)                 def get_default_filetype(cls):
(8)                     """
(8)                     Return the default savefig file format as specified in
(8)                     :rc:`savefig.format`.
(8)                     The returned string does not include a period. This method is
(8)                     overridden in backends that only support a single file type.
(8)                     """
(8)                     return rcParams['savefig.format']
(4)                 def get_default_filename(self):
(8)                     """
(8)                     Return a string, which includes extension, suitable for use as
(8)                     a default filename.
```

```
        """
        basename = (self.manager.get_window_title() if self.manager is not None
                    else '')
        basename = (basename or 'image').replace(' ', '_')
        filetype = self.get_default_filetype()
        filename = basename + '.' + filetype
        return filename
    @_api.deprecated("3.8")
    def switch_backends(self, FigureCanvasClass):
        """
        Instantiate an instance of FigureCanvasClass
        This is used for backend switching, e.g., to instantiate a
        FigureCanvasPS from a FigureCanvasGTK.  Note, deep copying is
        not done, so any changes to one of the instances (e.g., setting
        figure size or line props), will be reflected in the other
        """
        newCanvas = FigureCanvasClass(self.figure)
        newCanvas._is_saving = self._is_saving
        return newCanvas
    def mpl_connect(self, s, func):
        """
        Bind function *func* to event *s*.
        Parameters
        ----------
        s : str
            One of the following events ids:
            - 'button_press_event'
            - 'button_release_event'
            - 'draw_event'
            - 'key_press_event'
            - 'key_release_event'
            - 'motion_notify_event'
            - 'pick_event'
            - 'resize_event'
            - 'scroll_event'
            - 'figure_enter_event',
            - 'figure_leave_event',
            - 'axes_enter_event',
            - 'axes_leave_event'
            - 'close_event'.
        func : callable
            The callback function to be executed, which must have the
            signature::
                def func(event: Event) -> Any
            For the location events (button and key press/release), if the
            mouse is over the Axes, the ``inaxes`` attribute of the event will
            be set to the `~matplotlib.axes.Axes` the event occurs is over, and

            additionally, the variables ``xdata`` and ``ydata`` attributes will

            be set to the mouse location in data coordinates.  See `.KeyEvent`
            and `.MouseEvent` for more info.
            .. note::
                If func is a method, this only stores a weak reference to the
                method. Thus, the figure does not influence the lifetime of
                the associated object. Usually, you want to make sure that the
                object is kept alive throughout the lifetime of the figure by
                holding a reference to it.
        Returns
        -------
        cid
            A connection id that can be used with
            `.FigureCanvasBase.mpl_disconnect`.
        Examples
        --------
        ::
            def on_press(event):
                print('you pressed', event.button, event.xdata, event.ydata)
```

```
(12)                cid = canvas.mpl_connect('button_press_event', on_press)
(8)             """
(8)             return self.callbacks.connect(s, func)
(4)         def mpl_disconnect(self, cid):
(8)             """
(8)             Disconnect the callback with id *cid*.
(8)             Examples
(8)             --------
(8)             ::
(12)                cid = canvas.mpl_connect('button_press_event', on_press)
(12)                canvas.mpl_disconnect(cid)
(8)             """
(8)             self.callbacks.disconnect(cid)
(4)         _timer_cls = TimerBase
(4)         def new_timer(self, interval=None, callbacks=None):
(8)             """
(8)             Create a new backend-specific subclass of `.Timer`.
(8)             This is useful for getting periodic events through the backend's
native
(8)             event loop.  Implemented only for backends with GUIs.
(8)             Parameters
(8)             ----------
(8)             interval : int
(12)                Timer interval in milliseconds.
(8)             callbacks : list[tuple[callable, tuple, dict]]
(12)                Sequence of (func, args, kwargs) where ``func(*args, **kwargs)``
(12)                will be executed by the timer every *interval*.
(12)                Callbacks which return ``False`` or ``0`` will be removed from the
(12)                timer.
(8)             Examples
(8)             --------
(8)             >>> timer = fig.canvas.new_timer(callbacks=[(f1, (1,), {'a': 3})])
(8)             """
(8)             return self._timer_cls(interval=interval, callbacks=callbacks)
(4)         def flush_events(self):
(8)             """
(8)             Flush the GUI events for the figure.
(8)             Interactive backends need to reimplement this method.
(8)             """
(4)         def start_event_loop(self, timeout=0):
(8)             """
(8)             Start a blocking event loop.
(8)             Such an event loop is used by interactive functions, such as
(8)             `~.Figure.ginput` and `~.Figure.waitforbuttonpress`, to wait for
(8)             events.
(8)             The event loop blocks until a callback function triggers
(8)             `stop_event_loop`, or *timeout* is reached.
(8)             If *timeout* is 0 or negative, never timeout.
(8)             Only interactive backends need to reimplement this method and it
relies
(8)             on `flush_events` being properly implemented.
(8)             Interactive backends should implement this in a more native way.
(8)             """
(8)             if timeout <= 0:
(12)                timeout = np.inf
(8)             timestep = 0.01
(8)             counter = 0
(8)             self._looping = True
(8)             while self._looping and counter * timestep < timeout:
(12)                self.flush_events()
(12)                time.sleep(timestep)
(12)                counter += 1
(4)         def stop_event_loop(self):
(8)             """
(8)             Stop the current blocking event loop.
(8)             Interactive backends need to reimplement this to match
(8)             `start_event_loop`
(8)             """
(8)             self._looping = False
```

```
(0)           def key_press_handler(event, canvas=None, toolbar=None):
(4)               """
(4)               Implement the default Matplotlib key bindings for the canvas and toolbar
(4)               described at :ref:`key-event-handling`.
(4)               Parameters
(4)               ----------
(4)               event : `KeyEvent`
(8)                   A key press/release event.
(4)               canvas : `FigureCanvasBase`, default: ``event.canvas``
(8)                   The backend-specific canvas instance.  This parameter is kept for
(8)                   back-compatibility, but, if set, should always be equal to
(8)                   ``event.canvas``.
(4)               toolbar : `NavigationToolbar2`, default: ``event.canvas.toolbar``
(8)                   The navigation cursor toolbar.  This parameter is kept for
(8)                   back-compatibility, but, if set, should always be equal to
(8)                   ``event.canvas.toolbar``.
(4)               """
(4)               if event.key is None:
(8)                   return
(4)               if canvas is None:
(8)                   canvas = event.canvas
(4)               if toolbar is None:
(8)                   toolbar = canvas.toolbar
(4)               if event.key in rcParams['keymap.fullscreen']:
(8)                   try:
(12)                      canvas.manager.full_screen_toggle()
(8)                   except AttributeError:
(12)                      pass
(4)               if event.key in rcParams['keymap.quit']:
(8)                   Gcf.destroy_fig(canvas.figure)
(4)               if event.key in rcParams['keymap.quit_all']:
(8)                   Gcf.destroy_all()
(4)               if toolbar is not None:
(8)                   if event.key in rcParams['keymap.home']:
(12)                      toolbar.home()
(8)                   elif event.key in rcParams['keymap.back']:
(12)                      toolbar.back()
(8)                   elif event.key in rcParams['keymap.forward']:
(12)                      toolbar.forward()
(8)                   elif event.key in rcParams['keymap.pan']:
(12)                      toolbar.pan()
(12)                      toolbar._update_cursor(event)
(8)                   elif event.key in rcParams['keymap.zoom']:
(12)                      toolbar.zoom()
(12)                      toolbar._update_cursor(event)
(8)                   elif event.key in rcParams['keymap.save']:
(12)                      toolbar.save_figure()
(4)               if event.inaxes is None:
(8)                   return
(4)               def _get_uniform_gridstate(ticks):
(8)                   return (True if all(tick.gridline.get_visible() for tick in ticks)
else
(16)                          False if not any(tick.gridline.get_visible() for tick in
ticks) else
(16)                          None)
(4)               ax = event.inaxes
(4)               if (event.key in rcParams['keymap.grid']
(12)                      and None not in [_get_uniform_gridstate(ax.xaxis.minorTicks),
(29)                                       _get_uniform_gridstate(ax.yaxis.minorTicks)]):
(8)                   x_state = _get_uniform_gridstate(ax.xaxis.majorTicks)
(8)                   y_state = _get_uniform_gridstate(ax.yaxis.majorTicks)
(8)                   cycle = [(False, False), (True, False), (True, True), (False, True)]
(8)                   try:
(12)                      x_state, y_state = (
(16)                          cycle[(cycle.index((x_state, y_state)) + 1) % len(cycle)])
(8)                   except ValueError:
(12)                      pass
(8)                   else:
(12)                      ax.grid(x_state, which="major" if x_state else "both", axis="x")
```

```
1987: (12)                      ax.grid(y_state, which="major" if y_state else "both", axis="y")
1988: (12)                      canvas.draw_idle()
1989: (4)               if (event.key in rcParams['keymap.grid_minor']
1990: (12)                      and None not in [_get_uniform_gridstate(ax.xaxis.majorTicks),
1991: (29)                                       _get_uniform_gridstate(ax.yaxis.majorTicks)]):
1992: (8)                   x_state = _get_uniform_gridstate(ax.xaxis.minorTicks)
1993: (8)                   y_state = _get_uniform_gridstate(ax.yaxis.minorTicks)
1994: (8)                   cycle = [(False, False), (True, False), (True, True), (False, True)]
1995: (8)                   try:
1996: (12)                      x_state, y_state = (
1997: (16)                          cycle[(cycle.index((x_state, y_state)) + 1) % len(cycle)])
1998: (8)                   except ValueError:
1999: (12)                      pass
2000: (8)                   else:
2001: (12)                      ax.grid(x_state, which="both", axis="x")
2002: (12)                      ax.grid(y_state, which="both", axis="y")
2003: (12)                      canvas.draw_idle()
2004: (4)               elif event.key in rcParams['keymap.yscale']:
2005: (8)                   scale = ax.get_yscale()
2006: (8)                   if scale == 'log':
2007: (12)                      ax.set_yscale('linear')
2008: (12)                      ax.figure.canvas.draw_idle()
2009: (8)                   elif scale == 'linear':
2010: (12)                      try:
2011: (16)                          ax.set_yscale('log')
2012: (12)                      except ValueError as exc:
2013: (16)                          _log.warning(str(exc))
2014: (16)                          ax.set_yscale('linear')
2015: (12)                      ax.figure.canvas.draw_idle()
2016: (4)               elif event.key in rcParams['keymap.xscale']:
2017: (8)                   scalex = ax.get_xscale()
2018: (8)                   if scalex == 'log':
2019: (12)                      ax.set_xscale('linear')
2020: (12)                      ax.figure.canvas.draw_idle()
2021: (8)                   elif scalex == 'linear':
2022: (12)                      try:
2023: (16)                          ax.set_xscale('log')
2024: (12)                      except ValueError as exc:
2025: (16)                          _log.warning(str(exc))
2026: (16)                          ax.set_xscale('linear')
2027: (12)                      ax.figure.canvas.draw_idle()
2028: (0)           def button_press_handler(event, canvas=None, toolbar=None):
2029: (4)               """
2030: (4)               The default Matplotlib button actions for extra mouse buttons.
2031: (4)               Parameters are as for `key_press_handler`, except that *event* is a
2032: (4)               `MouseEvent`.
2033: (4)               """
2034: (4)               if canvas is None:
2035: (8)                   canvas = event.canvas
2036: (4)               if toolbar is None:
2037: (8)                   toolbar = canvas.toolbar
2038: (4)               if toolbar is not None:
2039: (8)                   button_name = str(MouseButton(event.button))
2040: (8)                   if button_name in rcParams['keymap.back']:
2041: (12)                      toolbar.back()
2042: (8)                   elif button_name in rcParams['keymap.forward']:
2043: (12)                      toolbar.forward()
2044: (0)           class NonGuiException(Exception):
2045: (4)               """Raised when trying show a figure in a non-GUI backend."""
2046: (4)               pass
2047: (0)           class FigureManagerBase:
2048: (4)               """
2049: (4)               A backend-independent abstraction of a figure container and controller.
2050: (4)               The figure manager is used by pyplot to interact with the window in a
2051: (4)               backend-independent way. It's an adapter for the real (GUI) framework that
2052: (4)               represents the visual figure on screen.
2053: (4)               The figure manager is connected to a specific canvas instance, which in
turn
2054: (4)               is connected to a specific figure instance. To access a figure manager for
```

```
(4)                 a given figure in user code, you typically use ``fig.canvas.manager``.
(4)                 GUI backends derive from this class to translate common operations such
(4)                 as *show* or *resize* to the GUI-specific code. Non-GUI backends do not
(4)                 support these operations and can just use the base class.
(4)                 This following basic operations are accessible:
(4)                 **Window operations**
(4)                 - `~.FigureManagerBase.show`
(4)                 - `~.FigureManagerBase.destroy`
(4)                 - `~.FigureManagerBase.full_screen_toggle`
(4)                 - `~.FigureManagerBase.resize`
(4)                 - `~.FigureManagerBase.get_window_title`
(4)                 - `~.FigureManagerBase.set_window_title`
(4)                 **Key and mouse button press handling**
(4)                 The figure manager sets up default key and mouse button press handling by
(4)                 hooking up the `.key_press_handler` to the matplotlib event system. This
(4)                 ensures the same shortcuts and mouse actions across backends.
(4)                 **Other operations**
(4)                 Subclasses will have additional attributes and functions to access
(4)                 additional functionality. This is of course backend-specific. For example,
(4)                 most GUI backends have ``window`` and ``toolbar`` attributes that give
(4)                 access to the native GUI widgets of the respective framework.
(4)                 Attributes
(4)                 ----------
(4)                 canvas : `FigureCanvasBase`
(8)                     The backend-specific canvas instance.
(4)                 num : int or str
(8)                     The figure number.
(4)                 key_press_handler_id : int
(8)                     The default key handler cid, when using the toolmanager.
(8)                     To disable the default key press handling use::
(12)                        figure.canvas.mpl_disconnect(
(16)                            figure.canvas.manager.key_press_handler_id)
(4)                 button_press_handler_id : int
(8)                     The default mouse button handler cid, when using the toolmanager.
(8)                     To disable the default button press handling use::
(12)                        figure.canvas.mpl_disconnect(
(16)                            figure.canvas.manager.button_press_handler_id)
(4)                 """
(4)                 _toolbar2_class = None
(4)                 _toolmanager_toolbar_class = None
(4)                 def __init__(self, canvas, num):
(8)                     self.canvas = canvas
(8)                     canvas.manager = self  # store a pointer to parent
(8)                     self.num = num
(8)                     self.set_window_title(f"Figure {num:d}")
(8)                     self.key_press_handler_id = None
(8)                     self.button_press_handler_id = None
(8)                     if rcParams['toolbar'] != 'toolmanager':
(12)                        self.key_press_handler_id = self.canvas.mpl_connect(
(16)                            'key_press_event', key_press_handler)
(12)                        self.button_press_handler_id = self.canvas.mpl_connect(
(16)                            'button_press_event', button_press_handler)
(8)                     self.toolmanager = (ToolManager(canvas.figure)
(28)                                        if mpl.rcParams['toolbar'] == 'toolmanager'
(28)                                        else None)
(8)                     if (mpl.rcParams["toolbar"] == "toolbar2"
(16)                            and self._toolbar2_class):
(12)                        self.toolbar = self._toolbar2_class(self.canvas)
(8)                     elif (mpl.rcParams["toolbar"] == "toolmanager"
(16)                            and self._toolmanager_toolbar_class):
(12)                        self.toolbar = self._toolmanager_toolbar_class(self.toolmanager)
(8)                     else:
(12)                        self.toolbar = None
(8)                     if self.toolmanager:
(12)                        tools.add_tools_to_manager(self.toolmanager)
(12)                        if self.toolbar:
(16)                            tools.add_tools_to_container(self.toolbar)
(8)                     @self.canvas.figure.add_axobserver
(8)                     def notify_axes_change(fig):
```

```
(12)                        if self.toolmanager is None and self.toolbar is not None:
(16)                            self.toolbar.update()
(4)                 @classmethod
(4)                 def create_with_canvas(cls, canvas_class, figure, num):
(8)                     """
(8)                     Create a manager for a given *figure* using a specific *canvas_class*.
(8)                     Backends should override this method if they have specific needs for
(8)                     setting up the canvas or the manager.
(8)                     """
(8)                     return cls(canvas_class(figure), num)
(4)                 @classmethod
(4)                 def start_main_loop(cls):
(8)                     """
(8)                     Start the main event loop.
(8)                     This method is called by `.FigureManagerBase.pyplot_show`, which is
the
(8)                     implementation of `.pyplot.show`.  To customize the behavior of
(8)                     `.pyplot.show`, interactive backends should usually override
(8)                     `~.FigureManagerBase.start_main_loop`; if more customized logic is
(8)                     necessary, `~.FigureManagerBase.pyplot_show` can also be overridden.
(8)                     """
(4)                 @classmethod
(4)                 def pyplot_show(cls, *, block=None):
(8)                     """
(8)                     Show all figures.  This method is the implementation of
`.pyplot.show`.
(8)                     To customize the behavior of `.pyplot.show`, interactive backends
(8)                     should usually override `~.FigureManagerBase.start_main_loop`; if more
(8)                     customized logic is necessary, `~.FigureManagerBase.pyplot_show` can
(8)                     also be overridden.
(8)                     Parameters
(8)                     ----------
(8)                     block : bool, optional
(12)                        Whether to block by calling ``start_main_loop``.  The default,
(12)                        None, means to block if we are neither in IPython's ``%pylab``
mode
(12)                        nor in ``interactive`` mode.
(8)                     """
(8)                     managers = Gcf.get_all_fig_managers()
(8)                     if not managers:
(12)                        return
(8)                     for manager in managers:
(12)                        try:
(16)                            manager.show()  # Emits a warning for non-interactive backend.
(12)                        except NonGuiException as exc:
(16)                            _api.warn_external(str(exc))
(8)                     if block is None:
(12)                        pyplot_show = getattr(sys.modules.get("matplotlib.pyplot"),
"show", None)
(12)                        ipython_pylab = hasattr(pyplot_show, "_needmain")
(12)                        block = not ipython_pylab and not is_interactive()
(8)                     if block:
(12)                        cls.start_main_loop()
(4)                 def show(self):
(8)                     """
(8)                     For GUI backends, show the figure window and redraw.
(8)                     For non-GUI backends, raise an exception, unless running headless
(i.e.
(8)                     on Linux with an unset DISPLAY); this exception is converted to a
(8)                     warning in `.Figure.show`.
(8)                     """
(8)                     if sys.platform == "linux" and not os.environ.get("DISPLAY"):
(12)                        return
(8)                     raise NonGuiException(
(12)                        f"{type(self.canvas).__name__} is non-interactive, and thus cannot
be "
(12)                        f"shown")
(4)                 def destroy(self):
(8)                     pass
```

```
2187: (4)               def full_screen_toggle(self):
2188: (8)                   pass
2189: (4)               def resize(self, w, h):
2190: (8)                   """For GUI backends, resize the window (in physical pixels)."""
2191: (4)               def get_window_title(self):
2192: (8)                   """
2193: (8)                   Return the title text of the window containing the figure, or None
2194: (8)                   if there is no window (e.g., a PS backend).
2195: (8)                   """
2196: (8)                   return 'image'
2197: (4)               def set_window_title(self, title):
2198: (8)                   """
2199: (8)                   Set the title text of the window containing the figure.
2200: (8)                   This has no effect for non-GUI (e.g., PS) backends.
2201: (8)                   Examples
2202: (8)                   --------
2203: (8)                   >>> fig = plt.figure()
2204: (8)                   >>> fig.canvas.manager.set_window_title('My figure')
2205: (8)                   """
2206: (0)           cursors = tools.cursors
2207: (0)           class _Mode(str, Enum):
2208: (4)               NONE = ""
2209: (4)               PAN = "pan/zoom"
2210: (4)               ZOOM = "zoom rect"
2211: (4)               def __str__(self):
2212: (8)                   return self.value
2213: (4)               @property
2214: (4)               def _navigate_mode(self):
2215: (8)                   return self.name if self is not _Mode.NONE else None
2216: (0)           class NavigationToolbar2:
2217: (4)               """
2218: (4)               Base class for the navigation cursor, version 2.
2219: (4)               Backends must implement a canvas that handles connections for
2220: (4)               'button_press_event' and 'button_release_event'.  See
2221: (4)               :meth:`FigureCanvasBase.mpl_connect` for more information.
2222: (4)               They must also define
2223: (4)               :meth:`save_figure`
2224: (8)                   Save the current figure.
2225: (4)               :meth:`draw_rubberband` (optional)
2226: (8)                   Draw the zoom to rect "rubberband" rectangle.
2227: (4)               :meth:`set_message` (optional)
2228: (8)                   Display message.
2229: (4)               :meth:`set_history_buttons` (optional)
2230: (8)                   You can change the history back / forward buttons to indicate disabled
/ enabled
2231: (8)                   state.
2232: (4)               and override ``__init__`` to set up the toolbar -- without forgetting to
2233: (4)               call the base-class init.  Typically, ``__init__`` needs to set up toolbar
2234: (4)               buttons connected to the `home`, `back`, `forward`, `pan`, `zoom`, and
2235: (4)               `save_figure` methods and using standard icons in the "images"
subdirectory
2236: (4)               of the data path.
2237: (4)               That's it, we'll do the rest!
2238: (4)               """
2239: (4)               toolitems = (
2240: (8)                   ('Home', 'Reset original view', 'home', 'home'),
2241: (8)                   ('Back', 'Back to previous view', 'back', 'back'),
2242: (8)                   ('Forward', 'Forward to next view', 'forward', 'forward'),
2243: (8)                   (None, None, None, None),
2244: (8)                   ('Pan',
2245: (9)                    'Left button pans, Right button zooms\n'
2246: (9)                    'x/y fixes axis, CTRL fixes aspect',
2247: (9)                    'move', 'pan'),
2248: (8)                   ('Zoom', 'Zoom to rectangle\nx/y fixes axis', 'zoom_to_rect', 'zoom'),
2249: (8)                   ('Subplots', 'Configure subplots', 'subplots', 'configure_subplots'),
2250: (8)                   (None, None, None, None),
2251: (8)                   ('Save', 'Save the figure', 'filesave', 'save_figure'),
2252: (6)                 )
2253: (4)               def __init__(self, canvas):
```

```
(8)                 self.canvas = canvas
(8)                 canvas.toolbar = self
(8)                 self._nav_stack = cbook._Stack()
(8)                 self._last_cursor = tools.Cursors.POINTER
(8)                 self._id_press = self.canvas.mpl_connect(
(12)                    'button_press_event', self._zoom_pan_handler)
(8)                 self._id_release = self.canvas.mpl_connect(
(12)                    'button_release_event', self._zoom_pan_handler)
(8)                 self._id_drag = self.canvas.mpl_connect(
(12)                    'motion_notify_event', self.mouse_move)
(8)                 self._pan_info = None
(8)                 self._zoom_info = None
(8)                 self.mode = _Mode.NONE  # a mode string for the status bar
(8)                 self.set_history_buttons()
(4)             def set_message(self, s):
(8)                 """Display a message on toolbar or in status bar."""
(4)             def draw_rubberband(self, event, x0, y0, x1, y1):
(8)                 """
(8)                 Draw a rectangle rubberband to indicate zoom limits.
(8)                 Note that it is not guaranteed that ``x0 <= x1`` and ``y0 <= y1``.
(8)                 """
(4)             def remove_rubberband(self):
(8)                 """Remove the rubberband."""
(4)             def home(self, *args):
(8)                 """
(8)                 Restore the original view.
(8)                 For convenience of being directly connected as a GUI callback, which
(8)                 often get passed additional parameters, this method accepts arbitrary
(8)                 parameters, but does not use them.
(8)                 """
(8)                 self._nav_stack.home()
(8)                 self.set_history_buttons()
(8)                 self._update_view()
(4)             def back(self, *args):
(8)                 """
(8)                 Move back up the view lim stack.
(8)                 For convenience of being directly connected as a GUI callback, which
(8)                 often get passed additional parameters, this method accepts arbitrary
(8)                 parameters, but does not use them.
(8)                 """
(8)                 self._nav_stack.back()
(8)                 self.set_history_buttons()
(8)                 self._update_view()
(4)             def forward(self, *args):
(8)                 """
(8)                 Move forward in the view lim stack.
(8)                 For convenience of being directly connected as a GUI callback, which
(8)                 often get passed additional parameters, this method accepts arbitrary
(8)                 parameters, but does not use them.
(8)                 """
(8)                 self._nav_stack.forward()
(8)                 self.set_history_buttons()
(8)                 self._update_view()
(4)             def _update_cursor(self, event):
(8)                 """
(8)                 Update the cursor after a mouse move event or a tool (de)activation.
(8)                 """
(8)                 if self.mode and event.inaxes and event.inaxes.get_navigate():
(12)                    if (self.mode == _Mode.ZOOM
(20)                            and self._last_cursor != tools.Cursors.SELECT_REGION):
(16)                        self.canvas.set_cursor(tools.Cursors.SELECT_REGION)
(16)                        self._last_cursor = tools.Cursors.SELECT_REGION
(12)                    elif (self.mode == _Mode.PAN
(18)                          and self._last_cursor != tools.Cursors.MOVE):
(16)                        self.canvas.set_cursor(tools.Cursors.MOVE)
(16)                        self._last_cursor = tools.Cursors.MOVE
(8)                 elif self._last_cursor != tools.Cursors.POINTER:
(12)                    self.canvas.set_cursor(tools.Cursors.POINTER)
(12)                    self._last_cursor = tools.Cursors.POINTER
```

```
(4)     @contextmanager
(4)     def _wait_cursor_for_draw_cm(self):
(8)         """
(8)         Set the cursor to a wait cursor when drawing the canvas.
(8)         In order to avoid constantly changing the cursor when the canvas
(8)         changes frequently, do nothing if this context was triggered during
the
(8)         last second.  (Optimally we'd prefer only setting the wait cursor if
(8)         the *current* draw takes too long, but the current draw blocks the GUI
(8)         thread).
(8)         """
(8)         self._draw_time, last_draw_time = (
(12)            time.time(), getattr(self, "_draw_time", -np.inf))
(8)         if self._draw_time - last_draw_time > 1:
(12)            try:
(16)                self.canvas.set_cursor(tools.Cursors.WAIT)
(16)                yield
(12)            finally:
(16)                self.canvas.set_cursor(self._last_cursor)
(8)         else:
(12)            yield
(4)     @staticmethod
(4)     def _mouse_event_to_message(event):
(8)         if event.inaxes and event.inaxes.get_navigate():
(12)            try:
(16)                s = event.inaxes.format_coord(event.xdata, event.ydata)
(12)            except (ValueError, OverflowError):
(16)                pass
(12)            else:
(16)                s = s.rstrip()
(16)                artists = [a for a in event.inaxes._mouseover_set
(27)                           if a.contains(event)[0] and a.get_visible()]
(16)                if artists:
(20)                    a = cbook._topmost_artist(artists)
(20)                    if a is not event.inaxes.patch:
(24)                        data = a.get_cursor_data(event)
(24)                        if data is not None:
(28)                            data_str = a.format_cursor_data(data).rstrip()
(28)                            if data_str:
(32)                                s = s + '\n' + data_str
(16)                return s
(8)         return ""
(4)     def mouse_move(self, event):
(8)         self._update_cursor(event)
(8)         self.set_message(self._mouse_event_to_message(event))
(4)     def _zoom_pan_handler(self, event):
(8)         if self.mode == _Mode.PAN:
(12)            if event.name == "button_press_event":
(16)                self.press_pan(event)
(12)            elif event.name == "button_release_event":
(16)                self.release_pan(event)
(8)         if self.mode == _Mode.ZOOM:
(12)            if event.name == "button_press_event":
(16)                self.press_zoom(event)
(12)            elif event.name == "button_release_event":
(16)                self.release_zoom(event)
(4)     def _start_event_axes_interaction(self, event, *, method):
(8)         def _ax_filter(ax):
(12)            return (ax.in_axes(event) and
(20)                    ax.get_navigate() and
(20)                    getattr(ax, f"can_{method}")()
(20)                    )
(8)         def _capture_events(ax):
(12)            f = ax.get_forward_navigation_events()
(12)            if f == "auto":  # (capture = patch visibility)
(16)                f = not ax.patch.get_visible()
(12)            return not f
(8)         axes = list(filter(_ax_filter, self.canvas.figure.get_axes()))
(8)         if len(axes) == 0:
```

```
2391: (12)                      return []
2392: (8)                   if self._nav_stack() is None:
2393: (12)                      self.push_current()   # Set the home button to this view.
2394: (8)                   grps = dict()
2395: (8)                   for ax in reversed(axes):
2396: (12)                      grps.setdefault(ax.get_zorder(), []).append(ax)
2397: (8)                   axes_to_trigger = []
2398: (8)                   for zorder in sorted(grps, reverse=True):
2399: (12)                      for ax in grps[zorder]:
2400: (16)                          axes_to_trigger.append(ax)
2401: (16)                          axes_to_trigger.extend(ax._twinned_axes.get_siblings(ax))
2402: (16)                          if _capture_events(ax):
2403: (20)                              break  # break if we hit a capturing axes
2404: (12)                      else:
2405: (16)                          continue
2406: (12)                      break
2407: (8)                   axes_to_trigger = list(dict.fromkeys(axes_to_trigger))
2408: (8)                   return axes_to_trigger
2409: (4)               def pan(self, *args):
2410: (8)                   """
2411: (8)                   Toggle the pan/zoom tool.
2412: (8)                   Pan with left button, zoom with right.
2413: (8)                   """
2414: (8)                   if not self.canvas.widgetlock.available(self):
2415: (12)                      self.set_message("pan unavailable")
2416: (12)                      return
2417: (8)                   if self.mode == _Mode.PAN:
2418: (12)                      self.mode = _Mode.NONE
2419: (12)                      self.canvas.widgetlock.release(self)
2420: (8)                   else:
2421: (12)                      self.mode = _Mode.PAN
2422: (12)                      self.canvas.widgetlock(self)
2423: (8)                   for a in self.canvas.figure.get_axes():
2424: (12)                      a.set_navigate_mode(self.mode._navigate_mode)
2425: (4)               _PanInfo = namedtuple("_PanInfo", "button axes cid")
2426: (4)               def press_pan(self, event):
2427: (8)                   """Callback for mouse button press in pan/zoom mode."""
2428: (8)                   if (event.button not in [MouseButton.LEFT, MouseButton.RIGHT]
2429: (16)                          or event.x is None or event.y is None):
2430: (12)                      return
2431: (8)                   axes = self._start_event_axes_interaction(event, method="pan")
2432: (8)                   if not axes:
2433: (12)                      return
2434: (8)                   for ax in axes:
2435: (12)                      ax.start_pan(event.x, event.y, event.button)
2436: (8)                   self.canvas.mpl_disconnect(self._id_drag)
2437: (8)                   id_drag = self.canvas.mpl_connect("motion_notify_event",
self.drag_pan)
2438: (8)                   self._pan_info = self._PanInfo(
2439: (12)                      button=event.button, axes=axes, cid=id_drag)
2440: (4)               def drag_pan(self, event):
2441: (8)                   """Callback for dragging in pan/zoom mode."""
2442: (8)                   for ax in self._pan_info.axes:
2443: (12)                      ax.drag_pan(self._pan_info.button, event.key, event.x, event.y)
2444: (8)                   self.canvas.draw_idle()
2445: (4)               def release_pan(self, event):
2446: (8)                   """Callback for mouse button release in pan/zoom mode."""
2447: (8)                   if self._pan_info is None:
2448: (12)                      return
2449: (8)                   self.canvas.mpl_disconnect(self._pan_info.cid)
2450: (8)                   self._id_drag = self.canvas.mpl_connect(
2451: (12)                      'motion_notify_event', self.mouse_move)
2452: (8)                   for ax in self._pan_info.axes:
2453: (12)                      ax.end_pan()
2454: (8)                   self.canvas.draw_idle()
2455: (8)                   self._pan_info = None
2456: (8)                   self.push_current()
2457: (4)               def zoom(self, *args):
2458: (8)                   if not self.canvas.widgetlock.available(self):
```

```
(12)                        self.set_message("zoom unavailable")
(12)                        return
(8)                     """Toggle zoom to rect mode."""
(8)                     if self.mode == _Mode.ZOOM:
(12)                        self.mode = _Mode.NONE
(12)                        self.canvas.widgetlock.release(self)
(8)                     else:
(12)                        self.mode = _Mode.ZOOM
(12)                        self.canvas.widgetlock(self)
(8)                     for a in self.canvas.figure.get_axes():
(12)                        a.set_navigate_mode(self.mode._navigate_mode)
(4)                 _ZoomInfo = namedtuple("_ZoomInfo", "direction start_xy axes cid cbar")
(4)                 def press_zoom(self, event):
(8)                     """Callback for mouse button press in zoom to rect mode."""
(8)                     if (event.button not in [MouseButton.LEFT, MouseButton.RIGHT]
(16)                            or event.x is None or event.y is None):
(12)                        return
(8)                     axes = self._start_event_axes_interaction(event, method="zoom")
(8)                     if not axes:
(12)                        return
(8)                     id_zoom = self.canvas.mpl_connect(
(12)                        "motion_notify_event", self.drag_zoom)
(8)                     parent_ax = axes[0]
(8)                     if hasattr(parent_ax, "_colorbar"):
(12)                        cbar = parent_ax._colorbar.orientation
(8)                     else:
(12)                        cbar = None
(8)                     self._zoom_info = self._ZoomInfo(
(12)                        direction="in" if event.button == 1 else "out",
(12)                        start_xy=(event.x, event.y), axes=axes, cid=id_zoom, cbar=cbar)
(4)                 def drag_zoom(self, event):
(8)                     """Callback for dragging in zoom mode."""
(8)                     start_xy = self._zoom_info.start_xy
(8)                     ax = self._zoom_info.axes[0]
(8)                     (x1, y1), (x2, y2) = np.clip(
(12)                        [start_xy, [event.x, event.y]], ax.bbox.min, ax.bbox.max)
(8)                     key = event.key
(8)                     if self._zoom_info.cbar == "horizontal":
(12)                        key = "x"
(8)                     elif self._zoom_info.cbar == "vertical":
(12)                        key = "y"
(8)                     if key == "x":
(12)                        y1, y2 = ax.bbox.intervaly
(8)                     elif key == "y":
(12)                        x1, x2 = ax.bbox.intervalx
(8)                     self.draw_rubberband(event, x1, y1, x2, y2)
(4)                 def release_zoom(self, event):
(8)                     """Callback for mouse button release in zoom to rect mode."""
(8)                     if self._zoom_info is None:
(12)                        return
(8)                     self.canvas.mpl_disconnect(self._zoom_info.cid)
(8)                     self.remove_rubberband()
(8)                     start_x, start_y = self._zoom_info.start_xy
(8)                     key = event.key
(8)                     if self._zoom_info.cbar == "horizontal":
(12)                        key = "x"
(8)                     elif self._zoom_info.cbar == "vertical":
(12)                        key = "y"
(8)                     if ((abs(event.x - start_x) < 5 and key != "y") or
(16)                            (abs(event.y - start_y) < 5 and key != "x")):
(12)                        self.canvas.draw_idle()
(12)                        self._zoom_info = None
(12)                        return
(8)                     for i, ax in enumerate(self._zoom_info.axes):
(12)                        twinx = any(ax.get_shared_x_axes().joined(ax, prev)
(24)                                    for prev in self._zoom_info.axes[:i])
(12)                        twiny = any(ax.get_shared_y_axes().joined(ax, prev)
(24)                                    for prev in self._zoom_info.axes[:i])
(12)                        ax._set_view_from_bbox(
```

```
2528: (16)                          (start_x, start_y, event.x, event.y),
2529: (16)                          self._zoom_info.direction, key, twinx, twiny)
2530: (8)                   self.canvas.draw_idle()
2531: (8)                   self._zoom_info = None
2532: (8)                   self.push_current()
2533: (4)               def push_current(self):
2534: (8)                   """Push the current view limits and position onto the stack."""
2535: (8)                   self._nav_stack.push(
2536: (12)                      WeakKeyDictionary(
2537: (16)                          {ax: (ax._get_view(),
2538: (22)                                (ax.get_position(True).frozen(),
2539: (23)                                 ax.get_position().frozen()))
2540: (17)                           for ax in self.canvas.figure.axes}))
2541: (8)                   self.set_history_buttons()
2542: (4)               def _update_view(self):
2543: (8)                   """
2544: (8)                   Update the viewlim and position from the view and position stack for
2545: (8)                   each Axes.
2546: (8)                   """
2547: (8)                   nav_info = self._nav_stack()
2548: (8)                   if nav_info is None:
2549: (12)                      return
2550: (8)                   items = list(nav_info.items())
2551: (8)                   for ax, (view, (pos_orig, pos_active)) in items:
2552: (12)                      ax._set_view(view)
2553: (12)                      ax._set_position(pos_orig, 'original')
2554: (12)                      ax._set_position(pos_active, 'active')
2555: (8)                   self.canvas.draw_idle()
2556: (4)               def configure_subplots(self, *args):
2557: (8)                   if hasattr(self, "subplot_tool"):
2558: (12)                      self.subplot_tool.figure.canvas.manager.show()
2559: (12)                      return
2560: (8)                   from matplotlib.figure import Figure
2561: (8)                   with mpl.rc_context({"toolbar": "none"}):  # No navbar for the
toolfig.
2562: (12)                      manager = type(self.canvas).new_manager(Figure(figsize=(6, 3)),
-1)
2563: (8)                   manager.set_window_title("Subplot configuration tool")
2564: (8)                   tool_fig = manager.canvas.figure
2565: (8)                   tool_fig.subplots_adjust(top=0.9)
2566: (8)                   self.subplot_tool = widgets.SubplotTool(self.canvas.figure, tool_fig)
2567: (8)                   cid = self.canvas.mpl_connect(
2568: (12)                      "close_event", lambda e: manager.destroy())
2569: (8)                   def on_tool_fig_close(e):
2570: (12)                      self.canvas.mpl_disconnect(cid)
2571: (12)                      del self.subplot_tool
2572: (8)                   tool_fig.canvas.mpl_connect("close_event", on_tool_fig_close)
2573: (8)                   manager.show()
2574: (8)                   return self.subplot_tool
2575: (4)               def save_figure(self, *args):
2576: (8)                   """Save the current figure."""
2577: (8)                   raise NotImplementedError
2578: (4)               def update(self):
2579: (8)                   """Reset the Axes stack."""
2580: (8)                   self._nav_stack.clear()
2581: (8)                   self.set_history_buttons()
2582: (4)               def set_history_buttons(self):
2583: (8)                   """Enable or disable the back/forward button."""
2584: (0)           class ToolContainerBase:
2585: (4)               """
2586: (4)               Base class for all tool containers, e.g. toolbars.
2587: (4)               Attributes
2588: (4)               ----------
2589: (4)               toolmanager : `.ToolManager`
2590: (8)                   The tools with which this `ToolContainer` wants to communicate.
2591: (4)               """
2592: (4)               _icon_extension = '.png'
2593: (4)               """
2594: (4)               Toolcontainer button icon image format extension
```

```
(4)                 **String**: Image extension
(4)                 """
(4)                 def __init__(self, toolmanager):
(8)                     self.toolmanager = toolmanager
(8)                     toolmanager.toolmanager_connect(
(12)                        'tool_message_event',
(12)                        lambda event: self.set_message(event.message))
(8)                     toolmanager.toolmanager_connect(
(12)                        'tool_removed_event',
(12)                        lambda event: self.remove_toolitem(event.tool.name))
(4)                 def _tool_toggled_cbk(self, event):
(8)                     """
(8)                     Capture the 'tool_trigger_[name]'
(8)                     This only gets used for toggled tools.
(8)                     """
(8)                     self.toggle_toolitem(event.tool.name, event.tool.toggled)
(4)                 def add_tool(self, tool, group, position=-1):
(8)                     """
(8)                     Add a tool to this container.
(8)                     Parameters
(8)                     ----------
(8)                     tool : tool_like
(12)                        The tool to add, see `.ToolManager.get_tool`.
(8)                     group : str
(12)                        The name of the group to add this tool to.
(8)                     position : int, default: -1
(12)                        The position within the group to place this tool.
(8)                     """
(8)                     tool = self.toolmanager.get_tool(tool)
(8)                     image = self._get_image_filename(tool)
(8)                     toggle = getattr(tool, 'toggled', None) is not None
(8)                     self.add_toolitem(tool.name, group, position,
(26)                                      image, tool.description, toggle)
(8)                     if toggle:
(12)                        self.toolmanager.toolmanager_connect('tool_trigger_%s' %
tool.name,
(49)                                                             self._tool_toggled_cbk)
(12)                        if tool.toggled:
(16)                            self.toggle_toolitem(tool.name, True)
(4)                 def _get_image_filename(self, tool):
(8)                     """Resolve a tool icon's filename."""
(8)                     if not tool.image:
(12)                        return None
(8)                     if os.path.isabs(tool.image):
(12)                        filename = tool.image
(8)                     else:
(12)                        if "image" in getattr(tool, "__dict__", {}):
(16)                            raise ValueError("If 'tool.image' is an instance variable, "
(33)                                             "it must be an absolute path")
(12)                        for cls in type(tool).__mro__:
(16)                            if "image" in vars(cls):
(20)                                try:
(24)                                    src = inspect.getfile(cls)
(24)                                    break
(20)                                except (OSError, TypeError):
(24)                                    raise ValueError("Failed to locate source file "
(41)                                                     "where 'tool.image' is defined") from
None
(12)                        else:
(16)                            raise ValueError("Failed to find parent class defining
'tool.image'")
(12)                        filename = str(pathlib.Path(src).parent / tool.image)
(8)                     for filename in [filename, filename + self._icon_extension]:
(12)                        if os.path.isfile(filename):
(16)                            return os.path.abspath(filename)
(8)                     for fname in [  # Fallback; once deprecation elapses.
(12)                        tool.image,
(12)                        tool.image + self._icon_extension,
(12)                        cbook._get_data_path("images", tool.image),
```

```
(12)                    cbook._get_data_path("images", tool.image + self._icon_extension),
(8)                 ]:
(12)                    if os.path.isfile(fname):
(16)                        _api.warn_deprecated(
(20)                            "3.9", message=f"Loading icon {tool.image!r} from the current "
(20)                            "directory or from Matplotlib's image directory.  This behavior "
(20)                            "is deprecated since %(since)s and will be removed %(removal)s; "
(20)                            "Tool.image should be set to a path relative to the Tool's source "
(20)                            "file, or to an absolute path.")
(16)                        return os.path.abspath(fname)
(4)             def trigger_tool(self, name):
(8)                 """
(8)                 Trigger the tool.
(8)                 Parameters
(8)                 ----------
(8)                 name : str
(12)                    Name (id) of the tool triggered from within the container.
(8)                 """
(8)                 self.toolmanager.trigger_tool(name, sender=self)
(4)             def add_toolitem(self, name, group, position, image, description, toggle):
(8)                 """
(8)                 A hook to add a toolitem to the container.
(8)                 This hook must be implemented in each backend and contains the
(8)                 backend-specific code to add an element to the toolbar.
(8)                 .. warning::
(12)                    This is part of the backend implementation and should
(12)                    not be called by end-users.  They should instead call
(12)                    `.ToolContainerBase.add_tool`.
(8)                 The callback associated with the button click event
(8)                 must be *exactly* ``self.trigger_tool(name)``.
(8)                 Parameters
(8)                 ----------
(8)                 name : str
(12)                    Name of the tool to add, this gets used as the tool's ID and as the
(12)                    default label of the buttons.
(8)                 group : str
(12)                    Name of the group that this tool belongs to.
(8)                 position : int
(12)                    Position of the tool within its group, if -1 it goes at the end.
(8)                 image : str
(12)                    Filename of the image for the button or `None`.
(8)                 description : str
(12)                    Description of the tool, used for the tooltips.
(8)                 toggle : bool
(12)                    * `True` : The button is a toggle (change the pressed/unpressed
(14)                      state between consecutive clicks).
(12)                    * `False` : The button is a normal button (returns to unpressed
(14)                      state after release).
(8)                 """
(8)                 raise NotImplementedError
(4)             def toggle_toolitem(self, name, toggled):
(8)                 """
(8)                 A hook to toggle a toolitem without firing an event.
(8)                 This hook must be implemented in each backend and contains the
(8)                 backend-specific code to silently toggle a toolbar element.
(8)                 .. warning::
(12)                    This is part of the backend implementation and should
(12)                    not be called by end-users.  They should instead call
(12)                    `.ToolManager.trigger_tool` or `.ToolContainerBase.trigger_tool`
(12)                    (which are equivalent).
(8)                 Parameters
(8)                 ----------
(8)                 name : str
(12)                    Id of the tool to toggle.
```

```
(8)                 toggled : bool
(12)                    Whether to set this tool as toggled or not.
(8)                 """
(8)                 raise NotImplementedError
(4)             def remove_toolitem(self, name):
(8)                 """
(8)                 A hook to remove a toolitem from the container.
(8)                 This hook must be implemented in each backend and contains the
(8)                 backend-specific code to remove an element from the toolbar; it is
(8)                 called when `.ToolManager` emits a `tool_removed_event`.
(8)                 Because some tools are present only on the `.ToolManager` but not on
(8)                 the `ToolContainer`, this method must be a no-op when called on a tool
(8)                 absent from the container.
(8)                 .. warning::
(12)                    This is part of the backend implementation and should
(12)                    not be called by end-users.  They should instead call
(12)                    `.ToolManager.remove_tool`.
(8)                 Parameters
(8)                 ----------
(8)                 name : str
(12)                    Name of the tool to remove.
(8)                 """
(8)                 raise NotImplementedError
(4)             def set_message(self, s):
(8)                 """
(8)                 Display a message on the toolbar.
(8)                 Parameters
(8)                 ----------
(8)                 s : str
(12)                    Message text.
(8)                 """
(8)                 raise NotImplementedError
(0)         class _Backend:
(4)             backend_version = "unknown"
(4)             FigureCanvas = None
(4)             FigureManager = FigureManagerBase
(4)             mainloop = None
(4)             @classmethod
(4)             def new_figure_manager(cls, num, *args, **kwargs):
(8)                 """Create a new figure manager instance."""
(8)                 from matplotlib.figure import Figure
(8)                 fig_cls = kwargs.pop('FigureClass', Figure)
(8)                 fig = fig_cls(*args, **kwargs)
(8)                 return cls.new_figure_manager_given_figure(num, fig)
(4)             @classmethod
(4)             def new_figure_manager_given_figure(cls, num, figure):
(8)                 """Create a new figure manager instance for the given figure."""
(8)                 return cls.FigureCanvas.new_manager(figure, num)
(4)             @classmethod
(4)             def draw_if_interactive(cls):
(8)                 manager_class = cls.FigureCanvas.manager_class
(8)                 backend_is_interactive = (
(12)                    manager_class.start_main_loop != FigureManagerBase.start_main_loop
(12)                    or manager_class.pyplot_show != FigureManagerBase.pyplot_show)
(8)                 if backend_is_interactive and is_interactive():
(12)                    manager = Gcf.get_active()
(12)                    if manager:
(16)                        manager.canvas.draw_idle()
(4)             @classmethod
(4)             def show(cls, *, block=None):
(8)                 """
(8)                 Show all figures.
(8)                 `show` blocks by calling `mainloop` if *block* is ``True``, or if it
is
(8)                 ``None`` and we are not in `interactive` mode and if IPython's
(8)                 ``%matplotlib`` integration has not been activated.
(8)                 """
(8)                 managers = Gcf.get_all_fig_managers()
(8)                 if not managers:
```

```
2793: (12)                      return
2794: (8)                   for manager in managers:
2795: (12)                      try:
2796: (16)                          manager.show()  # Emits a warning for non-interactive backend.
2797: (12)                      except NonGuiException as exc:
2798: (16)                          _api.warn_external(str(exc))
2799: (8)                   if cls.mainloop is None:
2800: (12)                      return
2801: (8)                   if block is None:
2802: (12)                      pyplot_show = getattr(sys.modules.get("matplotlib.pyplot"),
"show", None)
2803: (12)                      ipython_pylab = hasattr(pyplot_show, "_needmain")
2804: (12)                      block = not ipython_pylab and not is_interactive()
2805: (8)                   if block:
2806: (12)                      cls.mainloop()
2807: (4)               @staticmethod
2808: (4)               def export(cls):
2809: (8)                   for name in [
2810: (16)                          "backend_version",
2811: (16)                          "FigureCanvas",
2812: (16)                          "FigureManager",
2813: (16)                          "new_figure_manager",
2814: (16)                          "new_figure_manager_given_figure",
2815: (16)                          "draw_if_interactive",
2816: (16)                          "show",
2817: (8)                   ]:
2818: (12)                      setattr(sys.modules[cls.__module__], name, getattr(cls, name))
2819: (8)                   class Show(ShowBase):
2820: (12)                      def mainloop(self):
2821: (16)                          return cls.mainloop()
2822: (8)                   setattr(sys.modules[cls.__module__], "Show", Show)
2823: (8)                   return cls
2824: (0)           class ShowBase(_Backend):
2825: (4)               """
2826: (4)               Simple base class to generate a ``show()`` function in backends.
2827: (4)               Subclass must override ``mainloop()`` method.
2828: (4)               """
2829: (4)               def __call__(self, block=None):
2830: (8)                   return self.show(block=block)
```

File 25 - backend_managers.py:

```
1: (0)              from matplotlib import _api, backend_tools, cbook, widgets
2: (0)              class ToolEvent:
3: (4)                  """Event for tool manipulation (add/remove)."""
4: (4)                  def __init__(self, name, sender, tool, data=None):
5: (8)                      self.name = name
6: (8)                      self.sender = sender
7: (8)                      self.tool = tool
8: (8)                      self.data = data
9: (0)              class ToolTriggerEvent(ToolEvent):
10: (4)                 """Event to inform that a tool has been triggered."""
11: (4)                 def __init__(self, name, sender, tool, canvasevent=None, data=None):
12: (8)                     super().__init__(name, sender, tool, data)
13: (8)                     self.canvasevent = canvasevent
14: (0)             class ToolManagerMessageEvent:
15: (4)                 """
16: (4)                 Event carrying messages from toolmanager.
17: (4)                 Messages usually get displayed to the user by the toolbar.
18: (4)                 """
19: (4)                 def __init__(self, name, sender, message):
20: (8)                     self.name = name
21: (8)                     self.sender = sender
22: (8)                     self.message = message
23: (0)             class ToolManager:
24: (4)                 """
25: (4)                 Manager for actions triggered by user interactions (key press, toolbar
```

```
(4)     clicks, ...) on a Figure.
(4)     Attributes
(4)     ----------
(4)     figure : `.Figure`
(4)     keypresslock : `~matplotlib.widgets.LockDraw`
(8)         `.LockDraw` object to know if the `canvas` key_press_event is locked.
(4)     messagelock : `~matplotlib.widgets.LockDraw`
(8)         `.LockDraw` object to know if the message is available to write.
(4)     """
(4)     def __init__(self, figure=None):
(8)         self._key_press_handler_id = None
(8)         self._tools = {}
(8)         self._keys = {}
(8)         self._toggled = {}
(8)         self._callbacks = cbook.CallbackRegistry()
(8)         self.keypresslock = widgets.LockDraw()
(8)         self.messagelock = widgets.LockDraw()
(8)         self._figure = None
(8)         self.set_figure(figure)
(4)     @property
(4)     def canvas(self):
(8)         """Canvas managed by FigureManager."""
(8)         if not self._figure:
(12)            return None
(8)         return self._figure.canvas
(4)     @property
(4)     def figure(self):
(8)         """Figure that holds the canvas."""
(8)         return self._figure
(4)     @figure.setter
(4)     def figure(self, figure):
(8)         self.set_figure(figure)
(4)     def set_figure(self, figure, update_tools=True):
(8)         """
(8)         Bind the given figure to the tools.
(8)         Parameters
(8)         ----------
(8)         figure : `.Figure`
(8)         update_tools : bool, default: True
(12)            Force tools to update figure.
(8)         """
(8)         if self._key_press_handler_id:
(12)            self.canvas.mpl_disconnect(self._key_press_handler_id)
(8)         self._figure = figure
(8)         if figure:
(12)            self._key_press_handler_id = self.canvas.mpl_connect(
(16)                'key_press_event', self._key_press)
(8)         if update_tools:
(12)            for tool in self._tools.values():
(16)                tool.figure = figure
(4)     def toolmanager_connect(self, s, func):
(8)         """
(8)         Connect event with string *s* to *func*.
(8)         Parameters
(8)         ----------
(8)         s : str
(12)            The name of the event. The following events are recognized:
(12)            - 'tool_message_event'
(12)            - 'tool_removed_event'
(12)            - 'tool_added_event'
(12)            For every tool added a new event is created
(12)            - 'tool_trigger_TOOLNAME', where TOOLNAME is the id of the tool.
(8)         func : callable
(12)            Callback function for the toolmanager event with signature::
(16)                def func(event: ToolEvent) -> Any
(8)         Returns
(8)         -------
(8)         cid
(12)            The callback id for the connection. This can be used in
```

```
 95: (12)                        `.toolmanager_disconnect`.
 96: (8)                     """
 97: (8)                     return self._callbacks.connect(s, func)
 98: (4)                 def toolmanager_disconnect(self, cid):
 99: (8)                     """
 100: (8)                    Disconnect callback id *cid*.
 101: (8)                    Example usage::
 102: (12)                       cid = toolmanager.toolmanager_connect('tool_trigger_zoom',
onpress)
 103: (12)                       toolmanager.toolmanager_disconnect(cid)
 104: (8)                    """
 105: (8)                    return self._callbacks.disconnect(cid)
 106: (4)                def message_event(self, message, sender=None):
 107: (8)                    """Emit a `ToolManagerMessageEvent`."""
 108: (8)                    if sender is None:
 109: (12)                       sender = self
 110: (8)                    s = 'tool_message_event'
 111: (8)                    event = ToolManagerMessageEvent(s, sender, message)
 112: (8)                    self._callbacks.process(s, event)
 113: (4)                @property
 114: (4)                def active_toggle(self):
 115: (8)                    """Currently toggled tools."""
 116: (8)                    return self._toggled
 117: (4)                def get_tool_keymap(self, name):
 118: (8)                    """
 119: (8)                    Return the keymap associated with the specified tool.
 120: (8)                    Parameters
 121: (8)                    ----------
 122: (8)                    name : str
 123: (12)                       Name of the Tool.
 124: (8)                    Returns
 125: (8)                    -------
 126: (8)                    list of str
 127: (12)                       List of keys associated with the tool.
 128: (8)                    """
 129: (8)                    keys = [k for k, i in self._keys.items() if i == name]
 130: (8)                    return keys
 131: (4)                def _remove_keys(self, name):
 132: (8)                    for k in self.get_tool_keymap(name):
 133: (12)                       del self._keys[k]
 134: (4)                def update_keymap(self, name, key):
 135: (8)                    """
 136: (8)                    Set the keymap to associate with the specified tool.
 137: (8)                    Parameters
 138: (8)                    ----------
 139: (8)                    name : str
 140: (12)                       Name of the Tool.
 141: (8)                    key : str or list of str
 142: (12)                       Keys to associate with the tool.
 143: (8)                    """
 144: (8)                    if name not in self._tools:
 145: (12)                       raise KeyError(f'{name!r} not in Tools')
 146: (8)                    self._remove_keys(name)
 147: (8)                    if isinstance(key, str):
 148: (12)                       key = [key]
 149: (8)                    for k in key:
 150: (12)                       if k in self._keys:
 151: (16)                           _api.warn_external(
 152: (20)                               f'Key {k} changed from {self._keys[k]} to {name}')
 153: (12)                       self._keys[k] = name
 154: (4)                def remove_tool(self, name):
 155: (8)                    """
 156: (8)                    Remove tool named *name*.
 157: (8)                    Parameters
 158: (8)                    ----------
 159: (8)                    name : str
 160: (12)                       Name of the tool.
 161: (8)                    """
 162: (8)                    tool = self.get_tool(name)
```

```
163: (8)                    if getattr(tool, 'toggled', False):  # If it's a toggled toggle tool,
untoggle
164: (12)                       self.trigger_tool(tool, 'toolmanager')
165: (8)                    self._remove_keys(name)
166: (8)                    event = ToolEvent('tool_removed_event', self, tool)
167: (8)                    self._callbacks.process(event.name, event)
168: (8)                    del self._tools[name]
169: (4)                def add_tool(self, name, tool, *args, **kwargs):
170: (8)                    """
171: (8)                    Add *tool* to `ToolManager`.
172: (8)                    If successful, adds a new event ``tool_trigger_{name}`` where
173: (8)                    ``{name}`` is the *name* of the tool; the event is fired every time
the
174: (8)                    tool is triggered.
175: (8)                    Parameters
176: (8)                    ----------
177: (8)                    name : str
178: (12)                       Name of the tool, treated as the ID, has to be unique.
179: (8)                    tool : type
180: (12)                       Class of the tool to be added.  A subclass will be used
181: (12)                       instead if one was registered for the current canvas class.
182: (8)                    *args, **kwargs
183: (12)                       Passed to the *tool*'s constructor.
184: (8)                    See Also
185: (8)                    --------
186: (8)                    matplotlib.backend_tools.ToolBase : The base class for tools.
187: (8)                    """
188: (8)                    tool_cls = backend_tools._find_tool_class(type(self.canvas), tool)
189: (8)                    if not tool_cls:
190: (12)                       raise ValueError('Impossible to find class for %s' % str(tool))
191: (8)                    if name in self._tools:
192: (12)                       _api.warn_external('A "Tool class" with the same name already '
193: (31)                                          'exists, not added')
194: (12)                       return self._tools[name]
195: (8)                    tool_obj = tool_cls(self, name, *args, **kwargs)
196: (8)                    self._tools[name] = tool_obj
197: (8)                    if tool_obj.default_keymap is not None:
198: (12)                       self.update_keymap(name, tool_obj.default_keymap)
199: (8)                    if isinstance(tool_obj, backend_tools.ToolToggleBase):
200: (12)                       if tool_obj.radio_group is None:
201: (16)                           self._toggled.setdefault(None, set())
202: (12)                       else:
203: (16)                           self._toggled.setdefault(tool_obj.radio_group, None)
204: (12)                       if tool_obj.toggled:
205: (16)                           self._handle_toggle(tool_obj, None, None)
206: (8)                    tool_obj.set_figure(self.figure)
207: (8)                    event = ToolEvent('tool_added_event', self, tool_obj)
208: (8)                    self._callbacks.process(event.name, event)
209: (8)                    return tool_obj
210: (4)                def _handle_toggle(self, tool, canvasevent, data):
211: (8)                    """
212: (8)                    Toggle tools, need to untoggle prior to using other Toggle tool.
213: (8)                    Called from trigger_tool.
214: (8)                    Parameters
215: (8)                    ----------
216: (8)                    tool : `.ToolBase`
217: (8)                    canvasevent : Event
218: (12)                       Original Canvas event or None.
219: (8)                    data : object
220: (12)                       Extra data to pass to the tool when triggering.
221: (8)                    """
222: (8)                    radio_group = tool.radio_group
223: (8)                    if radio_group is None:
224: (12)                       if tool.name in self._toggled[None]:
225: (16)                           self._toggled[None].remove(tool.name)
226: (12)                       else:
227: (16)                           self._toggled[None].add(tool.name)
228: (12)                       return
229: (8)                    if self._toggled[radio_group] == tool.name:
```

```
(12)                        toggled = None
(8)                     elif self._toggled[radio_group] is None:
(12)                        toggled = tool.name
(8)                     else:
(12)                        self.trigger_tool(self._toggled[radio_group],
(30)                                          self,
(30)                                          canvasevent,
(30)                                          data)
(12)                        toggled = tool.name
(8)                     self._toggled[radio_group] = toggled
(4)                 def trigger_tool(self, name, sender=None, canvasevent=None, data=None):
(8)                     """
(8)                     Trigger a tool and emit the ``tool_trigger_{name}`` event.
(8)                     Parameters
(8)                     ----------
(8)                     name : str
(12)                        Name of the tool.
(8)                     sender : object
(12)                        Object that wishes to trigger the tool.
(8)                     canvasevent : Event
(12)                        Original Canvas event or None.
(8)                     data : object
(12)                        Extra data to pass to the tool when triggering.
(8)                     """
(8)                     tool = self.get_tool(name)
(8)                     if tool is None:
(12)                        return
(8)                     if sender is None:
(12)                        sender = self
(8)                     if isinstance(tool, backend_tools.ToolToggleBase):
(12)                        self._handle_toggle(tool, canvasevent, data)
(8)                     tool.trigger(sender, canvasevent, data)  # Actually trigger Tool.
(8)                     s = 'tool_trigger_%s' % name
(8)                     event = ToolTriggerEvent(s, sender, tool, canvasevent, data)
(8)                     self._callbacks.process(s, event)
(4)                 def _key_press(self, event):
(8)                     if event.key is None or self.keypresslock.locked():
(12)                        return
(8)                     name = self._keys.get(event.key, None)
(8)                     if name is None:
(12)                        return
(8)                     self.trigger_tool(name, canvasevent=event)
(4)                 @property
(4)                 def tools(self):
(8)                     """A dict mapping tool name -> controlled tool."""
(8)                     return self._tools
(4)                 def get_tool(self, name, warn=True):
(8)                     """
(8)                     Return the tool object with the given name.
(8)                     For convenience, this passes tool objects through.
(8)                     Parameters
(8)                     ----------
(8)                     name : str or `.ToolBase`
(12)                        Name of the tool, or the tool itself.
(8)                     warn : bool, default: True
(12)                        Whether a warning should be emitted it no tool with the given name
(12)                        exists.
(8)                     Returns
(8)                     -------
(8)                     `.ToolBase` or None
(12)                        The tool or None if no tool with the given name exists.
(8)                     """
(8)                     if (isinstance(name, backend_tools.ToolBase)
(16)                            and name.name in self._tools):
(12)                        return name
(8)                     if name not in self._tools:
(12)                        if warn:
(16)                            _api.warn_external(
(20)                                f"ToolManager does not control tool {name!r}")
```

```
299: (12)                        return None
300: (8)                    return self._tools[name]

----------------------------------------

File 26 - backend_tools.py:

1: (0)              """
2: (0)              Abstract base classes define the primitives for Tools.
3: (0)              These tools are used by `matplotlib.backend_managers.ToolManager`
4: (0)              :class:`ToolBase`
5: (4)                  Simple stateless tool
6: (0)              :class:`ToolToggleBase`
7: (4)                  Tool that has two states, only one Toggle tool can be
8: (4)                  active at any given time for the same
9: (4)                  `matplotlib.backend_managers.ToolManager`
10: (0)             """
11: (0)             import enum
12: (0)             import functools
13: (0)             import re
14: (0)             import time
15: (0)             from types import SimpleNamespace
16: (0)             import uuid
17: (0)             from weakref import WeakKeyDictionary
18: (0)             import numpy as np
19: (0)             import matplotlib as mpl
20: (0)             from matplotlib._pylab_helpers import Gcf
21: (0)             from matplotlib import _api, cbook
22: (0)             class Cursors(enum.IntEnum):  # Must subclass int for the macOS backend.
23: (4)                 """Backend-independent cursor types."""
24: (4)                 POINTER = enum.auto()
25: (4)                 HAND = enum.auto()
26: (4)                 SELECT_REGION = enum.auto()
27: (4)                 MOVE = enum.auto()
28: (4)                 WAIT = enum.auto()
29: (4)                 RESIZE_HORIZONTAL = enum.auto()
30: (4)                 RESIZE_VERTICAL = enum.auto()
31: (0)             cursors = Cursors  # Backcompat.
32: (0)             _tool_registry = set()
33: (0)             def _register_tool_class(canvas_cls, tool_cls=None):
34: (4)                 """Decorator registering *tool_cls* as a tool class for *canvas_cls*."""
35: (4)                 if tool_cls is None:
36: (8)                     return functools.partial(_register_tool_class, canvas_cls)
37: (4)                 _tool_registry.add((canvas_cls, tool_cls))
38: (4)                 return tool_cls
39: (0)             def _find_tool_class(canvas_cls, tool_cls):
40: (4)                 """Find a subclass of *tool_cls* registered for *canvas_cls*."""
41: (4)                 for canvas_parent in canvas_cls.__mro__:
42: (8)                     for tool_child in _api.recursive_subclasses(tool_cls):
43: (12)                        if (canvas_parent, tool_child) in _tool_registry:
44: (16)                            return tool_child
45: (4)                 return tool_cls
46: (0)             _views_positions = 'viewpos'
47: (0)             class ToolBase:
48: (4)                 """
49: (4)                 Base tool class.
50: (4)                 A base tool, only implements `trigger` method or no method at all.
51: (4)                 The tool is instantiated by `matplotlib.backend_managers.ToolManager`.
52: (4)                 """
53: (4)                 default_keymap = None
54: (4)                 """
55: (4)                 Keymap to associate with this tool.
56: (4)                 ``list[str]``: List of keys that will trigger this tool when a keypress
57: (4)                 event is emitted on ``self.figure.canvas``.  Note that this attribute is
58: (4)                 looked up on the instance, and can therefore be a property (this is used
59: (4)                 e.g. by the built-in tools to load the rcParams at instantiation time).
60: (4)                 """
61: (4)                 description = None
62: (4)                 """
```

```
63: (4)                 Description of the Tool.
64: (4)                 `str`: Tooltip used if the Tool is included in a Toolbar.
65: (4)                 """
66: (4)                 image = None
67: (4)                 """
68: (4)                 Icon filename.
69: (4)                 ``str | None``: Filename of the Toolbar icon; either absolute, or relative
to the
70: (4)                 directory containing the Python source file where the ``Tool.image`` class
attribute
71: (4)                 is defined (in the latter case, this cannot be defined as an instance
attribute).
72: (4)                 In either case, the extension is optional; leaving it off lets individual
backends
73: (4)                 select the icon format they prefer.  If None, the *name* is used as a
label in the
74: (4)                 toolbar button.
75: (4)                 """
76: (4)                 def __init__(self, toolmanager, name):
77: (8)                     self._name = name
78: (8)                     self._toolmanager = toolmanager
79: (8)                     self._figure = None
80: (4)                 name = property(
81: (8)                     lambda self: self._name,
82: (8)                     doc="The tool id (str, must be unique among tools of a tool
manager).")
83: (4)                 toolmanager = property(
84: (8)                     lambda self: self._toolmanager,
85: (8)                     doc="The `.ToolManager` that controls this tool.")
86: (4)                 canvas = property(
87: (8)                     lambda self: self._figure.canvas if self._figure is not None else
None,
88: (8)                     doc="The canvas of the figure affected by this tool, or None.")
89: (4)                 def set_figure(self, figure):
90: (8)                     self._figure = figure
91: (4)                 figure = property(
92: (8)                     lambda self: self._figure,
93: (8)                     lambda self, figure: self.set_figure(figure),
94: (8)                     doc="The Figure affected by this tool, or None.")
95: (4)                 def _make_classic_style_pseudo_toolbar(self):
96: (8)                     """
97: (8)                     Return a placeholder object with a single `canvas` attribute.
98: (8)                     This is useful to reuse the implementations of tools already provided
99: (8)                     by the classic Toolbars.
100: (8)                    """
101: (8)                    return SimpleNamespace(canvas=self.canvas)
102: (4)                trigger(self, sender, event, data=None):
103: (8)                    """
104: (8)                    Called when this tool gets used.
105: (8)                    This method is called by `.ToolManager.trigger_tool`.
106: (8)                    Parameters
107: (8)                    ----------
108: (8)                    event : `.Event`
109: (12)                       The canvas event that caused this tool to be called.
110: (8)                    sender : object
111: (12)                       Object that requested the tool to be triggered.
112: (8)                    data : object
113: (12)                       Extra data.
114: (8)                    """
115: (8)                    pass
116: (0)            class ToolToggleBase(ToolBase):
117: (4)                """
118: (4)                Toggleable tool.
119: (4)                Every time it is triggered, it switches between enable and disable.
120: (4)                Parameters
121: (4)                ----------
122: (4)                ``*args``
123: (8)                    Variable length argument to be used by the Tool.
124: (4)                ``**kwargs``
```

```
(8)                 `toggled` if present and True, sets the initial state of the Tool
(8)                 Arbitrary keyword arguments to be consumed by the Tool
(4)             """
(4)             radio_group = None
(4)             """
(4)             Attribute to group 'radio' like tools (mutually exclusive).
(4)             `str` that identifies the group or **None** if not belonging to a group.
(4)             """
(4)             cursor = None
(4)             """Cursor to use when the tool is active."""
(4)             default_toggled = False
(4)             """Default of toggled state."""
(4)             def __init__(self, *args, **kwargs):
(8)                 self._toggled = kwargs.pop('toggled', self.default_toggled)
(8)                 super().__init__(*args, **kwargs)
(4)             def trigger(self, sender, event, data=None):
(8)                 """Calls `enable` or `disable` based on `toggled` value."""
(8)                 if self._toggled:
(12)                    self.disable(event)
(8)                 else:
(12)                    self.enable(event)
(8)                 self._toggled = not self._toggled
(4)             def enable(self, event=None):
(8)                 """
(8)                 Enable the toggle tool.
(8)                 `trigger` calls this method when `toggled` is False.
(8)                 """
(8)                 pass
(4)             def disable(self, event=None):
(8)                 """
(8)                 Disable the toggle tool.
(8)                 `trigger` call this method when `toggled` is True.
(8)                 This can happen in different circumstances.
(8)                 * Click on the toolbar tool button.
(8)                 * Call to `matplotlib.backend_managers.ToolManager.trigger_tool`.
(8)                 * Another `ToolToggleBase` derived tool is triggered
(10)                  (from the same `.ToolManager`).
(8)                 """
(8)                 pass
(4)             @property
(4)             def toggled(self):
(8)                 """State of the toggled tool."""
(8)                 return self._toggled
(4)             def set_figure(self, figure):
(8)                 toggled = self.toggled
(8)                 if toggled:
(12)                    if self.figure:
(16)                        self.trigger(self, None)
(12)                    else:
(16)                        self._toggled = False
(8)                 super().set_figure(figure)
(8)                 if toggled:
(12)                    if figure:
(16)                        self.trigger(self, None)
(12)                    else:
(16)                        self._toggled = True
(0)         class ToolSetCursor(ToolBase):
(4)             """
(4)             Change to the current cursor while inaxes.
(4)             This tool, keeps track of all `ToolToggleBase` derived tools, and updates
(4)             the cursor when a tool gets triggered.
(4)             """
(4)             def __init__(self, *args, **kwargs):
(8)                 super().__init__(*args, **kwargs)
(8)                 self._id_drag = None
(8)                 self._current_tool = None
(8)                 self._default_cursor = cursors.POINTER
(8)                 self._last_cursor = self._default_cursor
(8)                 self.toolmanager.toolmanager_connect('tool_added_event',
```

```
194: (45)                                                       self._add_tool_cbk)
195: (8)                    for tool in self.toolmanager.tools.values():
196: (12)                       self._add_tool(tool)
197: (4)                def set_figure(self, figure):
198: (8)                    if self._id_drag:
199: (12)                       self.canvas.mpl_disconnect(self._id_drag)
200: (8)                    super().set_figure(figure)
201: (8)                    if figure:
202: (12)                       self._id_drag = self.canvas.mpl_connect(
203: (16)                           'motion_notify_event', self._set_cursor_cbk)
204: (4)                def _tool_trigger_cbk(self, event):
205: (8)                    if event.tool.toggled:
206: (12)                       self._current_tool = event.tool
207: (8)                    else:
208: (12)                       self._current_tool = None
209: (8)                    self._set_cursor_cbk(event.canvasevent)
210: (4)                def _add_tool(self, tool):
211: (8)                    """Set the cursor when the tool is triggered."""
212: (8)                    if getattr(tool, 'cursor', None) is not None:
213: (12)                       self.toolmanager.toolmanager_connect('tool_trigger_%s' %
tool.name,
214: (49)                                                               self._tool_trigger_cbk)
215: (4)                def _add_tool_cbk(self, event):
216: (8)                    """Process every newly added tool."""
217: (8)                    if event.tool is self:
218: (12)                       return
219: (8)                    self._add_tool(event.tool)
220: (4)                def _set_cursor_cbk(self, event):
221: (8)                    if not event or not self.canvas:
222: (12)                       return
223: (8)                    if (self._current_tool and getattr(event, "inaxes", None)
224: (16)                           and event.inaxes.get_navigate()):
225: (12)                       if self._last_cursor != self._current_tool.cursor:
226: (16)                           self.canvas.set_cursor(self._current_tool.cursor)
227: (16)                           self._last_cursor = self._current_tool.cursor
228: (8)                    elif self._last_cursor != self._default_cursor:
229: (12)                       self.canvas.set_cursor(self._default_cursor)
230: (12)                       self._last_cursor = self._default_cursor
231: (0)            class ToolCursorPosition(ToolBase):
232: (4)                """
233: (4)                Send message with the current pointer position.
234: (4)                This tool runs in the background reporting the position of the cursor.
235: (4)                """
236: (4)                def __init__(self, *args, **kwargs):
237: (8)                    self._id_drag = None
238: (8)                    super().__init__(*args, **kwargs)
239: (4)                def set_figure(self, figure):
240: (8)                    if self._id_drag:
241: (12)                       self.canvas.mpl_disconnect(self._id_drag)
242: (8)                    super().set_figure(figure)
243: (8)                    if figure:
244: (12)                       self._id_drag = self.canvas.mpl_connect(
245: (16)                           'motion_notify_event', self.send_message)
246: (4)                def send_message(self, event):
247: (8)                    """Call `matplotlib.backend_managers.ToolManager.message_event`."""
248: (8)                    if self.toolmanager.messagelock.locked():
249: (12)                       return
250: (8)                    from matplotlib.backend_bases import NavigationToolbar2
251: (8)                    message = NavigationToolbar2._mouse_event_to_message(event)
252: (8)                    self.toolmanager.message_event(message, self)
253: (0)            class RubberbandBase(ToolBase):
254: (4)                """Draw and remove a rubberband."""
255: (4)                def trigger(self, sender, event, data=None):
256: (8)                    """Call `draw_rubberband` or `remove_rubberband` based on data."""
257: (8)                    if not self.figure.canvas.widgetlock.available(sender):
258: (12)                       return
259: (8)                    if data is not None:
260: (12)                       self.draw_rubberband(*data)
261: (8)                    else:
```

```
(12)                        self.remove_rubberband()
(4)                     def draw_rubberband(self, *data):
(8)                         """
(8)                         Draw rubberband.
(8)                         This method must get implemented per backend.
(8)                         """
(8)                         raise NotImplementedError
(4)                     def remove_rubberband(self):
(8)                         """
(8)                         Remove rubberband.
(8)                         This method should get implemented per backend.
(8)                         """
(8)                         pass
(0)                 class ToolQuit(ToolBase):
(4)                     """Tool to call the figure manager destroy method."""
(4)                     description = 'Quit the figure'
(4)                     default_keymap = property(lambda self: mpl.rcParams['keymap.quit'])
(4)                     def trigger(self, sender, event, data=None):
(8)                         Gcf.destroy_fig(self.figure)
(0)                 class ToolQuitAll(ToolBase):
(4)                     """Tool to call the figure manager destroy method."""
(4)                     description = 'Quit all figures'
(4)                     default_keymap = property(lambda self: mpl.rcParams['keymap.quit_all'])
(4)                     def trigger(self, sender, event, data=None):
(8)                         Gcf.destroy_all()
(0)                 class ToolGrid(ToolBase):
(4)                     """Tool to toggle the major grids of the figure."""
(4)                     description = 'Toggle major grids'
(4)                     default_keymap = property(lambda self: mpl.rcParams['keymap.grid'])
(4)                     def trigger(self, sender, event, data=None):
(8)                         sentinel = str(uuid.uuid4())
(8)                         with cbook._setattr_cm(event, key=sentinel), \
(13)                             mpl.rc_context({'keymap.grid': sentinel}):
(12)                            mpl.backend_bases.key_press_handler(event, self.figure.canvas)
(0)                 class ToolMinorGrid(ToolBase):
(4)                     """Tool to toggle the major and minor grids of the figure."""
(4)                     description = 'Toggle major and minor grids'
(4)                     default_keymap = property(lambda self: mpl.rcParams['keymap.grid_minor'])
(4)                     def trigger(self, sender, event, data=None):
(8)                         sentinel = str(uuid.uuid4())
(8)                         with cbook._setattr_cm(event, key=sentinel), \
(13)                             mpl.rc_context({'keymap.grid_minor': sentinel}):
(12)                            mpl.backend_bases.key_press_handler(event, self.figure.canvas)
(0)                 class ToolFullScreen(ToolBase):
(4)                     """Tool to toggle full screen."""
(4)                     description = 'Toggle fullscreen mode'
(4)                     default_keymap = property(lambda self: mpl.rcParams['keymap.fullscreen'])
(4)                     def trigger(self, sender, event, data=None):
(8)                         self.figure.canvas.manager.full_screen_toggle()
(0)                 class AxisScaleBase(ToolToggleBase):
(4)                     """Base Tool to toggle between linear and logarithmic."""
(4)                     def trigger(self, sender, event, data=None):
(8)                         if event.inaxes is None:
(12)                            return
(8)                         super().trigger(sender, event, data)
(4)                     def enable(self, event=None):
(8)                         self.set_scale(event.inaxes, 'log')
(8)                         self.figure.canvas.draw_idle()
(4)                     def disable(self, event=None):
(8)                         self.set_scale(event.inaxes, 'linear')
(8)                         self.figure.canvas.draw_idle()
(0)                 class ToolYScale(AxisScaleBase):
(4)                     """Tool to toggle between linear and logarithmic scales on the Y axis."""
(4)                     description = 'Toggle scale Y axis'
(4)                     default_keymap = property(lambda self: mpl.rcParams['keymap.yscale'])
(4)                     def set_scale(self, ax, scale):
(8)                         ax.set_yscale(scale)
(0)                 class ToolXScale(AxisScaleBase):
(4)                     """Tool to toggle between linear and logarithmic scales on the X axis."""
```

```
(4)                 description = 'Toggle scale X axis'
(4)                 default_keymap = property(lambda self: mpl.rcParams['keymap.xscale'])
(4)                 def set_scale(self, ax, scale):
(8)                     ax.set_xscale(scale)
(0)             class ToolViewsPositions(ToolBase):
(4)                 """
(4)                 Auxiliary Tool to handle changes in views and positions.
(4)                 Runs in the background and should get used by all the tools that
(4)                 need to access the figure's history of views and positions, e.g.
(4)                 * `ToolZoom`
(4)                 * `ToolPan`
(4)                 * `ToolHome`
(4)                 * `ToolBack`
(4)                 * `ToolForward`
(4)                 """
(4)                 def __init__(self, *args, **kwargs):
(8)                     self.views = WeakKeyDictionary()
(8)                     self.positions = WeakKeyDictionary()
(8)                     self.home_views = WeakKeyDictionary()
(8)                     super().__init__(*args, **kwargs)
(4)                 def add_figure(self, figure):
(8)                     """Add the current figure to the stack of views and positions."""
(8)                     if figure not in self.views:
(12)                        self.views[figure] = cbook._Stack()
(12)                        self.positions[figure] = cbook._Stack()
(12)                        self.home_views[figure] = WeakKeyDictionary()
(12)                        self.push_current(figure)
(12)                        figure.add_axobserver(lambda fig: self.update_home_views(fig))
(4)                 def clear(self, figure):
(8)                     """Reset the Axes stack."""
(8)                     if figure in self.views:
(12)                        self.views[figure].clear()
(12)                        self.positions[figure].clear()
(12)                        self.home_views[figure].clear()
(12)                        self.update_home_views()
(4)                 def update_view(self):
(8)                     """
(8)                     Update the view limits and position for each Axes from the current
(8)                     stack position. If any Axes are present in the figure that aren't in
(8)                     the current stack position, use the home view limits for those Axes
and
(8)                     don't update *any* positions.
(8)                     """
(8)                     views = self.views[self.figure]()
(8)                     if views is None:
(12)                        return
(8)                     pos = self.positions[self.figure]()
(8)                     if pos is None:
(12)                        return
(8)                     home_views = self.home_views[self.figure]
(8)                     all_axes = self.figure.get_axes()
(8)                     for a in all_axes:
(12)                        if a in views:
(16)                            cur_view = views[a]
(12)                        else:
(16)                            cur_view = home_views[a]
(12)                        a._set_view(cur_view)
(8)                     if set(all_axes).issubset(pos):
(12)                        for a in all_axes:
(16)                            a._set_position(pos[a][0], 'original')
(16)                            a._set_position(pos[a][1], 'active')
(8)                     self.figure.canvas.draw_idle()
(4)                 def push_current(self, figure=None):
(8)                     """
(8)                     Push the current view limits and position onto their respective
stacks.
(8)                     """
(8)                     if not figure:
(12)                        figure = self.figure
```

```
(8)                 views = WeakKeyDictionary()
(8)                 pos = WeakKeyDictionary()
(8)                 for a in figure.get_axes():
(12)                    views[a] = a._get_view()
(12)                    pos[a] = self._axes_pos(a)
(8)                 self.views[figure].push(views)
(8)                 self.positions[figure].push(pos)
(4)             def _axes_pos(self, ax):
(8)                 """
(8)                 Return the original and modified positions for the specified Axes.
(8)                 Parameters
(8)                 ----------
(8)                 ax : matplotlib.axes.Axes
(12)                    The `.Axes` to get the positions for.
(8)                 Returns
(8)                 -------
(8)                 original_position, modified_position
(12)                    A tuple of the original and modified positions.
(8)                 """
(8)                 return (ax.get_position(True).frozen(),
(16)                        ax.get_position().frozen())
(4)             def update_home_views(self, figure=None):
(8)                 """
(8)                 Make sure that ``self.home_views`` has an entry for all Axes present
(8)                 in the figure.
(8)                 """
(8)                 if not figure:
(12)                    figure = self.figure
(8)                 for a in figure.get_axes():
(12)                    if a not in self.home_views[figure]:
(16)                        self.home_views[figure][a] = a._get_view()
(4)             def home(self):
(8)                 """Recall the first view and position from the stack."""
(8)                 self.views[self.figure].home()
(8)                 self.positions[self.figure].home()
(4)             def back(self):
(8)                 """Back one step in the stack of views and positions."""
(8)                 self.views[self.figure].back()
(8)                 self.positions[self.figure].back()
(4)             def forward(self):
(8)                 """Forward one step in the stack of views and positions."""
(8)                 self.views[self.figure].forward()
(8)                 self.positions[self.figure].forward()
(0)         class ViewsPositionsBase(ToolBase):
(4)             """Base class for `ToolHome`, `ToolBack` and `ToolForward`."""
(4)             _on_trigger = None
(4)             def trigger(self, sender, event, data=None):
(8)                 self.toolmanager.get_tool(_views_positions).add_figure(self.figure)
(8)                 getattr(self.toolmanager.get_tool(_views_positions),
(16)                        self._on_trigger)()
(8)                 self.toolmanager.get_tool(_views_positions).update_view()
(0)         class ToolHome(ViewsPositionsBase):
(4)             """Restore the original view limits."""
(4)             description = 'Reset original view'
(4)             image = 'mpl-data/images/home'
(4)             default_keymap = property(lambda self: mpl.rcParams['keymap.home'])
(4)             _on_trigger = 'home'
(0)         class ToolBack(ViewsPositionsBase):
(4)             """Move back up the view limits stack."""
(4)             description = 'Back to previous view'
(4)             image = 'mpl-data/images/back'
(4)             default_keymap = property(lambda self: mpl.rcParams['keymap.back'])
(4)             _on_trigger = 'back'
(0)         class ToolForward(ViewsPositionsBase):
(4)             """Move forward in the view lim stack."""
(4)             description = 'Forward to next view'
(4)             image = 'mpl-data/images/forward'
(4)             default_keymap = property(lambda self: mpl.rcParams['keymap.forward'])
(4)             _on_trigger = 'forward'
```

```
(0)  class ConfigureSubplotsBase(ToolBase):
(4)      """Base tool for the configuration of subplots."""
(4)      description = 'Configure subplots'
(4)      image = 'mpl-data/images/subplots'
(0)  class SaveFigureBase(ToolBase):
(4)      """Base tool for figure saving."""
(4)      description = 'Save the figure'
(4)      image = 'mpl-data/images/filesave'
(4)      default_keymap = property(lambda self: mpl.rcParams['keymap.save'])
(0)  class ZoomPanBase(ToolToggleBase):
(4)      """Base class for `ToolZoom` and `ToolPan`."""
(4)      def __init__(self, *args):
(8)          super().__init__(*args)
(8)          self._button_pressed = None
(8)          self._xypress = None
(8)          self._idPress = None
(8)          self._idRelease = None
(8)          self._idScroll = None
(8)          self.base_scale = 2.
(8)          self.scrollthresh = .5  # .5 second scroll threshold
(8)          self.lastscroll = time.time()-self.scrollthresh
(4)      def enable(self, event=None):
(8)          """Connect press/release events and lock the canvas."""
(8)          self.figure.canvas.widgetlock(self)
(8)          self._idPress = self.figure.canvas.mpl_connect(
(12)             'button_press_event', self._press)
(8)          self._idRelease = self.figure.canvas.mpl_connect(
(12)             'button_release_event', self._release)
(8)          self._idScroll = self.figure.canvas.mpl_connect(
(12)             'scroll_event', self.scroll_zoom)
(4)      def disable(self, event=None):
(8)          """Release the canvas and disconnect press/release events."""
(8)          self._cancel_action()
(8)          self.figure.canvas.widgetlock.release(self)
(8)          self.figure.canvas.mpl_disconnect(self._idPress)
(8)          self.figure.canvas.mpl_disconnect(self._idRelease)
(8)          self.figure.canvas.mpl_disconnect(self._idScroll)
(4)      def trigger(self, sender, event, data=None):
(8)          self.toolmanager.get_tool(_views_positions).add_figure(self.figure)
(8)          super().trigger(sender, event, data)
(8)          new_navigate_mode = self.name.upper() if self.toggled else None
(8)          for ax in self.figure.axes:
(12)             ax.set_navigate_mode(new_navigate_mode)
(4)      def scroll_zoom(self, event):
(8)          if event.inaxes is None:
(12)             return
(8)          if event.button == 'up':
(12)             scl = self.base_scale
(8)          elif event.button == 'down':
(12)             scl = 1/self.base_scale
(8)          else:
(12)             scl = 1
(8)          ax = event.inaxes
(8)          ax._set_view_from_bbox([event.x, event.y, scl])
(8)          if (time.time()-self.lastscroll) < self.scrollthresh:
(12)             self.toolmanager.get_tool(_views_positions).back()
(8)          self.figure.canvas.draw_idle()  # force re-draw
(8)          self.lastscroll = time.time()
(8)          self.toolmanager.get_tool(_views_positions).push_current()
(0)  class ToolZoom(ZoomPanBase):
(4)      """A Tool for zooming using a rectangle selector."""
(4)      description = 'Zoom to rectangle'
(4)      image = 'mpl-data/images/zoom_to_rect'
(4)      default_keymap = property(lambda self: mpl.rcParams['keymap.zoom'])
(4)      cursor = cursors.SELECT_REGION
(4)      radio_group = 'default'
(4)      def __init__(self, *args):
(8)          super().__init__(*args)
(8)          self._ids_zoom = []
```

```
(4)     def _cancel_action(self):
(8)         for zoom_id in self._ids_zoom:
(12)            self.figure.canvas.mpl_disconnect(zoom_id)
(8)         self.toolmanager.trigger_tool('rubberband', self)
(8)         self.figure.canvas.draw_idle()
(8)         self._xypress = None
(8)         self._button_pressed = None
(8)         self._ids_zoom = []
(8)         return
(4)     def _press(self, event):
(8)         """Callback for mouse button presses in zoom-to-rectangle mode."""
(8)         if self._ids_zoom:
(12)            self._cancel_action()
(8)         if event.button == 1:
(12)            self._button_pressed = 1
(8)         elif event.button == 3:
(12)            self._button_pressed = 3
(8)         else:
(12)            self._cancel_action()
(12)            return
(8)         x, y = event.x, event.y
(8)         self._xypress = []
(8)         for i, a in enumerate(self.figure.get_axes()):
(12)            if (x is not None and y is not None and a.in_axes(event) and
(20)                    a.get_navigate() and a.can_zoom()):
(16)                self._xypress.append((x, y, a, i, a._get_view()))
(8)         id1 = self.figure.canvas.mpl_connect(
(12)            'motion_notify_event', self._mouse_move)
(8)         id2 = self.figure.canvas.mpl_connect(
(12)            'key_press_event', self._switch_on_zoom_mode)
(8)         id3 = self.figure.canvas.mpl_connect(
(12)            'key_release_event', self._switch_off_zoom_mode)
(8)         self._ids_zoom = id1, id2, id3
(8)         self._zoom_mode = event.key
(4)     def _switch_on_zoom_mode(self, event):
(8)         self._zoom_mode = event.key
(8)         self._mouse_move(event)
(4)     def _switch_off_zoom_mode(self, event):
(8)         self._zoom_mode = None
(8)         self._mouse_move(event)
(4)     def _mouse_move(self, event):
(8)         """Callback for mouse moves in zoom-to-rectangle mode."""
(8)         if self._xypress:
(12)            x, y = event.x, event.y
(12)            lastx, lasty, a, ind, view = self._xypress[0]
(12)            (x1, y1), (x2, y2) = np.clip(
(16)                [[lastx, lasty], [x, y]], a.bbox.min, a.bbox.max)
(12)            if self._zoom_mode == "x":
(16)                y1, y2 = a.bbox.intervaly
(12)            elif self._zoom_mode == "y":
(16)                x1, x2 = a.bbox.intervalx
(12)            self.toolmanager.trigger_tool(
(16)                'rubberband', self, data=(x1, y1, x2, y2))
(4)     def _release(self, event):
(8)         """Callback for mouse button releases in zoom-to-rectangle mode."""
(8)         for zoom_id in self._ids_zoom:
(12)            self.figure.canvas.mpl_disconnect(zoom_id)
(8)         self._ids_zoom = []
(8)         if not self._xypress:
(12)            self._cancel_action()
(12)            return
(8)         done_ax = []
(8)         for cur_xypress in self._xypress:
(12)            x, y = event.x, event.y
(12)            lastx, lasty, a, _ind, view = cur_xypress
(12)            if abs(x - lastx) < 5 or abs(y - lasty) < 5:
(16)                self._cancel_action()
(16)                return
(12)            twinx = any(a.get_shared_x_axes().joined(a, a1) for a1 in done_ax)
```

```
(12)                        twiny = any(a.get_shared_y_axes().joined(a, a1) for a1 in done_ax)
(12)                        done_ax.append(a)
(12)                        if self._button_pressed == 1:
(16)                            direction = 'in'
(12)                        elif self._button_pressed == 3:
(16)                            direction = 'out'
(12)                        else:
(16)                            continue
(12)                        a._set_view_from_bbox((lastx, lasty, x, y), direction,
(34)                                              self._zoom_mode, twinx, twiny)
(8)                     self._zoom_mode = None
(8)                     self.toolmanager.get_tool(_views_positions).push_current()
(8)                     self._cancel_action()
(0)             class ToolPan(ZoomPanBase):
(4)                 """Pan Axes with left mouse, zoom with right."""
(4)                 default_keymap = property(lambda self: mpl.rcParams['keymap.pan'])
(4)                 description = 'Pan axes with left mouse, zoom with right'
(4)                 image = 'mpl-data/images/move'
(4)                 cursor = cursors.MOVE
(4)                 radio_group = 'default'
(4)                 def __init__(self, *args):
(8)                     super().__init__(*args)
(8)                     self._id_drag = None
(4)                 def _cancel_action(self):
(8)                     self._button_pressed = None
(8)                     self._xypress = []
(8)                     self.figure.canvas.mpl_disconnect(self._id_drag)
(8)                     self.toolmanager.messagelock.release(self)
(8)                     self.figure.canvas.draw_idle()
(4)                 def _press(self, event):
(8)                     if event.button == 1:
(12)                        self._button_pressed = 1
(8)                     elif event.button == 3:
(12)                        self._button_pressed = 3
(8)                     else:
(12)                        self._cancel_action()
(12)                        return
(8)                     x, y = event.x, event.y
(8)                     self._xypress = []
(8)                     for i, a in enumerate(self.figure.get_axes()):
(12)                        if (x is not None and y is not None and a.in_axes(event) and
(20)                                a.get_navigate() and a.can_pan()):
(16)                            a.start_pan(x, y, event.button)
(16)                            self._xypress.append((a, i))
(16)                            self.toolmanager.messagelock(self)
(16)                            self._id_drag = self.figure.canvas.mpl_connect(
(20)                                'motion_notify_event', self._mouse_move)
(4)                 def _release(self, event):
(8)                     if self._button_pressed is None:
(12)                        self._cancel_action()
(12)                        return
(8)                     self.figure.canvas.mpl_disconnect(self._id_drag)
(8)                     self.toolmanager.messagelock.release(self)
(8)                     for a, _ind in self._xypress:
(12)                        a.end_pan()
(8)                     if not self._xypress:
(12)                        self._cancel_action()
(12)                        return
(8)                     self.toolmanager.get_tool(_views_positions).push_current()
(8)                     self._cancel_action()
(4)                 def _mouse_move(self, event):
(8)                     for a, _ind in self._xypress:
(12)                        a.drag_pan(self._button_pressed, event.key, event.x, event.y)
(8)                     self.toolmanager.canvas.draw_idle()
(0)             class ToolHelpBase(ToolBase):
(4)                 description = 'Print tool list, shortcuts and description'
(4)                 default_keymap = property(lambda self: mpl.rcParams['keymap.help'])
(4)                 image = 'mpl-data/images/help'
(4)                 @staticmethod
```

```
(4)     def format_shortcut(key_sequence):
(8)         """
(8)         Convert a shortcut string from the notation used in rc config to the
(8)         standard notation for displaying shortcuts, e.g. 'ctrl+a' -> 'Ctrl+A'.
(8)         """
(8)         return (key_sequence if len(key_sequence) == 1 else
(16)                 re.sub(r"\+[A-Z]", r"+Shift\g<0>", key_sequence).title())
(4)     def _format_tool_keymap(self, name):
(8)         keymaps = self.toolmanager.get_tool_keymap(name)
(8)         return ", ".join(self.format_shortcut(keymap) for keymap in keymaps)
(4)     def _get_help_entries(self):
(8)         return [(name, self._format_tool_keymap(name), tool.description)
(16)                 for name, tool in sorted(self.toolmanager.tools.items())
(16)                 if tool.description]
(4)     def _get_help_text(self):
(8)         entries = self._get_help_entries()
(8)         entries = ["{}: {}\n\t{}".format(*entry) for entry in entries]
(8)         return "\n".join(entries)
(4)     def _get_help_html(self):
(8)         fmt = "<tr><td>{}</td><td>{}</td><td>{}</td></tr>"
(8)         rows = [fmt.format(
(12)            "<b>Action</b>", "<b>Shortcuts</b>", "<b>Description</b>")]
(8)         rows += [fmt.format(*row) for row in self._get_help_entries()]
(8)         return ("<style>td {padding: 0px 4px}</style>"
(16)                "<table><thead>" + rows[0] + "</thead>"
(16)                "<tbody>".join(rows[1:]) + "</tbody></table>")
(0)     class ToolCopyToClipboardBase(ToolBase):
(4)         """Tool to copy the figure to the clipboard."""
(4)         description = 'Copy the canvas figure to clipboard'
(4)         default_keymap = property(lambda self: mpl.rcParams['keymap.copy'])
(4)         def trigger(self, *args, **kwargs):
(8)         message = "Copy tool is not available"
(8)         self.toolmanager.message_event(message, self)
(0)     default_tools = {'home': ToolHome, 'back': ToolBack, 'forward': ToolForward,
(17)                     'zoom': ToolZoom, 'pan': ToolPan,
(17)                     'subplots': ConfigureSubplotsBase,
(17)                     'save': SaveFigureBase,
(17)                     'grid': ToolGrid,
(17)                     'grid_minor': ToolMinorGrid,
(17)                     'fullscreen': ToolFullScreen,
(17)                     'quit': ToolQuit,
(17)                     'quit_all': ToolQuitAll,
(17)                     'xscale': ToolXScale,
(17)                     'yscale': ToolYScale,
(17)                     'position': ToolCursorPosition,
(17)                     _views_positions: ToolViewsPositions,
(17)                     'cursor': ToolSetCursor,
(17)                     'rubberband': RubberbandBase,
(17)                     'help': ToolHelpBase,
(17)                     'copy': ToolCopyToClipboardBase,
(17)                     }
(0)     default_toolbar_tools = [['navigation', ['home', 'back', 'forward']],
(25)                             ['zoompan', ['pan', 'zoom', 'subplots']],
(25)                             ['io', ['save', 'help']]]
(0)     def add_tools_to_manager(toolmanager, tools=default_tools):
(4)         """
(4)         Add multiple tools to a `.ToolManager`.
(4)         Parameters
(4)         ----------
(4)         toolmanager : `.backend_managers.ToolManager`
(8)             Manager to which the tools are added.
(4)         tools : {str: class_like}, optional
(8)             The tools to add in a {name: tool} dict, see
(8)             `.backend_managers.ToolManager.add_tool` for more info.
(4)         """
(4)         for name, tool in tools.items():
(8)             toolmanager.add_tool(name, tool)
(0)     def add_tools_to_container(container, tools=default_toolbar_tools):
(4)         """
```

```
743: (4)                Add multiple tools to the container.
744: (4)                Parameters
745: (4)                ----------
746: (4)                container : Container
747: (8)                    `.backend_bases.ToolContainerBase` object that will get the tools
748: (8)                    added.
749: (4)                tools : list, optional
750: (8)                    List in the form ``[[group1, [tool1, tool2 ...]], [group2, [...]]]``
751: (8)                    where the tools ``[tool1, tool2, ...]`` will display in group1.
752: (8)                    See `.backend_bases.ToolContainerBase.add_tool` for details.
753: (4)                """
754: (4)                for group, grouptools in tools:
755: (8)                    for position, tool in enumerate(grouptools):
756: (12)                       container.add_tool(tool, group, position)
```

File 27 - bezier.py:

```
1: (0)              """
2: (0)              A module providing some utility functions regarding Bézier path manipulation.
3: (0)              """
4: (0)              from functools import lru_cache
5: (0)              import math
6: (0)              import warnings
7: (0)              import numpy as np
8: (0)              from matplotlib import _api
9: (0)              @np.vectorize
10: (0)             @lru_cache(maxsize=128)
11: (0)             def _comb(n, k):
12: (4)                 if k > n:
13: (8)                     return 0
14: (4)                 k = min(k, n - k)
15: (4)                 i = np.arange(1, k + 1)
16: (4)                 return np.prod((n + 1 - i)/i).astype(int)
17: (0)             class NonIntersectingPathException(ValueError):
18: (4)                 pass
19: (0)             def get_intersection(cx1, cy1, cos_t1, sin_t1,
20: (21)                                 cx2, cy2, cos_t2, sin_t2):
21: (4)                 """
22: (4)                 Return the intersection between the line through (*cx1*, *cy1*) at angle
23: (4)                 *t1* and the line through (*cx2*, *cy2*) at angle *t2*.
24: (4)                 """
25: (4)                 line1_rhs = sin_t1 * cx1 - cos_t1 * cy1
26: (4)                 line2_rhs = sin_t2 * cx2 - cos_t2 * cy2
27: (4)                 a, b = sin_t1, -cos_t1
28: (4)                 c, d = sin_t2, -cos_t2
29: (4)                 ad_bc = a * d - b * c
30: (4)                 if abs(ad_bc) < 1e-12:
31: (8)                     raise ValueError("Given lines do not intersect. Please verify that "
32: (25)                                     "the angles are not equal or differ by 180 degrees.")
33: (4)                 a_, b_ = d, -b
34: (4)                 c_, d_ = -c, a
35: (4)                 a_, b_, c_, d_ = [k / ad_bc for k in [a_, b_, c_, d_]]
36: (4)                 x = a_ * line1_rhs + b_ * line2_rhs
37: (4)                 y = c_ * line1_rhs + d_ * line2_rhs
38: (4)                 return x, y
39: (0)             def get_normal_points(cx, cy, cos_t, sin_t, length):
40: (4)                 """
41: (4)                 For a line passing through (*cx*, *cy*) and having an angle *t*, return
42: (4)                 locations of the two points located along its perpendicular line at the
43: (4)                 distance of *length*.
44: (4)                 """
45: (4)                 if length == 0.:
46: (8)                     return cx, cy, cx, cy
47: (4)                 cos_t1, sin_t1 = sin_t, -cos_t
48: (4)                 cos_t2, sin_t2 = -sin_t, cos_t
49: (4)                 x1, y1 = length * cos_t1 + cx, length * sin_t1 + cy
50: (4)                 x2, y2 = length * cos_t2 + cx, length * sin_t2 + cy
```

```
(4)                 return x1, y1, x2, y2
(0)             def _de_casteljau1(beta, t):
(4)                 next_beta = beta[:-1] * (1 - t) + beta[1:] * t
(4)                 return next_beta
(0)             def split_de_casteljau(beta, t):
(4)                 """
(4)                 Split a Bézier segment defined by its control points *beta* into two
(4)                 separate segments divided at *t* and return their control points.
(4)                 """
(4)                 beta = np.asarray(beta)
(4)                 beta_list = [beta]
(4)                 while True:
(8)                     beta = _de_casteljau1(beta, t)
(8)                     beta_list.append(beta)
(8)                     if len(beta) == 1:
(12)                        break
(4)                 left_beta = [beta[0] for beta in beta_list]
(4)                 right_beta = [beta[-1] for beta in reversed(beta_list)]
(4)                 return left_beta, right_beta
(0)             def find_bezier_t_intersecting_with_closedpath(
(8)                     bezier_point_at_t, inside_closedpath, t0=0., t1=1., tolerance=0.01):
(4)                 """
(4)                 Find the intersection of the Bézier curve with a closed path.
(4)                 The intersection point *t* is approximated by two parameters *t0*, *t1*
(4)                 such that *t0* <= *t* <= *t1*.
(4)                 Search starts from *t0* and *t1* and uses a simple bisecting algorithm
(4)                 therefore one of the end points must be inside the path while the other
(4)                 doesn't. The search stops when the distance of the points parametrized by
(4)                 *t0* and *t1* gets smaller than the given *tolerance*.
(4)                 Parameters
(4)                 ----------
(4)                 bezier_point_at_t : callable
(8)                     A function returning x, y coordinates of the Bézier at parameter *t*.
(8)                     It must have the signature::
(12)                        bezier_point_at_t(t: float) -> tuple[float, float]
(4)                 inside_closedpath : callable
(8)                     A function returning True if a given point (x, y) is inside the
(8)                     closed path. It must have the signature::
(12)                        inside_closedpath(point: tuple[float, float]) -> bool
(4)                 t0, t1 : float
(8)                     Start parameters for the search.
(4)                 tolerance : float
(8)                     Maximal allowed distance between the final points.
(4)                 Returns
(4)                 -------
(4)                 t0, t1 : float
(8)                     The Bézier path parameters.
(4)                 """
(4)                 start = bezier_point_at_t(t0)
(4)                 end = bezier_point_at_t(t1)
(4)                 start_inside = inside_closedpath(start)
(4)                 end_inside = inside_closedpath(end)
(4)                 if start_inside == end_inside and start != end:
(8)                     raise NonIntersectingPathException(
(12)                        "Both points are on the same side of the closed path")
(4)                 while True:
(8)                     if np.hypot(start[0] - end[0], start[1] - end[1]) < tolerance:
(12)                        return t0, t1
(8)                     middle_t = 0.5 * (t0 + t1)
(8)                     middle = bezier_point_at_t(middle_t)
(8)                     middle_inside = inside_closedpath(middle)
(8)                     if start_inside ^ middle_inside:
(12)                        t1 = middle_t
(12)                        if end == middle:
(16)                            return t0, t1
(12)                        end = middle
(8)                     else:
(12)                        t0 = middle_t
(12)                        if start == middle:
```

```
(16)                        return t0, t1
(12)                    start = middle
(12)                    start_inside = middle_inside
(0)         class BezierSegment:
(4)             """
(4)             A d-dimensional Bézier segment.
(4)             Parameters
(4)             ----------
(4)             control_points : (N, d) array
(8)                 Location of the *N* control points.
(4)             """
(4)             def __init__(self, control_points):
(8)                 self._cpoints = np.asarray(control_points)
(8)                 self._N, self._d = self._cpoints.shape
(8)                 self._orders = np.arange(self._N)
(8)                 coeff = [math.factorial(self._N - 1)
(17)                         // (math.factorial(i) * math.factorial(self._N - 1 - i))
(17)                         for i in range(self._N)]
(8)                 self._px = (self._cpoints.T * coeff).T
(4)             def __call__(self, t):
(8)                 """
(8)                 Evaluate the Bézier curve at point(s) *t* in [0, 1].
(8)                 Parameters
(8)                 ----------
(8)                 t : (k,) array-like
(12)                    Points at which to evaluate the curve.
(8)                 Returns
(8)                 -------
(8)                 (k, d) array
(12)                    Value of the curve for each point in *t*.
(8)                 """
(8)                 t = np.asarray(t)
(8)                 return (np.power.outer(1 - t, self._orders[::-1])
(16)                        * np.power.outer(t, self._orders)) @ self._px
(4)             def point_at_t(self, t):
(8)                 """
(8)                 Evaluate the curve at a single point, returning a tuple of *d* floats.
(8)                 """
(8)                 return tuple(self(t))
(4)             @property
(4)             def control_points(self):
(8)                 """The control points of the curve."""
(8)                 return self._cpoints
(4)             @property
(4)             def dimension(self):
(8)                 """The dimension of the curve."""
(8)                 return self._d
(4)             @property
(4)             def degree(self):
(8)                 """Degree of the polynomial. One less the number of control points."""
(8)                 return self._N - 1
(4)             @property
(4)             def polynomial_coefficients(self):
(8)                 r"""
(8)                 The polynomial coefficients of the Bézier curve.
(8)                 .. warning:: Follows opposite convention from `numpy.polyval`.
(8)                 Returns
(8)                 -------
(8)                 (n+1, d) array
(12)                    Coefficients after expanding in polynomial basis, where :math:`n`
(12)                    is the degree of the Bézier curve and :math:`d` its dimension.
(12)                    These are the numbers (:math:`C_j`) such that the curve can be
(12)                    written :math:`\sum_{j=0}^n C_j t^j`.
(8)                 Notes
(8)                 -----
(8)                 The coefficients are calculated as
(8)                 .. math::
(12)                    {n \choose j} \sum_{i=0}^j (-1)^{i+j} {j \choose i} P_i
(8)                 where :math:`P_i` are the control points of the curve.
```

```
189: (8)                    """
190: (8)                    n = self.degree
191: (8)                    if n > 10:
192: (12)                       warnings.warn("Polynomial coefficients formula unstable for high "
193: (26)                                     "order Bezier curves!", RuntimeWarning)
194: (8)                    P = self.control_points
195: (8)                    j = np.arange(n+1)[:, None]
196: (8)                    i = np.arange(n+1)[None, :]  # _comb is non-zero for i <= j
197: (8)                    prefactor = (-1)**(i + j) * _comb(j, i)  # j on axis 0, i on axis 1
198: (8)                    return _comb(n, j) * prefactor @ P  # j on axis 0, self.dimension on 1
199: (4)                def axis_aligned_extrema(self):
200: (8)                    """
201: (8)                    Return the dimension and location of the curve's interior extrema.
202: (8)                    The extrema are the points along the curve where one of its partial
203: (8)                    derivatives is zero.
204: (8)                    Returns
205: (8)                    -------
206: (8)                    dims : array of int
207: (12)                       Index :math:`i` of the partial derivative which is zero at each
208: (12)                       interior extrema.
209: (8)                    dzeros : array of float
210: (12)                       Of same size as dims. The :math:`t` such that :math:`d/dx_i B(t) =
211: (12)                       0`
212: (8)                    """
213: (8)                    n = self.degree
214: (8)                    if n <= 1:
215: (12)                       return np.array([]), np.array([])
216: (8)                    Cj = self.polynomial_coefficients
217: (8)                    dCj = np.arange(1, n+1)[:, None] * Cj[1:]
218: (8)                    dims = []
219: (8)                    roots = []
220: (8)                    for i, pi in enumerate(dCj.T):
221: (12)                       r = np.roots(pi[::-1])
222: (12)                       roots.append(r)
223: (12)                       dims.append(np.full_like(r, i))
224: (8)                    roots = np.concatenate(roots)
225: (8)                    dims = np.concatenate(dims)
226: (8)                    in_range = np.isreal(roots) & (roots >= 0) & (roots <= 1)
227: (8)                    return dims[in_range], np.real(roots)[in_range]
228: (0)            def split_bezier_intersecting_with_closedpath(
229: (8)                    bezier, inside_closedpath, tolerance=0.01):
230: (4)                """
231: (4)                Split a Bézier curve into two at the intersection with a closed path.
232: (4)                Parameters
233: (4)                ----------
234: (4)                bezier : (N, 2) array-like
235: (8)                    Control points of the Bézier segment. See `.BezierSegment`.
236: (4)                inside_closedpath : callable
237: (8)                    A function returning True if a given point (x, y) is inside the
238: (8)                    closed path. See also `.find_bezier_t_intersecting_with_closedpath`.
239: (4)                tolerance : float
240: (8)                    The tolerance for the intersection. See also
241: (8)                    `.find_bezier_t_intersecting_with_closedpath`.
242: (4)                Returns
243: (4)                -------
244: (4)                left, right
245: (8)                    Lists of control points for the two Bézier segments.
246: (4)                """
247: (4)                bz = BezierSegment(bezier)
248: (4)                bezier_point_at_t = bz.point_at_t
249: (4)                t0, t1 = find_bezier_t_intersecting_with_closedpath(
250: (8)                    bezier_point_at_t, inside_closedpath, tolerance=tolerance)
251: (4)                _left, _right = split_de_casteljau(bezier, (t0 + t1) / 2.)
252: (4)                return _left, _right
253: (0)            def split_path_inout(path, inside, tolerance=0.01, reorder_inout=False):
254: (4)                """
255: (4)                Divide a path into two segments at the point where ``inside(x, y)``
becomes
256: (4)                False.
```

```
(4)                 """
(4)                 from .path import Path
(4)                 path_iter = path.iter_segments()
(4)                 ctl_points, command = next(path_iter)
(4)                 begin_inside = inside(ctl_points[-2:])  # true if begin point is inside
(4)                 ctl_points_old = ctl_points
(4)                 iold = 0
(4)                 i = 1
(4)                 for ctl_points, command in path_iter:
(8)                     iold = i
(8)                     i += len(ctl_points) // 2
(8)                     if inside(ctl_points[-2:]) != begin_inside:
(12)                        bezier_path = np.concatenate([ctl_points_old[-2:], ctl_points])
(12)                        break
(8)                     ctl_points_old = ctl_points
(4)                 else:
(8)                     raise ValueError("The path does not intersect with the patch")
(4)                 bp = bezier_path.reshape((-1, 2))
(4)                 left, right = split_bezier_intersecting_with_closedpath(
(8)                     bp, inside, tolerance)
(4)                 if len(left) == 2:
(8)                     codes_left = [Path.LINETO]
(8)                     codes_right = [Path.MOVETO, Path.LINETO]
(4)                 elif len(left) == 3:
(8)                     codes_left = [Path.CURVE3, Path.CURVE3]
(8)                     codes_right = [Path.MOVETO, Path.CURVE3, Path.CURVE3]
(4)                 elif len(left) == 4:
(8)                     codes_left = [Path.CURVE4, Path.CURVE4, Path.CURVE4]
(8)                     codes_right = [Path.MOVETO, Path.CURVE4, Path.CURVE4, Path.CURVE4]
(4)                 else:
(8)                     raise AssertionError("This should never be reached")
(4)                 verts_left = left[1:]
(4)                 verts_right = right[:]
(4)                 if path.codes is None:
(8)                     path_in = Path(np.concatenate([path.vertices[:i], verts_left]))
(8)                     path_out = Path(np.concatenate([verts_right, path.vertices[i:]]))
(4)                 else:
(8)                     path_in = Path(np.concatenate([path.vertices[:iold], verts_left]),
(23)                                   np.concatenate([path.codes[:iold], codes_left]))
(8)                     path_out = Path(np.concatenate([verts_right, path.vertices[i:]]),
(24)                                    np.concatenate([codes_right, path.codes[i:]]))
(4)                 if reorder_inout and not begin_inside:
(8)                     path_in, path_out = path_out, path_in
(4)                 return path_in, path_out
(0)             def inside_circle(cx, cy, r):
(4)                 """
(4)                 Return a function that checks whether a point is in a circle with center
(4)                 (*cx*, *cy*) and radius *r*.
(4)                 The returned function has the signature::
(8)                     f(xy: tuple[float, float]) -> bool
(4)                 """
(4)                 r2 = r ** 2
(4)                 def _f(xy):
(8)                     x, y = xy
(8)                     return (x - cx) ** 2 + (y - cy) ** 2 < r2
(4)                 return _f
(0)             def get_cos_sin(x0, y0, x1, y1):
(4)                 dx, dy = x1 - x0, y1 - y0
(4)                 d = (dx * dx + dy * dy) ** .5
(4)                 if d == 0:
(8)                     return 0.0, 0.0
(4)                 return dx / d, dy / d
(0)             def check_if_parallel(dx1, dy1, dx2, dy2, tolerance=1.e-5):
(4)                 """
(4)                 Check if two lines are parallel.
(4)                 Parameters
(4)                 ----------
(4)                 dx1, dy1, dx2, dy2 : float
(8)                     The gradients *dy*/*dx* of the two lines.
```

```
(4)                 tolerance : float
(8)                     The angular tolerance in radians up to which the lines are considered
(8)                     parallel.
(4)                 Returns
(4)                 -------
(4)                 is_parallel
(8)                     - 1 if two lines are parallel in same direction.
(8)                     - -1 if two lines are parallel in opposite direction.
(8)                     - False otherwise.
(4)                 """
(4)                 theta1 = np.arctan2(dx1, dy1)
(4)                 theta2 = np.arctan2(dx2, dy2)
(4)                 dtheta = abs(theta1 - theta2)
(4)                 if dtheta < tolerance:
(8)                     return 1
(4)                 elif abs(dtheta - np.pi) < tolerance:
(8)                     return -1
(4)                 else:
(8)                     return False
(0)             def get_parallels(bezier2, width):
(4)                 """
(4)                 Given the quadratic Bézier control points *bezier2*, returns
(4)                 control points of quadratic Bézier lines roughly parallel to given
(4)                 one separated by *width*.
(4)                 """
(4)                 c1x, c1y = bezier2[0]
(4)                 cmx, cmy = bezier2[1]
(4)                 c2x, c2y = bezier2[2]
(4)                 parallel_test = check_if_parallel(c1x - cmx, c1y - cmy,
(38)                                                  cmx - c2x, cmy - c2y)
(4)                 if parallel_test == -1:
(8)                     _api.warn_external(
(12)                        "Lines do not intersect. A straight line is used instead.")
(8)                     cos_t1, sin_t1 = get_cos_sin(c1x, c1y, c2x, c2y)
(8)                     cos_t2, sin_t2 = cos_t1, sin_t1
(4)                 else:
(8)                     cos_t1, sin_t1 = get_cos_sin(c1x, c1y, cmx, cmy)
(8)                     cos_t2, sin_t2 = get_cos_sin(cmx, cmy, c2x, c2y)
(4)                 c1x_left, c1y_left, c1x_right, c1y_right = (
(8)                     get_normal_points(c1x, c1y, cos_t1, sin_t1, width)
(4)                 )
(4)                 c2x_left, c2y_left, c2x_right, c2y_right = (
(8)                     get_normal_points(c2x, c2y, cos_t2, sin_t2, width)
(4)                 )
(4)                 try:
(8)                     cmx_left, cmy_left = get_intersection(c1x_left, c1y_left, cos_t1,
(46)                                                          sin_t1, c2x_left, c2y_left,
(46)                                                          cos_t2, sin_t2)
(8)                     cmx_right, cmy_right = get_intersection(c1x_right, c1y_right, cos_t1,
(48)                                                            sin_t1, c2x_right, c2y_right,
(48)                                                            cos_t2, sin_t2)
(4)                 except ValueError:
(8)                     cmx_left, cmy_left = (
(12)                        0.5 * (c1x_left + c2x_left), 0.5 * (c1y_left + c2y_left)
(8)                     )
(8)                     cmx_right, cmy_right = (
(12)                        0.5 * (c1x_right + c2x_right), 0.5 * (c1y_right + c2y_right)
(8)                     )
(4)                 path_left = [(c1x_left, c1y_left),
(17)                             (cmx_left, cmy_left),
(17)                             (c2x_left, c2y_left)]
(4)                 path_right = [(c1x_right, c1y_right),
(18)                              (cmx_right, cmy_right),
(18)                              (c2x_right, c2y_right)]
(4)                 return path_left, path_right
(0)             def find_control_points(c1x, c1y, mmx, mmy, c2x, c2y):
(4)                 """
(4)                 Find control points of the Bézier curve passing through (*c1x*, *c1y*),
(4)                 (*mmx*, *mmy*), and (*c2x*, *c2y*), at parametric values 0, 0.5, and 1.
```

```
395: (4)                """
396: (4)                cmx = .5 * (4 * mmx - (c1x + c2x))
397: (4)                cmy = .5 * (4 * mmy - (c1y + c2y))
398: (4)                return [(c1x, c1y), (cmx, cmy), (c2x, c2y)]
399: (0)            def make_wedged_bezier2(bezier2, width, w1=1., wm=0.5, w2=0.):
400: (4)                """
401: (4)                Being similar to `get_parallels`, returns control points of two quadratic
402: (4)                Bézier lines having a width roughly parallel to given one separated by
403: (4)                *width*.
404: (4)                """
405: (4)                c1x, c1y = bezier2[0]
406: (4)                cmx, cmy = bezier2[1]
407: (4)                c3x, c3y = bezier2[2]
408: (4)                cos_t1, sin_t1 = get_cos_sin(c1x, c1y, cmx, cmy)
409: (4)                cos_t2, sin_t2 = get_cos_sin(cmx, cmy, c3x, c3y)
410: (4)                c1x_left, c1y_left, c1x_right, c1y_right = (
411: (8)                    get_normal_points(c1x, c1y, cos_t1, sin_t1, width * w1)
412: (4)                )
413: (4)                c3x_left, c3y_left, c3x_right, c3y_right = (
414: (8)                    get_normal_points(c3x, c3y, cos_t2, sin_t2, width * w2)
415: (4)                )
416: (4)                c12x, c12y = (c1x + cmx) * .5, (c1y + cmy) * .5
417: (4)                c23x, c23y = (cmx + c3x) * .5, (cmy + c3y) * .5
418: (4)                c123x, c123y = (c12x + c23x) * .5, (c12y + c23y) * .5
419: (4)                cos_t123, sin_t123 = get_cos_sin(c12x, c12y, c23x, c23y)
420: (4)                c123x_left, c123y_left, c123x_right, c123y_right = (
421: (8)                    get_normal_points(c123x, c123y, cos_t123, sin_t123, width * wm)
422: (4)                )
423: (4)                path_left = find_control_points(c1x_left, c1y_left,
424: (36)                                               c123x_left, c123y_left,
425: (36)                                               c3x_left, c3y_left)
426: (4)                path_right = find_control_points(c1x_right, c1y_right,
427: (37)                                                c123x_right, c123y_right,
428: (37)                                                c3x_right, c3y_right)
429: (4)                return path_left, path_right
```

File 28 - category.py:

```
1: (0)              """
2: (0)              Plotting of string "category" data: ``plot(['d', 'f', 'a'], [1, 2, 3])`` will
3: (0)              plot three points with x-axis values of 'd', 'f', 'a'.
4: (0)              See :doc:`/gallery/lines_bars_and_markers/categorical_variables` for an
5: (0)              example.
6: (0)              The module uses Matplotlib's `matplotlib.units` mechanism to convert from
7: (0)              strings to integers and provides a tick locator, a tick formatter, and the
8: (0)              `.UnitData` class that creates and stores the string-to-integer mapping.
9: (0)              """
10: (0)             from collections import OrderedDict
11: (0)             import dateutil.parser
12: (0)             import itertools
13: (0)             import logging
14: (0)             import numpy as np
15: (0)             from matplotlib import _api, ticker, units
16: (0)             _log = logging.getLogger(__name__)
17: (0)             class StrCategoryConverter(units.ConversionInterface):
18: (4)                 @staticmethod
19: (4)                 def convert(value, unit, axis):
20: (8)                     """
21: (8)                     Convert strings in *value* to floats using mapping information stored
22: (8)                     in the *unit* object.
23: (8)                     Parameters
24: (8)                     ----------
25: (8)                     value : str or iterable
26: (12)                        Value or list of values to be converted.
27: (8)                     unit : `.UnitData`
28: (12)                        An object mapping strings to integers.
29: (8)                     axis : `~matplotlib.axis.Axis`
```

```
 30: (12)                        The axis on which the converted value is plotted.
 31: (12)                        .. note:: *axis* is unused.
 32: (8)                     Returns
 33: (8)                     -------
 34: (8)                     float or `~numpy.ndarray` of float
 35: (8)                     """
 36: (8)                     if unit is None:
 37: (12)                        raise ValueError(
 38: (16)                            'Missing category information for StrCategoryConverter; '
 39: (16)                            'this might be caused by unintendedly mixing categorical and '
 40: (16)                            'numeric data')
 41: (8)                     StrCategoryConverter._validate_unit(unit)
 42: (8)                     values = np.atleast_1d(np.array(value, dtype=object))
 43: (8)                     unit.update(values)
 44: (8)                     return np.vectorize(unit._mapping.__getitem__, otypes=[float])(values)
 45: (4)                 @staticmethod
 46: (4)                 def axisinfo(unit, axis):
 47: (8)                     """
 48: (8)                     Set the default axis ticks and labels.
 49: (8)                     Parameters
 50: (8)                     ----------
 51: (8)                     unit : `.UnitData`
 52: (12)                        object string unit information for value
 53: (8)                     axis : `~matplotlib.axis.Axis`
 54: (12)                        axis for which information is being set
 55: (12)                        .. note:: *axis* is not used
 56: (8)                     Returns
 57: (8)                     -------
 58: (8)                     `~matplotlib.units.AxisInfo`
 59: (12)                        Information to support default tick labeling
 60: (8)                     """
 61: (8)                     StrCategoryConverter._validate_unit(unit)
 62: (8)                     majloc = StrCategoryLocator(unit._mapping)
 63: (8)                     majfmt = StrCategoryFormatter(unit._mapping)
 64: (8)                     return units.AxisInfo(majloc=majloc, majfmt=majfmt)
 65: (4)                 @staticmethod
 66: (4)                 def default_units(data, axis):
 67: (8)                     """
 68: (8)                     Set and update the `~matplotlib.axis.Axis` units.
 69: (8)                     Parameters
 70: (8)                     ----------
 71: (8)                     data : str or iterable of str
 72: (8)                     axis : `~matplotlib.axis.Axis`
 73: (12)                        axis on which the data is plotted
 74: (8)                     Returns
 75: (8)                     -------
 76: (8)                     `.UnitData`
 77: (12)                        object storing string to integer mapping
 78: (8)                     """
 79: (8)                     if axis.units is None:
 80: (12)                        axis.set_units(UnitData(data))
 81: (8)                     else:
 82: (12)                        axis.units.update(data)
 83: (8)                     return axis.units
 84: (4)                 @staticmethod
 85: (4)                 def _validate_unit(unit):
 86: (8)                     if not hasattr(unit, '_mapping'):
 87: (12)                        raise ValueError(
 88: (16)                            f'Provided unit "{unit}" is not valid for a categorical '
 89: (16)                            'converter, as it does not have a _mapping attribute.')
 90: (0)             class StrCategoryLocator(ticker.Locator):
 91: (4)                 """Tick at every integer mapping of the string data."""
 92: (4)                 def __init__(self, units_mapping):
 93: (8)                     """
 94: (8)                     Parameters
 95: (8)                     ----------
 96: (8)                     units_mapping : dict
 97: (12)                        Mapping of category names (str) to indices (int).
 98: (8)                     """
```

```
99: (8)                     self._units = units_mapping
100: (4)                def __call__(self):
101: (8)                    return list(self._units.values())
102: (4)                def tick_values(self, vmin, vmax):
103: (8)                    return self()
104: (0)            class StrCategoryFormatter(ticker.Formatter):
105: (4)                """String representation of the data at every tick."""
106: (4)                def __init__(self, units_mapping):
107: (8)                    """
108: (8)                    Parameters
109: (8)                    ----------
110: (8)                    units_mapping : dict
111: (12)                       Mapping of category names (str) to indices (int).
112: (8)                    """
113: (8)                    self._units = units_mapping
114: (4)                def __call__(self, x, pos=None):
115: (8)                    return self.format_ticks([x])[0]
116: (4)                def format_ticks(self, values):
117: (8)                    r_mapping = {v: self._text(k) for k, v in self._units.items()}
118: (8)                    return [r_mapping.get(round(val), '') for val in values]
119: (4)                @staticmethod
120: (4)                def _text(value):
121: (8)                    """Convert text values into utf-8 or ascii strings."""
122: (8)                    if isinstance(value, bytes):
123: (12)                       value = value.decode(encoding='utf-8')
124: (8)                    elif not isinstance(value, str):
125: (12)                       value = str(value)
126: (8)                    return value
127: (0)            class UnitData:
128: (4)                def __init__(self, data=None):
129: (8)                    """
130: (8)                    Create mapping between unique categorical values and integer ids.
131: (8)                    Parameters
132: (8)                    ----------
133: (8)                    data : iterable
134: (12)                       sequence of string values
135: (8)                    """
136: (8)                    self._mapping = OrderedDict()
137: (8)                    self._counter = itertools.count()
138: (8)                    if data is not None:
139: (12)                       self.update(data)
140: (4)                @staticmethod
141: (4)                def _str_is_convertible(val):
142: (8)                    """
143: (8)                    Helper method to check whether a string can be parsed as float or
date.
144: (8)                    """
145: (8)                    try:
146: (12)                       float(val)
147: (8)                    except ValueError:
148: (12)                       try:
149: (16)                           dateutil.parser.parse(val)
150: (12)                       except (ValueError, TypeError):
151: (16)                           return False
152: (8)                    return True
153: (4)                def update(self, data):
154: (8)                    """
155: (8)                    Map new values to integer identifiers.
156: (8)                    Parameters
157: (8)                    ----------
158: (8)                    data : iterable of str or bytes
159: (8)                    Raises
160: (8)                    ------
161: (8)                    TypeError
162: (12)                       If elements in *data* are neither str nor bytes.
163: (8)                    """
164: (8)                    data = np.atleast_1d(np.array(data, dtype=object))
165: (8)                    convertible = True
166: (8)                    for val in OrderedDict.fromkeys(data):
```

```
167: (12)                        _api.check_isinstance((str, bytes), value=val)
168: (12)                        if convertible:
169: (16)                            convertible = self._str_is_convertible(val)
170: (12)                        if val not in self._mapping:
171: (16)                            self._mapping[val] = next(self._counter)
172: (8)                    if data.size and convertible:
173: (12)                        _log.info('Using categorical units to plot a list of strings '
174: (22)                                  'that are all parsable as floats or dates. If these '
175: (22)                                  'strings should be plotted as numbers, cast to the '
176: (22)                                  'appropriate data type before plotting.')
177: (0)            units.registry[str] = StrCategoryConverter()
178: (0)            units.registry[np.str_] = StrCategoryConverter()
179: (0)            units.registry[bytes] = StrCategoryConverter()
180: (0)            units.registry[np.bytes_] = StrCategoryConverter()
```

File 29 - cbook.py:

```
1: (0)              """
2: (0)              A collection of utility functions and classes.  Originally, many
3: (0)              (but not all) were from the Python Cookbook -- hence the name cbook.
4: (0)              """
5: (0)              import collections
6: (0)              import collections.abc
7: (0)              import contextlib
8: (0)              import functools
9: (0)              import gzip
10: (0)             import itertools
11: (0)             import math
12: (0)             import operator
13: (0)             import os
14: (0)             from pathlib import Path
15: (0)             import shlex
16: (0)             import subprocess
17: (0)             import sys
18: (0)             import time
19: (0)             import traceback
20: (0)             import types
21: (0)             import weakref
22: (0)             import numpy as np
23: (0)             try:
24: (4)                 from numpy.exceptions import VisibleDeprecationWarning  # numpy >= 1.25
25: (0)             except ImportError:
26: (4)                 from numpy import VisibleDeprecationWarning
27: (0)             import matplotlib
28: (0)             from matplotlib import _api, _c_internal_utils
29: (0)             def _get_running_interactive_framework():
30: (4)                 """
31: (4)                 Return the interactive framework whose event loop is currently running, if
32: (4)                 any, or "headless" if no event loop can be started, or None.
33: (4)                 Returns
34: (4)                 -------
35: (4)                 Optional[str]
36: (8)                     One of the following values: "qt", "gtk3", "gtk4", "wx", "tk",
37: (8)                     "macosx", "headless", ``None``.
38: (4)                 """
39: (4)                 QtWidgets = (
40: (8)                     sys.modules.get("PyQt6.QtWidgets")
41: (8)                     or sys.modules.get("PySide6.QtWidgets")
42: (8)                     or sys.modules.get("PyQt5.QtWidgets")
43: (8)                     or sys.modules.get("PySide2.QtWidgets")
44: (4)                 )
45: (4)                 if QtWidgets and QtWidgets.QApplication.instance():
46: (8)                     return "qt"
47: (4)                 Gtk = sys.modules.get("gi.repository.Gtk")
48: (4)                 if Gtk:
49: (8)                     if Gtk.MAJOR_VERSION == 4:
50: (12)                        from gi.repository import GLib
```

```
51: (12)                        if GLib.main_depth():
52: (16)                            return "gtk4"
53: (8)                     if Gtk.MAJOR_VERSION == 3 and Gtk.main_level():
54: (12)                        return "gtk3"
55: (4)                 wx = sys.modules.get("wx")
56: (4)                 if wx and wx.GetApp():
57: (8)                     return "wx"
58: (4)                 tkinter = sys.modules.get("tkinter")
59: (4)                 if tkinter:
60: (8)                     codes = {tkinter.mainloop.__code__, tkinter.Misc.mainloop.__code__}
61: (8)                     for frame in sys._current_frames().values():
62: (12)                        while frame:
63: (16)                            if frame.f_code in codes:
64: (20)                                return "tk"
65: (16)                            frame = frame.f_back
66: (8)                     del frame
67: (4)                 macosx = sys.modules.get("matplotlib.backends._macosx")
68: (4)                 if macosx and macosx.event_loop_is_running():
69: (8)                     return "macosx"
70: (4)                 if not _c_internal_utils.display_is_valid():
71: (8)                     return "headless"
72: (4)                 return None
73: (0)             def _exception_printer(exc):
74: (4)                 if _get_running_interactive_framework() in ["headless", None]:
75: (8)                     raise exc
76: (4)                 else:
77: (8)                     traceback.print_exc()
78: (0)             class _StrongRef:
79: (4)                 """
80: (4)                 Wrapper similar to a weakref, but keeping a strong reference to the
object.
81: (4)                 """
82: (4)                 def __init__(self, obj):
83: (8)                     self._obj = obj
84: (4)                 def __call__(self):
85: (8)                     return self._obj
86: (4)                 def __eq__(self, other):
87: (8)                     return isinstance(other, _StrongRef) and self._obj == other._obj
88: (4)                 def __hash__(self):
89: (8)                     return hash(self._obj)
90: (0)             def _weak_or_strong_ref(func, callback):
91: (4)                 """
92: (4)                 Return a `WeakMethod` wrapping *func* if possible, else a `_StrongRef`.
93: (4)                 """
94: (4)                 try:
95: (8)                     return weakref.WeakMethod(func, callback)
96: (4)                 except TypeError:
97: (8)                     return _StrongRef(func)
98: (0)             class CallbackRegistry:
99: (4)                 """
100: (4)                Handle registering, processing, blocking, and disconnecting
101: (4)                for a set of signals and callbacks:
102: (8)                    >>> def oneat(x):
103: (8)                    ...     print('eat', x)
104: (8)                    >>> def ondrink(x):
105: (8)                    ...     print('drink', x)
106: (8)                    >>> from matplotlib.cbook import CallbackRegistry
107: (8)                    >>> callbacks = CallbackRegistry()
108: (8)                    >>> id_eat = callbacks.connect('eat', oneat)
109: (8)                    >>> id_drink = callbacks.connect('drink', ondrink)
110: (8)                    >>> callbacks.process('drink', 123)
111: (8)                    drink 123
112: (8)                    >>> callbacks.process('eat', 456)
113: (8)                    eat 456
114: (8)                    >>> callbacks.process('be merry', 456)   # nothing will be called
115: (8)                    >>> callbacks.disconnect(id_eat)
116: (8)                    >>> callbacks.process('eat', 456)        # nothing will be called
117: (8)                    >>> with callbacks.blocked(signal='drink'):
118: (8)                    ...     callbacks.process('drink', 123)  # nothing will be called
```

```
(8)                 >>> callbacks.process('drink', 123)
(8)                 drink 123
(4)             In practice, one should always disconnect all callbacks when they are
(4)             no longer needed to avoid dangling references (and thus memory leaks).
(4)             However, real code in Matplotlib rarely does so, and due to its design,
(4)             it is rather difficult to place this kind of code.  To get around this,
(4)             and prevent this class of memory leaks, we instead store weak references
(4)             to bound methods only, so when the destination object needs to die, the
(4)             CallbackRegistry won't keep it alive.
(4)             Parameters
(4)             ----------
(4)             exception_handler : callable, optional
(7)                If not None, *exception_handler* must be a function that takes an
(7)                `Exception` as single parameter.  It gets called with any `Exception`
(7)                raised by the callbacks during `CallbackRegistry.process`, and may
(7)                either re-raise the exception or handle it in another manner.
(7)                The default handler prints the exception (with `traceback.print_exc`)
if
(7)                an interactive event loop is running; it re-raises the exception if no
(7)                interactive event loop is running.
(4)             signals : list, optional
(8)                 If not None, *signals* is a list of signals that this registry
handles:
(8)                 attempting to `process` or to `connect` to a signal not in the list
(8)                 throws a `ValueError`.  The default, None, does not restrict the
(8)                 handled signals.
(4)             """
(4)             def __init__(self, exception_handler=_exception_printer, *, signals=None):
(8)                 self._signals = None if signals is None else list(signals)  # Copy it.
(8)                 self.exception_handler = exception_handler
(8)                 self.callbacks = {}
(8)                 self._cid_gen = itertools.count()
(8)                 self._func_cid_map = {}
(8)                 self._pickled_cids = set()
(4)             def __getstate__(self):
(8)                 return {
(12)                    **vars(self),
(12)                    "callbacks": {s: {cid: proxy() for cid, proxy in d.items()
(30)                                      if cid in self._pickled_cids}
(26)                                  for s, d in self.callbacks.items()},
(12)                    "_func_cid_map": None,
(12)                    "_cid_gen": next(self._cid_gen)
(8)                 }
(4)             def __setstate__(self, state):
(8)                 cid_count = state.pop('_cid_gen')
(8)                 vars(self).update(state)
(8)                 self.callbacks = {
(12)                    s: {cid: _weak_or_strong_ref(func, self._remove_proxy)
(16)                        for cid, func in d.items()}
(12)                    for s, d in self.callbacks.items()}
(8)                 self._func_cid_map = {
(12)                    s: {proxy: cid for cid, proxy in d.items()}
(12)                    for s, d in self.callbacks.items()}
(8)                 self._cid_gen = itertools.count(cid_count)
(4)             def connect(self, signal, func):
(8)                 """Register *func* to be called when signal *signal* is generated."""
(8)                 if self._signals is not None:
(12)                    _api.check_in_list(self._signals, signal=signal)
(8)                 self._func_cid_map.setdefault(signal, {})
(8)                 proxy = _weak_or_strong_ref(func, self._remove_proxy)
(8)                 if proxy in self._func_cid_map[signal]:
(12)                    return self._func_cid_map[signal][proxy]
(8)                 cid = next(self._cid_gen)
(8)                 self._func_cid_map[signal][proxy] = cid
(8)                 self.callbacks.setdefault(signal, {})
(8)                 self.callbacks[signal][cid] = proxy
(8)                 return cid
(4)             def _connect_picklable(self, signal, func):
(8)                 """
```

```
186: (8)                    Like `.connect`, but the callback is kept when pickling/unpickling.
187: (8)                    Currently internal-use only.
188: (8)                    """
189: (8)                    cid = self.connect(signal, func)
190: (8)                    self._pickled_cids.add(cid)
191: (8)                    return cid
192: (4)                def _remove_proxy(self, proxy, *, _is_finalizing=sys.is_finalizing):
193: (8)                    if _is_finalizing():
194: (12)                       return
195: (8)                    for signal, proxy_to_cid in list(self._func_cid_map.items()):
196: (12)                       cid = proxy_to_cid.pop(proxy, None)
197: (12)                       if cid is not None:
198: (16)                           del self.callbacks[signal][cid]
199: (16)                           self._pickled_cids.discard(cid)
200: (16)                           break
201: (8)                    else:
202: (12)                       return
203: (8)                    if len(self.callbacks[signal]) == 0:
204: (12)                       del self.callbacks[signal]
205: (12)                       del self._func_cid_map[signal]
206: (4)                def disconnect(self, cid):
207: (8)                    """
208: (8)                    Disconnect the callback registered with callback id *cid*.
209: (8)                    No error is raised if such a callback does not exist.
210: (8)                    """
211: (8)                    self._pickled_cids.discard(cid)
212: (8)                    for signal, cid_to_proxy in list(self.callbacks.items()):
213: (12)                       proxy = cid_to_proxy.pop(cid, None)
214: (12)                       if proxy is not None:
215: (16)                           break
216: (8)                    else:
217: (12)                       return
218: (8)                    proxy_to_cid = self._func_cid_map[signal]
219: (8)                    for current_proxy, current_cid in list(proxy_to_cid.items()):
220: (12)                       if current_cid == cid:
221: (16)                           assert proxy is current_proxy
222: (16)                           del proxy_to_cid[current_proxy]
223: (8)                    if len(self.callbacks[signal]) == 0:
224: (12)                       del self.callbacks[signal]
225: (12)                       del self._func_cid_map[signal]
226: (4)                def process(self, s, *args, **kwargs):
227: (8)                    """
228: (8)                    Process signal *s*.
229: (8)                    All of the functions registered to receive callbacks on *s* will be
230: (8)                    called with ``*args`` and ``**kwargs``.
231: (8)                    """
232: (8)                    if self._signals is not None:
233: (12)                       _api.check_in_list(self._signals, signal=s)
234: (8)                    for ref in list(self.callbacks.get(s, {}).values()):
235: (12)                       func = ref()
236: (12)                       if func is not None:
237: (16)                           try:
238: (20)                               func(*args, **kwargs)
239: (16)                           except Exception as exc:
240: (20)                               if self.exception_handler is not None:
241: (24)                                   self.exception_handler(exc)
242: (20)                               else:
243: (24)                                   raise
244: (4)                @contextlib.contextmanager
245: (4)                def blocked(self, *, signal=None):
246: (8)                    """
247: (8)                    Block callback signals from being processed.
248: (8)                    A context manager to temporarily block/disable callback signals
249: (8)                    from being processed by the registered listeners.
250: (8)                    Parameters
251: (8)                    ----------
252: (8)                    signal : str, optional
253: (12)                       The callback signal to block. The default is to block all signals.
254: (8)                    """
```

```
255: (8)                    orig = self.callbacks
256: (8)                    try:
257: (12)                       if signal is None:
258: (16)                           self.callbacks = {}
259: (12)                       else:
260: (16)                           self.callbacks = {k: orig[k] for k in orig if k != signal}
261: (12)                       yield
262: (8)                    finally:
263: (12)                       self.callbacks = orig
264: (0)            class silent_list(list):
265: (4)                """
266: (4)                A list with a short ``repr()``.
267: (4)                This is meant to be used for a homogeneous list of artists, so that they
268: (4)                don't cause long, meaningless output.
269: (4)                Instead of ::
270: (8)                    [<matplotlib.lines.Line2D object at 0x7f5749fed3c8>,
271: (9)                     <matplotlib.lines.Line2D object at 0x7f5749fed4e0>,
272: (9)                     <matplotlib.lines.Line2D object at 0x7f5758016550>]
273: (4)                one will get ::
274: (8)                    <a list of 3 Line2D objects>
275: (4)                If ``self.type`` is None, the type name is obtained from the first item in
276: (4)                the list (if any).
277: (4)                """
278: (4)                def __init__(self, type, seq=None):
279: (8)                    self.type = type
280: (8)                    if seq is not None:
281: (12)                       self.extend(seq)
282: (4)                def __repr__(self):
283: (8)                    if self.type is not None or len(self) != 0:
284: (12)                       tp = self.type if self.type is not None else
type(self[0]).__name__
285: (12)                       return f"<a list of {len(self)} {tp} objects>"
286: (8)                    else:
287: (12)                       return "<an empty list>"
288: (0)            def _local_over_kwdict(
289: (8)                    local_var, kwargs, *keys,
290: (8)                    warning_cls=_api.MatplotlibDeprecationWarning):
291: (4)                out = local_var
292: (4)                for key in keys:
293: (8)                    kwarg_val = kwargs.pop(key, None)
294: (8)                    if kwarg_val is not None:
295: (12)                       if out is None:
296: (16)                           out = kwarg_val
297: (12)                       else:
298: (16)                           _api.warn_external(f'"{key}" keyword argument will be
ignored',
299: (35)                                              warning_cls)
300: (4)                return out
301: (0)            def strip_math(s):
302: (4)                """
303: (4)                Remove latex formatting from mathtext.
304: (4)                Only handles fully math and fully non-math strings.
305: (4)                """
306: (4)                if len(s) >= 2 and s[0] == s[-1] == "$":
307: (8)                    s = s[1:-1]
308: (8)                    for tex, plain in [
309: (16)                           (r"\times", "x"),  # Specifically for Formatter support.
310: (16)                           (r"\mathdefault", ""),
311: (16)                           (r"\rm", ""),
312: (16)                           (r"\cal", ""),
313: (16)                           (r"\tt", ""),
314: (16)                           (r"\it", ""),
315: (16)                           ("\\", ""),
316: (16)                           ("{", ""),
317: (16)                           ("}", ""),
318: (8)                    ]:
319: (12)                       s = s.replace(tex, plain)
320: (4)                return s
321: (0)            def _strip_comment(s):
```

```
(4)                 """Strip everything from the first unquoted #."""
(4)                 pos = 0
(4)                 while True:
(8)                     quote_pos = s.find('"', pos)
(8)                     hash_pos = s.find('#', pos)
(8)                     if quote_pos < 0:
(12)                        without_comment = s if hash_pos < 0 else s[:hash_pos]
(12)                        return without_comment.strip()
(8)                     elif 0 <= hash_pos < quote_pos:
(12)                        return s[:hash_pos].strip()
(8)                     else:
(12)                        closing_quote_pos = s.find('"', quote_pos + 1)
(12)                        if closing_quote_pos < 0:
(16)                            raise ValueError(
(20)                                f"Missing closing quote in: {s!r}. If you need a double-"
(20)                                'quote inside a string, use escaping: e.g. "the \" char"')
(12)                        pos = closing_quote_pos + 1  # behind closing quote
(0)             def is_writable_file_like(obj):
(4)                 """Return whether *obj* looks like a file object with a *write* method."""
(4)                 return callable(getattr(obj, 'write', None))
(0)             def file_requires_unicode(x):
(4)                 """
(4)                 Return whether the given writable file-like object requires Unicode to be
(4)                 written to it.
(4)                 """
(4)                 try:
(8)                     x.write(b'')
(4)                 except TypeError:
(8)                     return True
(4)                 else:
(8)                     return False
(0)             def to_filehandle(fname, flag='r', return_opened=False, encoding=None):
(4)                 """
(4)                 Convert a path to an open file handle or pass-through a file-like object.
(4)                 Consider using `open_file_cm` instead, as it allows one to properly close
(4)                 newly created file objects more easily.
(4)                 Parameters
(4)                 ----------
(4)                 fname : str or path-like or file-like
(8)                     If `str` or `os.PathLike`, the file is opened using the flags
specified
(8)                     by *flag* and *encoding*.  If a file-like object, it is passed
through.
(4)                 flag : str, default: 'r'
(8)                     Passed as the *mode* argument to `open` when *fname* is `str` or
(8)                     `os.PathLike`; ignored if *fname* is file-like.
(4)                 return_opened : bool, default: False
(8)                     If True, return both the file object and a boolean indicating whether
(8)                     this was a new file (that the caller needs to close).  If False,
return
(8)                     only the new file.
(4)                 encoding : str or None, default: None
(8)                     Passed as the *mode* argument to `open` when *fname* is `str` or
(8)                     `os.PathLike`; ignored if *fname* is file-like.
(4)                 Returns
(4)                 -------
(4)                 fh : file-like
(4)                 opened : bool
(8)                     *opened* is only returned if *return_opened* is True.
(4)                 """
(4)                 if isinstance(fname, os.PathLike):
(8)                     fname = os.fspath(fname)
(4)                 if isinstance(fname, str):
(8)                     if fname.endswith('.gz'):
(12)                        fh = gzip.open(fname, flag)
(8)                     elif fname.endswith('.bz2'):
(12)                        import bz2
(12)                        fh = bz2.BZ2File(fname, flag)
(8)                     else:
```

```
388: (12)                        fh = open(fname, flag, encoding=encoding)
389: (8)                    opened = True
390: (4)                elif hasattr(fname, 'seek'):
391: (8)                    fh = fname
392: (8)                    opened = False
393: (4)                else:
394: (8)                    raise ValueError('fname must be a PathLike or file handle')
395: (4)                if return_opened:
396: (8)                    return fh, opened
397: (4)                return fh
398: (0)            def open_file_cm(path_or_file, mode="r", encoding=None):
399: (4)                r"""Pass through file objects and context-manage path-likes."""
400: (4)                fh, opened = to_filehandle(path_or_file, mode, True, encoding)
401: (4)                return fh if opened else contextlib.nullcontext(fh)
402: (0)            def is_scalar_or_string(val):
403: (4)                """Return whether the given object is a scalar or string like."""
404: (4)                return isinstance(val, str) or not np.iterable(val)
405: (0)            @_api.delete_parameter(
406: (4)                "3.8", "np_load", alternative="open(get_sample_data(...,
asfileobj=False))")
407: (0)            def get_sample_data(fname, asfileobj=True, *, np_load=True):
408: (4)                """
409: (4)                Return a sample data file.  *fname* is a path relative to the
410: (4)                :file:`mpl-data/sample_data` directory.  If *asfileobj* is `True`
411: (4)                return a file object, otherwise just a file path.
412: (4)                Sample data files are stored in the 'mpl-data/sample_data' directory
within
413: (4)                the Matplotlib package.
414: (4)                If the filename ends in .gz, the file is implicitly ungzipped.  If the
415: (4)                filename ends with .npy or .npz, and *asfileobj* is `True`, the file is
416: (4)                loaded with `numpy.load`.
417: (4)                """
418: (4)                path = _get_data_path('sample_data', fname)
419: (4)                if asfileobj:
420: (8)                    suffix = path.suffix.lower()
421: (8)                    if suffix == '.gz':
422: (12)                       return gzip.open(path)
423: (8)                    elif suffix in ['.npy', '.npz']:
424: (12)                       if np_load:
425: (16)                           return np.load(path)
426: (12)                       else:
427: (16)                           return path.open('rb')
428: (8)                    elif suffix in ['.csv', '.xrc', '.txt']:
429: (12)                       return path.open('r')
430: (8)                    else:
431: (12)                       return path.open('rb')
432: (4)                else:
433: (8)                    return str(path)
434: (0)            def _get_data_path(*args):
435: (4)                """
436: (4)                Return the `pathlib.Path` to a resource file provided by Matplotlib.
437: (4)                ``*args`` specify a path relative to the base data path.
438: (4)                """
439: (4)                return Path(matplotlib.get_data_path(), *args)
440: (0)            def flatten(seq, scalarp=is_scalar_or_string):
441: (4)                """
442: (4)                Return a generator of flattened nested containers.
443: (4)                For example:
444: (8)                    >>> from matplotlib.cbook import flatten
445: (8)                    >>> l = (('John', ['Hunter']), (1, 23), [[([42, (5, 23)], )]])
446: (8)                    >>> print(list(flatten(l)))
447: (8)                    ['John', 'Hunter', 1, 23, 42, 5, 23]
448: (4)                By: Composite of Holger Krekel and Luther Blissett
449: (4)                From: https://code.activestate.com/recipes/121294/
450: (4)                and Recipe 1.12 in cookbook
451: (4)                """
452: (4)                for item in seq:
453: (8)                    if scalarp(item) or item is None:
454: (12)                       yield item
```

```
455: (8)                    else:
456: (12)                       yield from flatten(item, scalarp)
457: (0)            @_api.deprecated("3.8")
458: (0)            class Stack:
459: (4)                """
460: (4)                Stack of elements with a movable cursor.
461: (4)                Mimics home/back/forward in a web browser.
462: (4)                """
463: (4)                def __init__(self, default=None):
464: (8)                    self.clear()
465: (8)                    self._default = default
466: (4)                def __call__(self):
467: (8)                    """Return the current element, or None."""
468: (8)                    if not self._elements:
469: (12)                       return self._default
470: (8)                    else:
471: (12)                       return self._elements[self._pos]
472: (4)                def __len__(self):
473: (8)                    return len(self._elements)
474: (4)                def __getitem__(self, ind):
475: (8)                    return self._elements[ind]
476: (4)                def forward(self):
477: (8)                    """Move the position forward and return the current element."""
478: (8)                    self._pos = min(self._pos + 1, len(self._elements) - 1)
479: (8)                    return self()
480: (4)                def back(self):
481: (8)                    """Move the position back and return the current element."""
482: (8)                    if self._pos > 0:
483: (12)                       self._pos -= 1
484: (8)                    return self()
485: (4)                def push(self, o):
486: (8)                    """
487: (8)                    Push *o* to the stack at current position.  Discard all later
elements.
488: (8)                    *o* is returned.
489: (8)                    """
490: (8)                    self._elements = self._elements[:self._pos + 1] + [o]
491: (8)                    self._pos = len(self._elements) - 1
492: (8)                    return self()
493: (4)                def home(self):
494: (8)                    """
495: (8)                    Push the first element onto the top of the stack.
496: (8)                    The first element is returned.
497: (8)                    """
498: (8)                    if not self._elements:
499: (12)                       return
500: (8)                    self.push(self._elements[0])
501: (8)                    return self()
502: (4)                def empty(self):
503: (8)                    """Return whether the stack is empty."""
504: (8)                    return len(self._elements) == 0
505: (4)                def clear(self):
506: (8)                    """Empty the stack."""
507: (8)                    self._pos = -1
508: (8)                    self._elements = []
509: (4)                def bubble(self, o):
510: (8)                    """
511: (8)                    Raise all references of *o* to the top of the stack, and return it.
512: (8)                    Raises
513: (8)                    ------
514: (8)                    ValueError
515: (12)                       If *o* is not in the stack.
516: (8)                    """
517: (8)                    if o not in self._elements:
518: (12)                       raise ValueError('Given element not contained in the stack')
519: (8)                    old_elements = self._elements.copy()
520: (8)                    self.clear()
521: (8)                    top_elements = []
522: (8)                    for elem in old_elements:
```

```
(12)                        if elem == o:
(16)                            top_elements.append(elem)
(12)                        else:
(16)                            self.push(elem)
(8)                     for _ in top_elements:
(12)                        self.push(o)
(8)                     return o
(4)                 def remove(self, o):
(8)                     """
(8)                     Remove *o* from the stack.
(8)                     Raises
(8)                     ------
(8)                     ValueError
(12)                        If *o* is not in the stack.
(8)                     """
(8)                     if o not in self._elements:
(12)                        raise ValueError('Given element not contained in the stack')
(8)                     old_elements = self._elements.copy()
(8)                     self.clear()
(8)                     for elem in old_elements:
(12)                        if elem != o:
(16)                            self.push(elem)
(0)             class _Stack:
(4)                 """
(4)                 Stack of elements with a movable cursor.
(4)                 Mimics home/back/forward in a web browser.
(4)                 """
(4)                 def __init__(self):
(8)                     self._pos = -1
(8)                     self._elements = []
(4)                 def clear(self):
(8)                     """Empty the stack."""
(8)                     self._pos = -1
(8)                     self._elements = []
(4)                 def __call__(self):
(8)                     """Return the current element, or None."""
(8)                     return self._elements[self._pos] if self._elements else None
(4)                 def __len__(self):
(8)                     return len(self._elements)
(4)                 def __getitem__(self, ind):
(8)                     return self._elements[ind]
(4)                 def forward(self):
(8)                     """Move the position forward and return the current element."""
(8)                     self._pos = min(self._pos + 1, len(self._elements) - 1)
(8)                     return self()
(4)                 def back(self):
(8)                     """Move the position back and return the current element."""
(8)                     self._pos = max(self._pos - 1, 0)
(8)                     return self()
(4)                 def push(self, o):
(8)                     """
(8)                     Push *o* to the stack after the current position, and return *o*.
(8)                     Discard all later elements.
(8)                     """
(8)                     self._elements[self._pos + 1:] = [o]
(8)                     self._pos = len(self._elements) - 1
(8)                     return o
(4)                 def home(self):
(8)                     """
(8)                     Push the first element onto the top of the stack.
(8)                     The first element is returned.
(8)                     """
(8)                     return self.push(self._elements[0]) if self._elements else None
(0)             def safe_masked_invalid(x, copy=False):
(4)                 x = np.array(x, subok=True, copy=copy)
(4)                 if not x.dtype.isnative:
(8)                     x = x.byteswap(inplace=copy).view(x.dtype.newbyteorder('N'))
(4)                 try:
(8)                     xm = np.ma.masked_where(~(np.isfinite(x)), x, copy=False)
```

```
(4)                 except TypeError:
(8)                     return x
(4)                 return xm
(0)             def print_cycles(objects, outstream=sys.stdout, show_progress=False):
(4)                 """
(4)                 Print loops of cyclic references in the given *objects*.
(4)                 It is often useful to pass in ``gc.garbage`` to find the cycles that are
(4)                 preventing some objects from being garbage collected.
(4)                 Parameters
(4)                 ----------
(4)                 objects
(8)                     A list of objects to find cycles in.
(4)                 outstream
(8)                     The stream for output.
(4)                 show_progress : bool
(8)                     If True, print the number of objects reached as they are found.
(4)                 """
(4)                 import gc
(4)                 def print_path(path):
(8)                     for i, step in enumerate(path):
(12)                        next = path[(i + 1) % len(path)]
(12)                        outstream.write("   %s -- " % type(step))
(12)                        if isinstance(step, dict):
(16)                            for key, val in step.items():
(20)                                if val is next:
(24)                                    outstream.write(f"[{key!r}]")
(24)                                    break
(20)                                if key is next:
(24)                                    outstream.write(f"[key] = {val!r}")
(24)                                    break
(12)                        elif isinstance(step, list):
(16)                            outstream.write("[%d]" % step.index(next))
(12)                        elif isinstance(step, tuple):
(16)                            outstream.write("( tuple )")
(12)                        else:
(16)                            outstream.write(repr(step))
(12)                        outstream.write(" ->\n")
(8)                     outstream.write("\n")
(4)                 def recurse(obj, start, all, current_path):
(8)                     if show_progress:
(12)                        outstream.write("%d\r" % len(all))
(8)                     all[id(obj)] = None
(8)                     referents = gc.get_referents(obj)
(8)                     for referent in referents:
(12)                        if referent is start:
(16)                            print_path(current_path)
(12)                        elif referent is objects or isinstance(referent, types.FrameType):
(16)                            continue
(12)                        elif id(referent) not in all:
(16)                            recurse(referent, start, all, current_path + [obj])
(4)                 for obj in objects:
(8)                     outstream.write(f"Examining: {obj!r}\n")
(8)                     recurse(obj, obj, {}, [])
(0)             class Grouper:
(4)                 """
(4)                 A disjoint-set data structure.
(4)                 Objects can be joined using :meth:`join`, tested for connectedness
(4)                 using :meth:`joined`, and all disjoint sets can be retrieved by
(4)                 using the object as an iterator.
(4)                 The objects being joined must be hashable and weak-referenceable.
(4)                 Examples
(4)                 --------
(4)                 >>> from matplotlib.cbook import Grouper
(4)                 >>> class Foo:
(4)                 ...     def __init__(self, s):
(4)                 ...         self.s = s
(4)                 ...     def __repr__(self):
(4)                 ...         return self.s
(4)                 ...
```

```
(4)                 >>> a, b, c, d, e, f = [Foo(x) for x in 'abcdef']
(4)                 >>> grp = Grouper()
(4)                 >>> grp.join(a, b)
(4)                 >>> grp.join(b, c)
(4)                 >>> grp.join(d, e)
(4)                 >>> list(grp)
(4)                 [[a, b, c], [d, e]]
(4)                 >>> grp.joined(a, b)
(4)                 True
(4)                 >>> grp.joined(a, c)
(4)                 True
(4)                 >>> grp.joined(a, d)
(4)                 False
(4)                 """
(4)                 def __init__(self, init=()):
(8)                     self._mapping = weakref.WeakKeyDictionary(
(12)                        {x: weakref.WeakSet([x]) for x in init})
(8)                     self._ordering = weakref.WeakKeyDictionary()
(8)                     for x in init:
(12)                        if x not in self._ordering:
(16)                            self._ordering[x] = len(self._ordering)
(8)                     self._next_order = len(self._ordering)  # Plain int to simplify
pickling.
(4)                 def __getstate__(self):
(8)                     return {
(12)                        **vars(self),
(12)                        "_mapping": {k: set(v) for k, v in self._mapping.items()},
(12)                        "_ordering": {**self._ordering},
(8)                     }
(4)                 def __setstate__(self, state):
(8)                     vars(self).update(state)
(8)                     self._mapping = weakref.WeakKeyDictionary(
(12)                        {k: weakref.WeakSet(v) for k, v in self._mapping.items()})
(8)                     self._ordering = weakref.WeakKeyDictionary(self._ordering)
(4)                 def __contains__(self, item):
(8)                     return item in self._mapping
(4)                 @_api.deprecated("3.8", alternative="none, you no longer need to clean a
Grouper")
(4)                 def clean(self):
(8)                     """Clean dead weak references from the dictionary."""
(4)                 def join(self, a, *args):
(8)                     """
(8)                     Join given arguments into the same set.  Accepts one or more
arguments.
(8)                     """
(8)                     mapping = self._mapping
(8)                     try:
(12)                        set_a = mapping[a]
(8)                     except KeyError:
(12)                        set_a = mapping[a] = weakref.WeakSet([a])
(12)                        self._ordering[a] = self._next_order
(12)                        self._next_order += 1
(8)                     for arg in args:
(12)                        try:
(16)                            set_b = mapping[arg]
(12)                        except KeyError:
(16)                            set_b = mapping[arg] = weakref.WeakSet([arg])
(16)                            self._ordering[arg] = self._next_order
(16)                            self._next_order += 1
(12)                        if set_b is not set_a:
(16)                            if len(set_b) > len(set_a):
(20)                                set_a, set_b = set_b, set_a
(16)                            set_a.update(set_b)
(16)                            for elem in set_b:
(20)                                mapping[elem] = set_a
(4)                 def joined(self, a, b):
(8)                     """Return whether *a* and *b* are members of the same set."""
(8)                     return (self._mapping.get(a, object()) is self._mapping.get(b))
(4)                 def remove(self, a):
```

```
727: (8)                    """Remove *a* from the grouper, doing nothing if it is not there."""
728: (8)                    self._mapping.pop(a, {a}).remove(a)
729: (8)                    self._ordering.pop(a, None)
730: (4)                def __iter__(self):
731: (8)                    """
732: (8)                    Iterate over each of the disjoint sets as a list.
733: (8)                    The iterator is invalid if interleaved with calls to join().
734: (8)                    """
735: (8)                    unique_groups = {id(group): group for group in self._mapping.values()}
736: (8)                    for group in unique_groups.values():
737: (12)                       yield sorted(group, key=self._ordering.__getitem__)
738: (4)                def get_siblings(self, a):
739: (8)                    """Return all of the items joined with *a*, including itself."""
740: (8)                    siblings = self._mapping.get(a, [a])
741: (8)                    return sorted(siblings, key=self._ordering.get)
742: (0)            class GrouperView:
743: (4)                """Immutable view over a `.Grouper`."""
744: (4)                def __init__(self, grouper): self._grouper = grouper
745: (4)                def __contains__(self, item): return item in self._grouper
746: (4)                def __iter__(self): return iter(self._grouper)
747: (4)                def joined(self, a, b): return self._grouper.joined(a, b)
748: (4)                def get_siblings(self, a): return self._grouper.get_siblings(a)
749: (0)            def simple_linear_interpolation(a, steps):
750: (4)                """
751: (4)                Resample an array with ``steps - 1`` points between original point pairs.
752: (4)                Along each column of *a*, ``(steps - 1)`` points are introduced between
753: (4)                each original values; the values are linearly interpolated.
754: (4)                Parameters
755: (4)                ----------
756: (4)                a : array, shape (n, ...)
757: (4)                steps : int
758: (4)                Returns
759: (4)                -------
760: (4)                array
761: (8)                    shape ``((n - 1) * steps + 1, ...)``
762: (4)                """
763: (4)                fps = a.reshape((len(a), -1))
764: (4)                xp = np.arange(len(a)) * steps
765: (4)                x = np.arange((len(a) - 1) * steps + 1)
766: (4)                return (np.column_stack([np.interp(x, xp, fp) for fp in fps.T])
767: (12)                       .reshape((len(x),) + a.shape[1:]))
768: (0)            def delete_masked_points(*args):
769: (4)                """
770: (4)                Find all masked and/or non-finite points in a set of arguments,
771: (4)                and return the arguments with only the unmasked points remaining.
772: (4)                Arguments can be in any of 5 categories:
773: (4)                1) 1-D masked arrays
774: (4)                2) 1-D ndarrays
775: (4)                3) ndarrays with more than one dimension
776: (4)                4) other non-string iterables
777: (4)                5) anything else
778: (4)                The first argument must be in one of the first four categories;
779: (4)                any argument with a length differing from that of the first
780: (4)                argument (and hence anything in category 5) then will be
781: (4)                passed through unchanged.
782: (4)                Masks are obtained from all arguments of the correct length
783: (4)                in categories 1, 2, and 4; a point is bad if masked in a masked
784: (4)                array or if it is a nan or inf.  No attempt is made to
785: (4)                extract a mask from categories 2, 3, and 4 if `numpy.isfinite`
786: (4)                does not yield a Boolean array.
787: (4)                All input arguments that are not passed unchanged are returned
788: (4)                as ndarrays after removing the points or rows corresponding to
789: (4)                masks in any of the arguments.
790: (4)                A vastly simpler version of this function was originally
791: (4)                written as a helper for Axes.scatter().
792: (4)                """
793: (4)                if not len(args):
794: (8)                    return ()
795: (4)                if is_scalar_or_string(args[0]):
```

```
(8)                 raise ValueError("First argument must be a sequence")
(4)             nrecs = len(args[0])
(4)             margs = []
(4)             seqlist = [False] * len(args)
(4)             for i, x in enumerate(args):
(8)                 if not isinstance(x, str) and np.iterable(x) and len(x) == nrecs:
(12)                    seqlist[i] = True
(12)                    if isinstance(x, np.ma.MaskedArray):
(16)                        if x.ndim > 1:
(20)                            raise ValueError("Masked arrays must be 1-D")
(12)                    else:
(16)                        x = np.asarray(x)
(8)                 margs.append(x)
(4)             masks = []  # List of masks that are True where good.
(4)             for i, x in enumerate(margs):
(8)                 if seqlist[i]:
(12)                    if x.ndim > 1:
(16)                        continue  # Don't try to get nan locations unless 1-D.
(12)                    if isinstance(x, np.ma.MaskedArray):
(16)                        masks.append(~np.ma.getmaskarray(x))  # invert the mask
(16)                        xd = x.data
(12)                    else:
(16)                        xd = x
(12)                    try:
(16)                        mask = np.isfinite(xd)
(16)                        if isinstance(mask, np.ndarray):
(20)                            masks.append(mask)
(12)                    except Exception:  # Fixme: put in tuple of possible exceptions?
(16)                        pass
(4)             if len(masks):
(8)                 mask = np.logical_and.reduce(masks)
(8)                 igood = mask.nonzero()[0]
(8)                 if len(igood) < nrecs:
(12)                    for i, x in enumerate(margs):
(16)                        if seqlist[i]:
(20)                            margs[i] = x[igood]
(4)             for i, x in enumerate(margs):
(8)                 if seqlist[i] and isinstance(x, np.ma.MaskedArray):
(12)                    margs[i] = x.filled()
(4)             return margs
(0)         def _combine_masks(*args):
(4)             """
(4)             Find all masked and/or non-finite points in a set of arguments,
(4)             and return the arguments as masked arrays with a common mask.
(4)             Arguments can be in any of 5 categories:
(4)             1) 1-D masked arrays
(4)             2) 1-D ndarrays
(4)             3) ndarrays with more than one dimension
(4)             4) other non-string iterables
(4)             5) anything else
(4)             The first argument must be in one of the first four categories;
(4)             any argument with a length differing from that of the first
(4)             argument (and hence anything in category 5) then will be
(4)             passed through unchanged.
(4)             Masks are obtained from all arguments of the correct length
(4)             in categories 1, 2, and 4; a point is bad if masked in a masked
(4)             array or if it is a nan or inf.  No attempt is made to
(4)             extract a mask from categories 2 and 4 if `numpy.isfinite`
(4)             does not yield a Boolean array.  Category 3 is included to
(4)             support RGB or RGBA ndarrays, which are assumed to have only
(4)             valid values and which are passed through unchanged.
(4)             All input arguments that are not passed unchanged are returned
(4)             as masked arrays if any masked points are found, otherwise as
(4)             ndarrays.
(4)             """
(4)             if not len(args):
(8)                 return ()
(4)             if is_scalar_or_string(args[0]):
(8)                 raise ValueError("First argument must be a sequence")
```

```
865: (4)                nrecs = len(args[0])
866: (4)                margs = []  # Output args; some may be modified.
867: (4)                seqlist = [False] * len(args)  # Flags: True if output will be masked.
868: (4)                masks = []    # List of masks.
869: (4)                for i, x in enumerate(args):
870: (8)                    if is_scalar_or_string(x) or len(x) != nrecs:
871: (12)                       margs.append(x)  # Leave it unmodified.
872: (8)                    else:
873: (12)                       if isinstance(x, np.ma.MaskedArray) and x.ndim > 1:
874: (16)                           raise ValueError("Masked arrays must be 1-D")
875: (12)                       try:
876: (16)                           x = np.asanyarray(x)
877: (12)                       except (VisibleDeprecationWarning, ValueError):
878: (16)                           x = np.asanyarray(x, dtype=object)
879: (12)                       if x.ndim == 1:
880: (16)                           x = safe_masked_invalid(x)
881: (16)                           seqlist[i] = True
882: (16)                           if np.ma.is_masked(x):
883: (20)                               masks.append(np.ma.getmaskarray(x))
884: (12)                       margs.append(x)  # Possibly modified.
885: (4)                if len(masks):
886: (8)                    mask = np.logical_or.reduce(masks)
887: (8)                    for i, x in enumerate(margs):
888: (12)                       if seqlist[i]:
889: (16)                           margs[i] = np.ma.array(x, mask=mask)
890: (4)                return margs
891: (0)            def _broadcast_with_masks(*args, compress=False):
892: (4)                """
893: (4)                Broadcast inputs, combining all masked arrays.
894: (4)                Parameters
895: (4)                ----------
896: (4)                *args : array-like
897: (8)                    The inputs to broadcast.
898: (4)                compress : bool, default: False
899: (8)                    Whether to compress the masked arrays. If False, the masked values
900: (8)                    are replaced by NaNs.
901: (4)                Returns
902: (4)                -------
903: (4)                list of array-like
904: (8)                    The broadcasted and masked inputs.
905: (4)                """
906: (4)                masks = [k.mask for k in args if isinstance(k, np.ma.MaskedArray)]
907: (4)                bcast = np.broadcast_arrays(*args, *masks)
908: (4)                inputs = bcast[:len(args)]
909: (4)                masks = bcast[len(args):]
910: (4)                if masks:
911: (8)                    mask = np.logical_or.reduce(masks)
912: (8)                    if compress:
913: (12)                       inputs = [np.ma.array(k, mask=mask).compressed()
914: (22)                                 for k in inputs]
915: (8)                    else:
916: (12)                       inputs = [np.ma.array(k, mask=mask,
dtype=float).filled(np.nan).ravel()
917: (22)                                 for k in inputs]
918: (4)                else:
919: (8)                    inputs = [np.ravel(k) for k in inputs]
920: (4)                return inputs
921: (0)            def boxplot_stats(X, whis=1.5, bootstrap=None, labels=None, autorange=False):
922: (4)                r"""
923: (4)                Return a list of dictionaries of statistics used to draw a series of box
924: (4)                and whisker plots using `~.Axes.bxp`.
925: (4)                Parameters
926: (4)                ----------
927: (4)                X : array-like
928: (8)                    Data that will be represented in the boxplots. Should have 2 or
929: (8)                    fewer dimensions.
930: (4)                whis : float or (float, float), default: 1.5
931: (8)                    The position of the whiskers.
932: (8)                    If a float, the lower whisker is at the lowest datum above
```

```
(8)                     ``Q1 - whis*(Q3-Q1)``, and the upper whisker at the highest datum
below
(8)                     ``Q3 + whis*(Q3-Q1)``, where Q1 and Q3 are the first and third
(8)                     quartiles.  The default value of ``whis = 1.5`` corresponds to Tukey's
(8)                     original definition of boxplots.
(8)                     If a pair of floats, they indicate the percentiles at which to draw
the
(8)                     whiskers (e.g., (5, 95)).  In particular, setting this to (0, 100)
(8)                     results in whiskers covering the whole range of the data.
(8)                     In the edge case where ``Q1 == Q3``, *whis* is automatically set to
(8)                     (0, 100) (cover the whole range of the data) if *autorange* is True.
(8)                     Beyond the whiskers, data are considered outliers and are plotted as
(8)                     individual points.
(4)                 bootstrap : int, optional
(8)                     Number of times the confidence intervals around the median
(8)                     should be bootstrapped (percentile method).
(4)                 labels : list of str, optional
(8)                     Labels for each dataset. Length must be compatible with
(8)                     dimensions of *X*.
(4)                 autorange : bool, optional (False)
(8)                     When `True` and the data are distributed such that the 25th and 75th
(8)                     percentiles are equal, ``whis`` is set to (0, 100) such that the
(8)                     whisker ends are at the minimum and maximum of the data.
(4)                 Returns
(4)                 -------
(4)                 list of dict
(8)                     A list of dictionaries containing the results for each column
(8)                     of data. Keys of each dictionary are the following:
(8)                     ========   ===================================
(8)                     Key        Value Description
(8)                     ========   ===================================
(8)                     label      tick label for the boxplot
(8)                     mean       arithmetic mean value
(8)                     med        50th percentile
(8)                     q1         first quartile (25th percentile)
(8)                     q3         third quartile (75th percentile)
(8)                     iqr        interquartile range
(8)                     cilo       lower notch around the median
(8)                     cihi       upper notch around the median
(8)                     whislo     end of the lower whisker
(8)                     whishi     end of the upper whisker
(8)                     fliers     outliers
(8)                     ========   ===================================
(4)                 Notes
(4)                 -----
(4)                 Non-bootstrapping approach to confidence interval uses Gaussian-based
(4)                 asymptotic approximation:
(4)                 .. math::
(8)                     \mathrm{med} \pm 1.57 \times \frac{\mathrm{iqr}}{\sqrt{N}}
(4)                 General approach from:
(4)                 McGill, R., Tukey, J.W., and Larsen, W.A. (1978) "Variations of
(4)                 Boxplots", The American Statistician, 32:12-16.
(4)                 """
(4)                 def _bootstrap_median(data, N=5000):
(8)                     M = len(data)
(8)                     percentiles = [2.5, 97.5]
(8)                     bs_index = np.random.randint(M, size=(N, M))
(8)                     bsData = data[bs_index]
(8)                     estimate = np.median(bsData, axis=1, overwrite_input=True)
(8)                     CI = np.percentile(estimate, percentiles)
(8)                     return CI
(4)                 def _compute_conf_interval(data, med, iqr, bootstrap):
(8)                     if bootstrap is not None:
(12)                        CI = _bootstrap_median(data, N=bootstrap)
(12)                        notch_min = CI[0]
(12)                        notch_max = CI[1]
(8)                     else:
(12)                        N = len(data)
(12)                        notch_min = med - 1.57 * iqr / np.sqrt(N)
```

```
(12)                    notch_max = med + 1.57 * iqr / np.sqrt(N)
(8)                 return notch_min, notch_max
(4)             bxpstats = []
(4)             X = _reshape_2D(X, "X")
(4)             ncols = len(X)
(4)             if labels is None:
(8)                 labels = itertools.repeat(None)
(4)             elif len(labels) != ncols:
(8)                 raise ValueError("Dimensions of labels and X must be compatible")
(4)             input_whis = whis
(4)             for ii, (x, label) in enumerate(zip(X, labels)):
(8)                 stats = {}
(8)                 if label is not None:
(12)                    stats['label'] = label
(8)                 whis = input_whis
(8)                 bxpstats.append(stats)
(8)                 if len(x) == 0:
(12)                    stats['fliers'] = np.array([])
(12)                    stats['mean'] = np.nan
(12)                    stats['med'] = np.nan
(12)                    stats['q1'] = np.nan
(12)                    stats['q3'] = np.nan
(12)                    stats['iqr'] = np.nan
(12)                    stats['cilo'] = np.nan
(12)                    stats['cihi'] = np.nan
(12)                    stats['whislo'] = np.nan
(12)                    stats['whishi'] = np.nan
(12)                    continue
(8)                 x = np.ma.asarray(x)
(8)                 x = x.data[~x.mask].ravel()
(8)                 stats['mean'] = np.mean(x)
(8)                 q1, med, q3 = np.percentile(x, [25, 50, 75])
(8)                 stats['iqr'] = q3 - q1
(8)                 if stats['iqr'] == 0 and autorange:
(12)                    whis = (0, 100)
(8)                 stats['cilo'], stats['cihi'] = _compute_conf_interval(
(12)                    x, med, stats['iqr'], bootstrap
(8)                 )
(8)                 if np.iterable(whis) and not isinstance(whis, str):
(12)                    loval, hival = np.percentile(x, whis)
(8)                 elif np.isreal(whis):
(12)                    loval = q1 - whis * stats['iqr']
(12)                    hival = q3 + whis * stats['iqr']
(8)                 else:
(12)                    raise ValueError('whis must be a float or list of percentiles')
(8)                 wiskhi = x[x <= hival]
(8)                 if len(wiskhi) == 0 or np.max(wiskhi) < q3:
(12)                    stats['whishi'] = q3
(8)                 else:
(12)                    stats['whishi'] = np.max(wiskhi)
(8)                 wisklo = x[x >= loval]
(8)                 if len(wisklo) == 0 or np.min(wisklo) > q1:
(12)                    stats['whislo'] = q1
(8)                 else:
(12)                    stats['whislo'] = np.min(wisklo)
(8)                 stats['fliers'] = np.concatenate([
(12)                    x[x < stats['whislo']],
(12)                    x[x > stats['whishi']],
(8)                 ])
(8)                 stats['q1'], stats['med'], stats['q3'] = q1, med, q3
(4)             return bxpstats
(0)         ls_mapper = {'-': 'solid', '--': 'dashed', '-.': 'dashdot', ':': 'dotted'}
(0)         ls_mapper_r = {v: k for k, v in ls_mapper.items()}
(0)         def contiguous_regions(mask):
(4)             """
(4)             Return a list of (ind0, ind1) such that ``mask[ind0:ind1].all()`` is
(4)             True and we cover all such regions.
(4)             """
(4)             mask = np.asarray(mask, dtype=bool)
```

```
(4)             if not mask.size:
(8)                 return []
(4)             idx, = np.nonzero(mask[:-1] != mask[1:])
(4)             idx += 1
(4)             idx = idx.tolist()
(4)             if mask[0]:
(8)                 idx = [0] + idx
(4)             if mask[-1]:
(8)                 idx.append(len(mask))
(4)             return list(zip(idx[::2], idx[1::2]))
(0)         def is_math_text(s):
(4)             """
(4)             Return whether the string *s* contains math expressions.
(4)             This is done by checking whether *s* contains an even number of
(4)             non-escaped dollar signs.
(4)             """
(4)             s = str(s)
(4)             dollar_count = s.count(r'$') - s.count(r'\$')
(4)             even_dollars = (dollar_count > 0 and dollar_count % 2 == 0)
(4)             return even_dollars
(0)         def _to_unmasked_float_array(x):
(4)             """
(4)             Convert a sequence to a float array; if input was a masked array, masked
(4)             values are converted to nans.
(4)             """
(4)             if hasattr(x, 'mask'):
(8)                 return np.ma.asarray(x, float).filled(np.nan)
(4)             else:
(8)                 return np.asarray(x, float)
(0)         def _check_1d(x):
(4)             """Convert scalars to 1D arrays; pass-through arrays as is."""
(4)             x = _unpack_to_numpy(x)
(4)             if (not hasattr(x, 'shape') or
(12)                    not hasattr(x, 'ndim') or
(12)                    len(x.shape) < 1):
(8)                 return np.atleast_1d(x)
(4)             else:
(8)                 return x
(0)         def _reshape_2D(X, name):
(4)             """
(4)             Use Fortran ordering to convert ndarrays and lists of iterables to lists
of
(4)             1D arrays.
(4)             Lists of iterables are converted by applying `numpy.asanyarray` to each of
(4)             their elements.  1D ndarrays are returned in a singleton list containing
(4)             them.  2D ndarrays are converted to the list of their *columns*.
(4)             *name* is used to generate the error message for invalid inputs.
(4)             """
(4)             X = _unpack_to_numpy(X)
(4)             if isinstance(X, np.ndarray):
(8)                 X = X.T
(8)                 if len(X) == 0:
(12)                    return [[]]
(8)                 elif X.ndim == 1 and np.ndim(X[0]) == 0:
(12)                    return [X]
(8)                 elif X.ndim in [1, 2]:
(12)                    return [np.reshape(x, -1) for x in X]
(8)                 else:
(12)                    raise ValueError(f'{name} must have 2 or fewer dimensions')
(4)             if len(X) == 0:
(8)                 return [[]]
(4)             result = []
(4)             is_1d = True
(4)             for xi in X:
(8)                 if not isinstance(xi, str):
(12)                    try:
(16)                        iter(xi)
(12)                    except TypeError:
(16)                        pass
```

```
(12)                    else:
(16)                        is_1d = False
(8)                 xi = np.asanyarray(xi)
(8)                 nd = np.ndim(xi)
(8)                 if nd > 1:
(12)                    raise ValueError(f'{name} must have 2 or fewer dimensions')
(8)                 result.append(xi.reshape(-1))
(4)             if is_1d:
(8)                 return [np.reshape(result, -1)]
(4)             else:
(8)                 return result
(0)         def violin_stats(X, method, points=100, quantiles=None):
(4)             """
(4)             Return a list of dictionaries of data which can be used to draw a series
(4)             of violin plots.
(4)             See the ``Returns`` section below to view the required keys of the
(4)             dictionary.
(4)             Users can skip this function and pass a user-defined set of dictionaries
(4)             with the same keys to `~.axes.Axes.violinplot` instead of using Matplotlib
(4)             to do the calculations. See the *Returns* section below for the keys
(4)             that must be present in the dictionaries.
(4)             Parameters
(4)             ----------
(4)             X : array-like
(8)                 Sample data that will be used to produce the gaussian kernel density
(8)                 estimates. Must have 2 or fewer dimensions.
(4)             method : callable
(8)                 The method used to calculate the kernel density estimate for each
(8)                 column of data. When called via ``method(v, coords)``, it should
(8)                 return a vector of the values of the KDE evaluated at the values
(8)                 specified in coords.
(4)             points : int, default: 100
(8)                 Defines the number of points to evaluate each of the gaussian kernel
(8)                 density estimates at.
(4)             quantiles : array-like, default: None
(8)                 Defines (if not None) a list of floats in interval [0, 1] for each
(8)                 column of data, which represents the quantiles that will be rendered
(8)                 for that column of data. Must have 2 or fewer dimensions. 1D array
will
(8)                 be treated as a singleton list containing them.
(4)             Returns
(4)             -------
(4)             list of dict
(8)                 A list of dictionaries containing the results for each column of data.
(8)                 The dictionaries contain at least the following:
(8)                 - coords: A list of scalars containing the coordinates this particular
(10)                  kernel density estimate was evaluated at.
(8)                 - vals: A list of scalars containing the values of the kernel density
(10)                  estimate at each of the coordinates given in *coords*.
(8)                 - mean: The mean value for this column of data.
(8)                 - median: The median value for this column of data.
(8)                 - min: The minimum value for this column of data.
(8)                 - max: The maximum value for this column of data.
(8)                 - quantiles: The quantile values for this column of data.
(4)             """
(4)             vpstats = []
(4)             X = _reshape_2D(X, "X")
(4)             if quantiles is not None and len(quantiles) != 0:
(8)                 quantiles = _reshape_2D(quantiles, "quantiles")
(4)             else:
(8)                 quantiles = [[]] * len(X)
(4)             if len(X) != len(quantiles):
(8)                 raise ValueError("List of violinplot statistics and quantiles values"
(25)                                 " must have the same length")
(4)             for (x, q) in zip(X, quantiles):
(8)                 stats = {}
(8)                 min_val = np.min(x)
(8)                 max_val = np.max(x)
(8)                 quantile_val = np.percentile(x, 100 * q)
```

```
(8)                 coords = np.linspace(min_val, max_val, points)
(8)                 stats['vals'] = method(x, coords)
(8)                 stats['coords'] = coords
(8)                 stats['mean'] = np.mean(x)
(8)                 stats['median'] = np.median(x)
(8)                 stats['min'] = min_val
(8)                 stats['max'] = max_val
(8)                 stats['quantiles'] = np.atleast_1d(quantile_val)
(8)                 vpstats.append(stats)
(4)             return vpstats
(0)         def pts_to_prestep(x, *args):
(4)             """
(4)             Convert continuous line to pre-steps.
(4)             Given a set of ``N`` points, convert to ``2N - 1`` points, which when
(4)             connected linearly give a step function which changes values at the
(4)             beginning of the intervals.
(4)             Parameters
(4)             ----------
(4)             x : array
(8)                 The x location of the steps. May be empty.
(4)             y1, ..., yp : array
(8)                 y arrays to be turned into steps; all must be the same length as
``x``.
(4)             Returns
(4)             -------
(4)             array
(8)                 The x and y values converted to steps in the same order as the input;
(8)                 can be unpacked as ``x_out, y1_out, ..., yp_out``.  If the input is
(8)                 length ``N``, each of these arrays will be length ``2N + 1``. For
(8)                 ``N=0``, the length will be 0.
(4)             Examples
(4)             --------
(4)             >>> x_s, y1_s, y2_s = pts_to_prestep(x, y1, y2)
(4)             """
(4)             steps = np.zeros((1 + len(args), max(2 * len(x) - 1, 0)))
(4)             steps[0, 0::2] = x
(4)             steps[0, 1::2] = steps[0, 0:-2:2]
(4)             steps[1:, 0::2] = args
(4)             steps[1:, 1::2] = steps[1:, 2::2]
(4)             return steps
(0)         def pts_to_poststep(x, *args):
(4)             """
(4)             Convert continuous line to post-steps.
(4)             Given a set of ``N`` points convert to ``2N + 1`` points, which when
(4)             connected linearly give a step function which changes values at the end of
(4)             the intervals.
(4)             Parameters
(4)             ----------
(4)             x : array
(8)                 The x location of the steps. May be empty.
(4)             y1, ..., yp : array
(8)                 y arrays to be turned into steps; all must be the same length as
``x``.
(4)             Returns
(4)             -------
(4)             array
(8)                 The x and y values converted to steps in the same order as the input;
(8)                 can be unpacked as ``x_out, y1_out, ..., yp_out``.  If the input is
(8)                 length ``N``, each of these arrays will be length ``2N + 1``. For
(8)                 ``N=0``, the length will be 0.
(4)             Examples
(4)             --------
(4)             >>> x_s, y1_s, y2_s = pts_to_poststep(x, y1, y2)
(4)             """
(4)             steps = np.zeros((1 + len(args), max(2 * len(x) - 1, 0)))
(4)             steps[0, 0::2] = x
(4)             steps[0, 1::2] = steps[0, 2::2]
(4)             steps[1:, 0::2] = args
(4)             steps[1:, 1::2] = steps[1:, 0:-2:2]
```

```
(4)             return steps
(0)         def pts_to_midstep(x, *args):
(4)             """
(4)             Convert continuous line to mid-steps.
(4)             Given a set of ``N`` points convert to ``2N`` points which when connected
(4)             linearly give a step function which changes values at the middle of the
(4)             intervals.
(4)             Parameters
(4)             ----------
(4)             x : array
(8)                 The x location of the steps. May be empty.
(4)             y1, ..., yp : array
(8)                 y arrays to be turned into steps; all must be the same length as
(8)                 ``x``.
(4)             Returns
(4)             -------
(4)             array
(8)                 The x and y values converted to steps in the same order as the input;
(8)                 can be unpacked as ``x_out, y1_out, ..., yp_out``.  If the input is
(8)                 length ``N``, each of these arrays will be length ``2N``.
(4)             Examples
(4)             --------
(4)             >>> x_s, y1_s, y2_s = pts_to_midstep(x, y1, y2)
(4)             """
(4)             steps = np.zeros((1 + len(args), 2 * len(x)))
(4)             x = np.asanyarray(x)
(4)             steps[0, 1:-1:2] = steps[0, 2::2] = (x[:-1] + x[1:]) / 2
(4)             steps[0, :1] = x[:1]  # Also works for zero-sized input.
(4)             steps[0, -1:] = x[-1:]
(4)             steps[1:, 0::2] = args
(4)             steps[1:, 1::2] = steps[1:, 0::2]
(4)             return steps
(0)         STEP_LOOKUP_MAP = {'default': lambda x, y: (x, y),
(19)                           'steps': pts_to_prestep,
(19)                           'steps-pre': pts_to_prestep,
(19)                           'steps-post': pts_to_poststep,
(19)                           'steps-mid': pts_to_midstep}
(0)         def index_of(y):
(4)             """
(4)             A helper function to create reasonable x values for the given *y*.
(4)             This is used for plotting (x, y) if x values are not explicitly given.
(4)             First try ``y.index`` (assuming *y* is a `pandas.Series`), if that
(4)             fails, use ``range(len(y))``.
(4)             This will be extended in the future to deal with more types of
(4)             labeled data.
(4)             Parameters
(4)             ----------
(4)             y : float or array-like
(4)             Returns
(4)             -------
(4)             x, y : ndarray
(7)                The x and y values to plot.
(4)             """
(4)             try:
(8)                 return y.index.to_numpy(), y.to_numpy()
(4)             except AttributeError:
(8)                 pass
(4)             try:
(8)                 y = _check_1d(y)
(4)             except (VisibleDeprecationWarning, ValueError):
(8)                 pass
(4)             else:
(8)                 return np.arange(y.shape[0], dtype=float), y
(4)             raise ValueError('Input could not be cast to an at-least-1D NumPy array')
(0)         def safe_first_element(obj):
(4)             """
(4)             Return the first element in *obj*.
(4)             This is a type-independent way of obtaining the first element,
(4)             supporting both index access and the iterator protocol.
```

```
(4)                 """
(4)                 if isinstance(obj, collections.abc.Iterator):
(8)                     try:
(12)                        return obj[0]
(8)                     except TypeError:
(12)                        pass
(8)                     raise RuntimeError("matplotlib does not support generators as input")
(4)                 return next(iter(obj))
(0)             def _safe_first_finite(obj):
(4)                 """
(4)                 Return the first finite element in *obj* if one is available and
skip_nonfinite is
(4)                 True. Otherwise, return the first element.
(4)                 This is a method for internal use.
(4)                 This is a type-independent way of obtaining the first finite element,
supporting
(4)                 both index access and the iterator protocol.
(4)                 """
(4)                 def safe_isfinite(val):
(8)                     if val is None:
(12)                        return False
(8)                     try:
(12)                        return math.isfinite(val)
(8)                     except (TypeError, ValueError):
(12)                        pass
(8)                     try:
(12)                        return np.isfinite(val) if np.isscalar(val) else True
(8)                     except TypeError:
(12)                        return True
(4)                 if isinstance(obj, np.flatiter):
(8)                     return obj[0]
(4)                 elif isinstance(obj, collections.abc.Iterator):
(8)                     raise RuntimeError("matplotlib does not support generators as input")
(4)                 else:
(8)                     for val in obj:
(12)                        if safe_isfinite(val):
(16)                            return val
(8)                     return safe_first_element(obj)
(0)             def sanitize_sequence(data):
(4)                 """
(4)                 Convert dictview objects to list. Other inputs are returned unchanged.
(4)                 """
(4)                 return (list(data) if isinstance(data, collections.abc.MappingView)
(12)                        else data)
(0)             def normalize_kwargs(kw, alias_mapping=None):
(4)                 """
(4)                 Helper function to normalize kwarg inputs.
(4)                 Parameters
(4)                 ----------
(4)                 kw : dict or None
(8)                     A dict of keyword arguments.  None is explicitly supported and treated
(8)                     as an empty dict, to support functions with an optional parameter of
(8)                     the form ``props=None``.
(4)                 alias_mapping : dict or Artist subclass or Artist instance, optional
(8)                     A mapping between a canonical name to a list of aliases, in order of
(8)                     precedence from lowest to highest.
(8)                     If the canonical value is not in the list it is assumed to have the
(8)                     highest priority.
(8)                     If an Artist subclass or instance is passed, use its properties alias
(8)                     mapping.
(4)                 Raises
(4)                 ------
(4)                 TypeError
(8)                     To match what Python raises if invalid arguments/keyword arguments are
(8)                     passed to a callable.
(4)                 """
(4)                 from matplotlib.artist import Artist
(4)                 if kw is None:
(8)                     return {}
```

```
1408: (4)               if alias_mapping is None:
1409: (8)                   alias_mapping = {}
1410: (4)               elif (isinstance(alias_mapping, type) and issubclass(alias_mapping,
Artist)
1411: (10)                    or isinstance(alias_mapping, Artist)):
1412: (8)                   alias_mapping = getattr(alias_mapping, "_alias_map", {})
1413: (4)               to_canonical = {alias: canonical
1414: (20)                              for canonical, alias_list in alias_mapping.items()
1415: (20)                              for alias in alias_list}
1416: (4)               canonical_to_seen = {}
1417: (4)               ret = {}  # output dictionary
1418: (4)               for k, v in kw.items():
1419: (8)                   canonical = to_canonical.get(k, k)
1420: (8)                   if canonical in canonical_to_seen:
1421: (12)                      raise TypeError(f"Got both {canonical_to_seen[canonical]!r} and "
1422: (28)                                      f"{k!r}, which are aliases of one another")
1423: (8)                   canonical_to_seen[canonical] = k
1424: (8)                   ret[canonical] = v
1425: (4)               return ret
1426: (0)           @contextlib.contextmanager
1427: (0)           def _lock_path(path):
1428: (4)               """
1429: (4)               Context manager for locking a path.
1430: (4)               Usage::
1431: (8)                   with _lock_path(path):
1432: (12)                      ...
1433: (4)               Another thread or process that attempts to lock the same path will wait
1434: (4)               until this context manager is exited.
1435: (4)               The lock is implemented by creating a temporary file in the parent
1436: (4)               directory, so that directory must exist and be writable.
1437: (4)               """
1438: (4)               path = Path(path)
1439: (4)               lock_path = path.with_name(path.name + ".matplotlib-lock")
1440: (4)               retries = 50
1441: (4)               sleeptime = 0.1
1442: (4)               for _ in range(retries):
1443: (8)                   try:
1444: (12)                      with lock_path.open("xb"):
1445: (16)                          break
1446: (8)                   except FileExistsError:
1447: (12)                      time.sleep(sleeptime)
1448: (4)               else:
1449: (8)                   raise TimeoutError("""\
1450: (0)           Lock error: Matplotlib failed to acquire the following lock file:
1451: (4)               {}
1452: (0)           This maybe due to another process holding this lock file.  If you are sure no
1453: (0)           other Matplotlib process is running, remove this file and try
again.""".format(
1454: (12)                      lock_path))
1455: (4)               try:
1456: (8)                   yield
1457: (4)               finally:
1458: (8)                   lock_path.unlink()
1459: (0)           def _topmost_artist(
1460: (8)                   artists,
1461: (8)                   _cached_max=functools.partial(max,
key=operator.attrgetter("zorder"))):
1462: (4)               """
1463: (4)               Get the topmost artist of a list.
1464: (4)               In case of a tie, return the *last* of the tied artists, as it will be
1465: (4)               drawn on top of the others. `max` returns the first maximum in case of
1466: (4)               ties, so we need to iterate over the list in reverse order.
1467: (4)               """
1468: (4)               return _cached_max(reversed(artists))
1469: (0)           def _str_equal(obj, s):
1470: (4)               """
1471: (4)               Return whether *obj* is a string equal to string *s*.
1472: (4)               This helper solely exists to handle the case where *obj* is a numpy array,
1473: (4)               because in such cases, a naive ``obj == s`` would yield an array, which
```

```
    cannot be used in a boolean context.
    """
    return isinstance(obj, str) and obj == s
def _str_lower_equal(obj, s):
    """
    Return whether *obj* is a string equal, when lowercased, to string *s*.
    This helper solely exists to handle the case where *obj* is a numpy array,
    because in such cases, a naive ``obj == s`` would yield an array, which
    cannot be used in a boolean context.
    """
    return isinstance(obj, str) and obj.lower() == s
def _array_perimeter(arr):
    """
    Get the elements on the perimeter of *arr*.
    Parameters
    ----------
    arr : ndarray, shape (M, N)
        The input array.
    Returns
    -------
    ndarray, shape (2*(M - 1) + 2*(N - 1),)
        The elements on the perimeter of the array::
           [arr[0, 0], ..., arr[0, -1], ..., arr[-1, -1], ..., arr[-1, 0],
...]
    Examples
    --------
    >>> i, j = np.ogrid[:3, :4]
    >>> a = i*10 + j
    >>> a
    array([[ 0,  1,  2,  3],
           [10, 11, 12, 13],
           [20, 21, 22, 23]])
    >>> _array_perimeter(a)
    array([ 0,  1,  2,  3, 13, 23, 22, 21, 20, 10])
    """
    forward = np.s_[0:-1]      # [0 ... -1)
    backward = np.s_[-1:0:-1]  # [-1 ... 0)
    return np.concatenate((
        arr[0, forward],
        arr[forward, -1],
        arr[-1, backward],
        arr[backward, 0],
    ))
def _unfold(arr, axis, size, step):
    """
    Append an extra dimension containing sliding windows along *axis*.
    All windows are of size *size* and begin with every *step* elements.
    Parameters
    ----------
    arr : ndarray, shape (N_1, ..., N_k)
        The input array
    axis : int
        Axis along which the windows are extracted
    size : int
        Size of the windows
    step : int
        Stride between first elements of subsequent windows.
    Returns
    -------
    ndarray, shape (N_1, ..., 1 + (N_axis-size)/step, ..., N_k, size)
    Examples
    --------
    >>> i, j = np.ogrid[:3, :7]
    >>> a = i*10 + j
    >>> a
    array([[ 0,  1,  2,  3,  4,  5,  6],
           [10, 11, 12, 13, 14, 15, 16],
           [20, 21, 22, 23, 24, 25, 26]])
    >>> _unfold(a, axis=1, size=3, step=2)
```

```
(4)             array([[[ 0,  1,  2],
(12)                    [ 2,  3,  4],
(12)                    [ 4,  5,  6]],
(11)                   [[10, 11, 12],
(12)                    [12, 13, 14],
(12)                    [14, 15, 16]],
(11)                   [[20, 21, 22],
(12)                    [22, 23, 24],
(12)                    [24, 25, 26]]])
(4)             """
(4)             new_shape = [*arr.shape, size]
(4)             new_strides = [*arr.strides, arr.strides[axis]]
(4)             new_shape[axis] = (new_shape[axis] - size) // step + 1
(4)             new_strides[axis] = new_strides[axis] * step
(4)             return np.lib.stride_tricks.as_strided(arr,
(43)                                                   shape=new_shape,
(43)                                                   strides=new_strides,
(43)                                                   writeable=False)
(0)         def _array_patch_perimeters(x, rstride, cstride):
(4)             """
(4)             Extract perimeters of patches from *arr*.
(4)             Extracted patches are of size (*rstride* + 1) x (*cstride* + 1) and
(4)             share perimeters with their neighbors. The ordering of the vertices
matches
(4)             that returned by ``_array_perimeter``.
(4)             Parameters
(4)             ----------
(4)             x : ndarray, shape (N, M)
(8)                 Input array
(4)             rstride : int
(8)                 Vertical (row) stride between corresponding elements of each patch
(4)             cstride : int
(8)                 Horizontal (column) stride between corresponding elements of each
patch
(4)             Returns
(4)             -------
(4)             ndarray, shape (N/rstride * M/cstride, 2 * (rstride + cstride))
(4)             """
(4)             assert rstride > 0 and cstride > 0
(4)             assert (x.shape[0] - 1) % rstride == 0
(4)             assert (x.shape[1] - 1) % cstride == 0
(4)             top = _unfold(x[:-1:rstride, :-1], 1, cstride, cstride)
(4)             bottom = _unfold(x[rstride::rstride, 1:], 1, cstride, cstride)[..., ::-1]
(4)             right = _unfold(x[:-1, cstride::cstride], 0, rstride, rstride)
(4)             left = _unfold(x[1:, :-1:cstride], 0, rstride, rstride)[..., ::-1]
(4)             return (np.concatenate((top, right, bottom, left), axis=2)
(14)                      .reshape(-1, 2 * (rstride + cstride)))
(0)         @contextlib.contextmanager
(0)         def _setattr_cm(obj, **kwargs):
(4)             """
(4)             Temporarily set some attributes; restore original state at context exit.
(4)             """
(4)             sentinel = object()
(4)             origs = {}
(4)             for attr in kwargs:
(8)                 orig = getattr(obj, attr, sentinel)
(8)                 if attr in obj.__dict__ or orig is sentinel:
(12)                    origs[attr] = orig
(8)                 else:
(12)                    cls_orig = getattr(type(obj), attr)
(12)                    if isinstance(cls_orig, property):
(16)                        origs[attr] = orig
(12)                    else:
(16)                        origs[attr] = sentinel
(4)             try:
(8)                 for attr, val in kwargs.items():
(12)                    setattr(obj, attr, val)
(8)                 yield
(4)             finally:
```

```
(8)                 for attr, orig in origs.items():
(12)                    if orig is sentinel:
(16)                        delattr(obj, attr)
(12)                    else:
(16)                        setattr(obj, attr, orig)
(0)         class _OrderedSet(collections.abc.MutableSet):
(4)             def __init__(self):
(8)                 self._od = collections.OrderedDict()
(4)             def __contains__(self, key):
(8)                 return key in self._od
(4)             def __iter__(self):
(8)                 return iter(self._od)
(4)             def __len__(self):
(8)                 return len(self._od)
(4)             def add(self, key):
(8)                 self._od.pop(key, None)
(8)                 self._od[key] = None
(4)             def discard(self, key):
(8)                 self._od.pop(key, None)
(0)         def _premultiplied_argb32_to_unmultiplied_rgba8888(buf):
(4)             """
(4)             Convert a premultiplied ARGB32 buffer to an unmultiplied RGBA8888 buffer.
(4)             """
(4)             rgba = np.take(  # .take() ensures C-contiguity of the result.
(8)                 buf,
(8)                 [2, 1, 0, 3] if sys.byteorder == "little" else [1, 2, 3, 0], axis=2)
(4)             rgb = rgba[..., :-1]
(4)             alpha = rgba[..., -1]
(4)             mask = alpha != 0
(4)             for channel in np.rollaxis(rgb, -1):
(8)                 channel[mask] = (
(12)                    (channel[mask].astype(int) * 255 + alpha[mask] // 2)
(12)                    // alpha[mask])
(4)             return rgba
(0)         def _unmultiplied_rgba8888_to_premultiplied_argb32(rgba8888):
(4)             """
(4)             Convert an unmultiplied RGBA8888 buffer to a premultiplied ARGB32 buffer.
(4)             """
(4)             if sys.byteorder == "little":
(8)                 argb32 = np.take(rgba8888, [2, 1, 0, 3], axis=2)
(8)                 rgb24 = argb32[..., :-1]
(8)                 alpha8 = argb32[..., -1:]
(4)             else:
(8)                 argb32 = np.take(rgba8888, [3, 0, 1, 2], axis=2)
(8)                 alpha8 = argb32[..., :1]
(8)                 rgb24 = argb32[..., 1:]
(4)             if alpha8.min() != 0xff:
(8)                 np.multiply(rgb24, alpha8 / 0xff, out=rgb24, casting="unsafe")
(4)             return argb32
(0)         def _get_nonzero_slices(buf):
(4)             """
(4)             Return the bounds of the nonzero region of a 2D array as a pair of slices.
(4)             ``buf[_get_nonzero_slices(buf)]`` is the smallest sub-rectangle in *buf*
(4)             that encloses all non-zero entries in *buf*.  If *buf* is fully zero, then
(4)             ``(slice(0, 0), slice(0, 0))`` is returned.
(4)             """
(4)             x_nz, = buf.any(axis=0).nonzero()
(4)             y_nz, = buf.any(axis=1).nonzero()
(4)             if len(x_nz) and len(y_nz):
(8)                 l, r = x_nz[[0, -1]]
(8)                 b, t = y_nz[[0, -1]]
(8)                 return slice(b, t + 1), slice(l, r + 1)
(4)             else:
(8)                 return slice(0, 0), slice(0, 0)
(0)         def _pformat_subprocess(command):
(4)             """Pretty-format a subprocess command for printing/logging purposes."""
(4)             return (command if isinstance(command, str)
(12)                    else " ".join(shlex.quote(os.fspath(arg)) for arg in command))
(0)         def _check_and_log_subprocess(command, logger, **kwargs):
```

```
1678: (4)                """
1679: (4)                Run *command*, returning its stdout output if it succeeds.
1680: (4)                If it fails (exits with nonzero return code), raise an exception whose
text
1681: (4)                includes the failed command and captured stdout and stderr output.
1682: (4)                Regardless of the return code, the command is logged at DEBUG level on
1683: (4)                *logger*.  In case of success, the output is likewise logged.
1684: (4)                """
1685: (4)                logger.debug('%s', _pformat_subprocess(command))
1686: (4)                proc = subprocess.run(command, capture_output=True, **kwargs)
1687: (4)                if proc.returncode:
1688: (8)                    stdout = proc.stdout
1689: (8)                    if isinstance(stdout, bytes):
1690: (12)                       stdout = stdout.decode()
1691: (8)                    stderr = proc.stderr
1692: (8)                    if isinstance(stderr, bytes):
1693: (12)                       stderr = stderr.decode()
1694: (8)                    raise RuntimeError(
1695: (12)                       f"The command\n"
1696: (12)                       f"    {_pformat_subprocess(command)}\n"
1697: (12)                       f"failed and generated the following output:\n"
1698: (12)                       f"{stdout}\n"
1699: (12)                       f"and the following error:\n"
1700: (12)                       f"{stderr}")
1701: (4)                if proc.stdout:
1702: (8)                    logger.debug("stdout:\n%s", proc.stdout)
1703: (4)                if proc.stderr:
1704: (8)                    logger.debug("stderr:\n%s", proc.stderr)
1705: (4)                return proc.stdout
1706: (0)            def _setup_new_guiapp():
1707: (4)                """
1708: (4)                Perform OS-dependent setup when Matplotlib creates a new GUI application.
1709: (4)                """
1710: (4)                try:
1711: (8)                    _c_internal_utils.Win32_GetCurrentProcessExplicitAppUserModelID()
1712: (4)                except OSError:
1713: (8)                    _c_internal_utils.Win32_SetCurrentProcessExplicitAppUserModelID(
1714: (12)                       "matplotlib")
1715: (0)            def _format_approx(number, precision):
1716: (4)                """
1717: (4)                Format the number with at most the number of decimals given as precision.
1718: (4)                Remove trailing zeros and possibly the decimal point.
1719: (4)                """
1720: (4)                return f'{number:.{precision}f}'.rstrip('0').rstrip('.') or '0'
1721: (0)            def _g_sig_digits(value, delta):
1722: (4)                """
1723: (4)                Return the number of significant digits to %g-format *value*, assuming
that
1724: (4)                it is known with an error of *delta*.
1725: (4)                """
1726: (4)                if delta == 0:
1727: (8)                    if value == 0:
1728: (12)                       return 3
1729: (8)                    delta = abs(np.spacing(value))
1730: (4)                return max(
1731: (8)                    0,
1732: (8)                    (math.floor(math.log10(abs(value))) + 1 if value else 1)
1733: (8)                    - math.floor(math.log10(delta))) if math.isfinite(value) else 0
1734: (0)            def _unikey_or_keysym_to_mplkey(unikey, keysym):
1735: (4)                """
1736: (4)                Convert a Unicode key or X keysym to a Matplotlib key name.
1737: (4)                The Unicode key is checked first; this avoids having to list most
printable
1738: (4)                keysyms such as ``EuroSign``.
1739: (4)                """
1740: (4)                if unikey and unikey.isprintable():
1741: (8)                    return unikey
1742: (4)                key = keysym.lower()
1743: (4)                if key.startswith("kp_"):  # keypad_x (including kp_enter).
```

```
 1744: (8)                   key = key[3:]
 1745: (4)               if key.startswith("page_"):  # page_{up,down}
 1746: (8)                   key = key.replace("page_", "page")
 1747: (4)               if key.endswith(("_l", "_r")):  # alt_l, ctrl_l, shift_l.
 1748: (8)                   key = key[:-2]
 1749: (4)               if sys.platform == "darwin" and key == "meta":
 1750: (8)                   key = "cmd"
 1751: (4)               key = {
 1752: (8)                   "return": "enter",
 1753: (8)                   "prior": "pageup",  # Used by tk.
 1754: (8)                   "next": "pagedown",  # Used by tk.
 1755: (4)               }.get(key, key)
 1756: (4)               return key
 1757: (0)           @functools.cache
 1758: (0)           def _make_class_factory(mixin_class, fmt, attr_name=None):
 1759: (4)               """
 1760: (4)               Return a function that creates picklable classes inheriting from a mixin.
 1761: (4)               After ::
 1762: (8)                   factory = _make_class_factory(FooMixin, fmt, attr_name)
 1763: (8)                   FooAxes = factory(Axes)
 1764: (4)               ``Foo`` is a class that inherits from ``FooMixin`` and ``Axes`` and **is
 1765: (4)               picklable** (picklability is what differentiates this from a plain call to
 1766: (4)               `type`).  Its ``__name__`` is set to ``fmt.format(Axes.__name__)`` and the
 1767: (4)               base class is stored in the ``attr_name`` attribute, if not None.
 1768: (4)               Moreover, the return value of ``factory`` is memoized: calls with the same
 1769: (4)               ``Axes`` class always return the same subclass.
 1770: (4)               """
 1771: (4)               @functools.cache
 1772: (4)               def class_factory(axes_class):
 1773: (8)                   if issubclass(axes_class, mixin_class):
 1774: (12)                      return axes_class
 1775: (8)                   base_class = axes_class
 1776: (8)                   class subcls(mixin_class, base_class):
 1777: (12)                      __module__ = mixin_class.__module__
 1778: (12)                      def __reduce__(self):
 1779: (16)                          return (_picklable_class_constructor,
 1780: (24)                                  (mixin_class, fmt, attr_name, base_class),
 1781: (24)                                  self.__getstate__())
 1782: (8)                   subcls.__name__ = subcls.__qualname__ =
fmt.format(base_class.__name__)
 1783: (8)                   if attr_name is not None:
 1784: (12)                      setattr(subcls, attr_name, base_class)
 1785: (8)                   return subcls
 1786: (4)               class_factory.__module__ = mixin_class.__module__
 1787: (4)               return class_factory
 1788: (0)           def _picklable_class_constructor(mixin_class, fmt, attr_name, base_class):
 1789: (4)               """Internal helper for _make_class_factory."""
 1790: (4)               factory = _make_class_factory(mixin_class, fmt, attr_name)
 1791: (4)               cls = factory(base_class)
 1792: (4)               return cls.__new__(cls)
 1793: (0)           def _is_torch_array(x):
 1794: (4)               """Check if 'x' is a PyTorch Tensor."""
 1795: (4)               try:
 1796: (8)                   return isinstance(x, sys.modules['torch'].Tensor)
 1797: (4)               except Exception:  # TypeError, KeyError, AttributeError, maybe others?
 1798: (8)                   return False
 1799: (0)           def _is_jax_array(x):
 1800: (4)               """Check if 'x' is a JAX Array."""
 1801: (4)               try:
 1802: (8)                   return isinstance(x, sys.modules['jax'].Array)
 1803: (4)               except Exception:  # TypeError, KeyError, AttributeError, maybe others?
 1804: (8)                   return False
 1805: (0)           def _unpack_to_numpy(x):
 1806: (4)               """Internal helper to extract data from e.g. pandas and xarray objects."""
 1807: (4)               if isinstance(x, np.ndarray):
 1808: (8)                   return x
 1809: (4)               if hasattr(x, 'to_numpy'):
 1810: (8)                   return x.to_numpy()
 1811: (4)               if hasattr(x, 'values'):
```

```
1812: (8)                   xtmp = x.values
1813: (8)                   if isinstance(xtmp, np.ndarray):
1814: (12)                      return xtmp
1815: (4)               if _is_torch_array(x) or _is_jax_array(x):
1816: (8)                   xtmp = x.__array__()
1817: (8)                   if isinstance(xtmp, np.ndarray):
1818: (12)                      return xtmp
1819: (4)               return x
1820: (0)           def _auto_format_str(fmt, value):
1821: (4)               """
1822: (4)               Apply *value* to the format string *fmt*.
1823: (4)               This works both with unnamed %-style formatting and
1824: (4)               unnamed {}-style formatting. %-style formatting has priority.
1825: (4)               If *fmt* is %-style formattable that will be used. Otherwise,
1826: (4)               {}-formatting is applied. Strings without formatting placeholders
1827: (4)               are passed through as is.
1828: (4)               Examples
1829: (4)               --------
1830: (4)               >>> _auto_format_str('%.2f m', 0.2)
1831: (4)               '0.20 m'
1832: (4)               >>> _auto_format_str('{} m', 0.2)
1833: (4)               '0.2 m'
1834: (4)               >>> _auto_format_str('const', 0.2)
1835: (4)               'const'
1836: (4)               >>> _auto_format_str('%d or {}', 0.2)
1837: (4)               '0 or {}'
1838: (4)               """
1839: (4)               try:
1840: (8)                   return fmt % (value,)
1841: (4)               except (TypeError, ValueError):
1842: (8)                   return fmt.format(value)
```

File 30 - cm.py:

```
1: (0)              """
2: (0)              Builtin colormaps, colormap handling utilities, and the `ScalarMappable`
mixin.
3: (0)              .. seealso::
4: (2)                :doc:`/gallery/color/colormap_reference` for a list of builtin colormaps.
5: (2)                :ref:`colormap-manipulation` for examples of how to make
6: (2)                colormaps.
7: (2)                :ref:`colormaps` an in-depth discussion of choosing
8: (2)                colormaps.
9: (2)                :ref:`colormapnorms` for more details about data normalization.
10: (0)             """
11: (0)             from collections.abc import Mapping
12: (0)             import functools
13: (0)             import numpy as np
14: (0)             from numpy import ma
15: (0)             import matplotlib as mpl
16: (0)             from matplotlib import _api, colors, cbook, scale
17: (0)             from matplotlib._cm import datad
18: (0)             from matplotlib._cm_listed import cmaps as cmaps_listed
19: (0)             _LUTSIZE = mpl.rcParams['image.lut']
20: (0)             def _gen_cmap_registry():
21: (4)                 """
22: (4)                 Generate a dict mapping standard colormap names to standard colormaps, as
23: (4)                 well as the reversed colormaps.
24: (4)                 """
25: (4)                 cmap_d = {**cmaps_listed}
26: (4)                 for name, spec in datad.items():
27: (8)                     cmap_d[name] = (  # Precache the cmaps at a fixed lutsize..
28: (12)                        colors.LinearSegmentedColormap(name, spec, _LUTSIZE)
29: (12)                        if 'red' in spec else
30: (12)                        colors.ListedColormap(spec['listed'], name)
31: (12)                        if 'listed' in spec else
32: (12)                        colors.LinearSegmentedColormap.from_list(name, spec, _LUTSIZE))
```

```
33: (4)                 cmap_d['grey'] = cmap_d['gray']
34: (4)                 cmap_d['gist_grey'] = cmap_d['gist_gray']
35: (4)                 cmap_d['gist_yerg'] = cmap_d['gist_yarg']
36: (4)                 cmap_d['Grays'] = cmap_d['Greys']
37: (4)                 for cmap in list(cmap_d.values()):
38: (8)                     rmap = cmap.reversed()
39: (8)                     cmap_d[rmap.name] = rmap
40: (4)                 return cmap_d
41: (0)             class ColormapRegistry(Mapping):
42: (4)                 r"""
43: (4)                 Container for colormaps that are known to Matplotlib by name.
44: (4)                 The universal registry instance is `matplotlib.colormaps`. There should be
45: (4)                 no need for users to instantiate `.ColormapRegistry` themselves.
46: (4)                 Read access uses a dict-like interface mapping names to `.Colormap`\s::
47: (8)                     import matplotlib as mpl
48: (8)                     cmap = mpl.colormaps['viridis']
49: (4)                 Returned `.Colormap`\s are copies, so that their modification does not
50: (4)                 change the global definition of the colormap.
51: (4)                 Additional colormaps can be added via `.ColormapRegistry.register`::
52: (8)                     mpl.colormaps.register(my_colormap)
53: (4)                 To get a list of all registered colormaps, you can do::
54: (8)                     from matplotlib import colormaps
55: (8)                     list(colormaps)
56: (4)                 """
57: (4)                 def __init__(self, cmaps):
58: (8)                     self._cmaps = cmaps
59: (8)                     self._builtin_cmaps = tuple(cmaps)
60: (4)                 def __getitem__(self, item):
61: (8)                     try:
62: (12)                        return self._cmaps[item].copy()
63: (8)                     except KeyError:
64: (12)                        raise KeyError(f"{item!r} is not a known colormap name") from None
65: (4)                 def __iter__(self):
66: (8)                     return iter(self._cmaps)
67: (4)                 def __len__(self):
68: (8)                     return len(self._cmaps)
69: (4)                 def __str__(self):
70: (8)                     return ('ColormapRegistry; available colormaps:\n' +
71: (16)                            ', '.join(f"'{name}'" for name in self))
72: (4)                 def __call__(self):
73: (8)                     """
74: (8)                     Return a list of the registered colormap names.
75: (8)                     This exists only for backward-compatibility in `.pyplot` which had a
76: (8)                     ``plt.colormaps()`` method. The recommended way to get this list is
77: (8)                     now ``list(colormaps)``.
78: (8)                     """
79: (8)                     return list(self)
80: (4)                 def register(self, cmap, *, name=None, force=False):
81: (8)                     """
82: (8)                     Register a new colormap.
83: (8)                     The colormap name can then be used as a string argument to any
``cmap``
84: (8)                     parameter in Matplotlib. It is also available in ``pyplot.get_cmap``.
85: (8)                     The colormap registry stores a copy of the given colormap, so that
86: (8)                     future changes to the original colormap instance do not affect the
87: (8)                     registered colormap. Think of this as the registry taking a snapshot
88: (8)                     of the colormap at registration.
89: (8)                     Parameters
90: (8)                     ----------
91: (8)                     cmap : matplotlib.colors.Colormap
92: (12)                        The colormap to register.
93: (8)                     name : str, optional
94: (12)                        The name for the colormap. If not given, ``cmap.name`` is used.
95: (8)                     force : bool, default: False
96: (12)                        If False, a ValueError is raised if trying to overwrite an already
97: (12)                        registered name. True supports overwriting registered colormaps
98: (12)                        other than the builtin colormaps.
99: (8)                     """
100: (8)                    _api.check_isinstance(colors.Colormap, cmap=cmap)
```

```
101: (8)                    name = name or cmap.name
102: (8)                    if name in self:
103: (12)                       if not force:
104: (16)                           raise ValueError(
105: (20)                               f'A colormap named "{name}" is already registered.')
106: (12)                       elif name in self._builtin_cmaps:
107: (16)                           raise ValueError("Re-registering the builtin cmap "
108: (33)                                             f"{name!r} is not allowed.")
109: (12)                       _api.warn_external(f"Overwriting the cmap {name!r} "
110: (31)                                          "that was already in the registry.")
111: (8)                    self._cmaps[name] = cmap.copy()
112: (8)                    if self._cmaps[name].name != name:
113: (12)                       self._cmaps[name].name = name
114: (4)                def unregister(self, name):
115: (8)                    """
116: (8)                    Remove a colormap from the registry.
117: (8)                    You cannot remove built-in colormaps.
118: (8)                    If the named colormap is not registered, returns with no error, raises
119: (8)                    if you try to de-register a default colormap.
120: (8)                    .. warning::
121: (12)                       Colormap names are currently a shared namespace that may be used
122: (12)                       by multiple packages. Use `unregister` only if you know you
123: (12)                       have registered that name before. In particular, do not
124: (12)                       unregister just in case to clean the name before registering a
125: (12)                       new colormap.
126: (8)                    Parameters
127: (8)                    ----------
128: (8)                    name : str
129: (12)                       The name of the colormap to be removed.
130: (8)                    Raises
131: (8)                    ------
132: (8)                    ValueError
133: (12)                       If you try to remove a default built-in colormap.
134: (8)                    """
135: (8)                    if name in self._builtin_cmaps:
136: (12)                       raise ValueError(f"cannot unregister {name!r} which is a builtin "
137: (29)                                        "colormap.")
138: (8)                    self._cmaps.pop(name, None)
139: (4)                def get_cmap(self, cmap):
140: (8)                    """
141: (8)                    Return a color map specified through *cmap*.
142: (8)                    Parameters
143: (8)                    ----------
144: (8)                    cmap : str or `~matplotlib.colors.Colormap` or None
145: (12)                       - if a `.Colormap`, return it
146: (12)                       - if a string, look it up in ``mpl.colormaps``
147: (12)                       - if None, return the Colormap defined in :rc:`image.cmap`
148: (8)                    Returns
149: (8)                    -------
150: (8)                    Colormap
151: (8)                    """
152: (8)                    if cmap is None:
153: (12)                       return self[mpl.rcParams["image.cmap"]]
154: (8)                    if isinstance(cmap, colors.Colormap):
155: (12)                       return cmap
156: (8)                    if isinstance(cmap, str):
157: (12)                       _api.check_in_list(sorted(_colormaps), cmap=cmap)
158: (12)                       return self[cmap]
159: (8)                    raise TypeError(
160: (12)                       'get_cmap expects None or an instance of a str or Colormap . ' +
161: (12)                       f'you passed {cmap!r} of type {type(cmap)}'
162: (8)                    )
163: (0)            _colormaps = ColormapRegistry(_gen_cmap_registry())
164: (0)            globals().update(_colormaps)
165: (0)            @_api.deprecated(
166: (4)                '3.7',
167: (4)                removal='3.11',
168: (4)                alternative="``matplotlib.colormaps[name]`` or
``matplotlib.colormaps.get_cmap()``"
```

```
(16)                            " or ``pyplot.get_cmap()``"
(4)                 )
(0)             def get_cmap(name=None, lut=None):
(4)                 """
(4)                 Get a colormap instance, defaulting to rc values if *name* is None.
(4)                 Parameters
(4)                 ----------
(4)                 name : `~matplotlib.colors.Colormap` or str or None, default: None
(8)                     If a `.Colormap` instance, it will be returned. Otherwise, the name of
(8)                     a colormap known to Matplotlib, which will be resampled by *lut*. The
(8)                     default, None, means :rc:`image.cmap`.
(4)                 lut : int or None, default: None
(8)                     If *name* is not already a Colormap instance and *lut* is not None,
the
(8)                     colormap will be resampled to have *lut* entries in the lookup table.
(4)                 Returns
(4)                 -------
(4)                 Colormap
(4)                 """
(4)                 if name is None:
(8)                     name = mpl.rcParams['image.cmap']
(4)                 if isinstance(name, colors.Colormap):
(8)                     return name
(4)                 _api.check_in_list(sorted(_colormaps), name=name)
(4)                 if lut is None:
(8)                     return _colormaps[name]
(4)                 else:
(8)                     return _colormaps[name].resampled(lut)
(0)             def _auto_norm_from_scale(scale_cls):
(4)                 """
(4)                 Automatically generate a norm class from *scale_cls*.
(4)                 This differs from `.colors.make_norm_from_scale` in the following points:
(4)                 - This function is not a class decorator, but directly returns a norm
class
(6)                   (as if decorating `.Normalize`).
(4)                 - The scale is automatically constructed with ``nonpositive="mask"``, if
it
(6)                   supports such a parameter, to work around the difference in defaults
(6)                   between standard scales (which use "clip") and norms (which use "mask").
(4)                 Note that ``make_norm_from_scale`` caches the generated norm classes
(4)                 (not the instances) and reuses them for later calls.  For example,
(4)                 ``type(_auto_norm_from_scale("log")) == LogNorm``.
(4)                 """
(4)                 try:
(8)                     norm = colors.make_norm_from_scale(
(12)                        functools.partial(scale_cls, nonpositive="mask"))(
(12)                        colors.Normalize)()
(4)                 except TypeError:
(8)                     norm = colors.make_norm_from_scale(scale_cls)(
(12)                        colors.Normalize)()
(4)                 return type(norm)
(0)             class ScalarMappable:
(4)                 """
(4)                 A mixin class to map scalar data to RGBA.
(4)                 The ScalarMappable applies data normalization before returning RGBA colors
(4)                 from the given colormap.
(4)                 """
(4)                 def __init__(self, norm=None, cmap=None):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     norm : `.Normalize` (or subclass thereof) or str or None
(12)                        The normalizing object which scales data, typically into the
(12)                        interval ``[0, 1]``.
(12)                        If a `str`, a `.Normalize` subclass is dynamically generated based
(12)                        on the scale with the corresponding name.
(12)                        If *None*, *norm* defaults to a *colors.Normalize* object which
(12)                        initializes its scaling based on the first data processed.
(8)                     cmap : str or `~matplotlib.colors.Colormap`
```

```
(12)                        The colormap used to map normalized data values to RGBA colors.
(8)                     """
(8)                     self._A = None
(8)                     self._norm = None  # So that the setter knows we're initializing.
(8)                     self.set_norm(norm)  # The Normalize instance of this ScalarMappable.
(8)                     self.cmap = None  # So that the setter knows we're initializing.
(8)                     self.set_cmap(cmap)  # The Colormap instance of this ScalarMappable.
(8)                     self.colorbar = None
(8)                     self.callbacks = cbook.CallbackRegistry(signals=["changed"])
(4)                 def _scale_norm(self, norm, vmin, vmax):
(8)                     """
(8)                     Helper for initial scaling.
(8)                     Used by public functions that create a ScalarMappable and support
(8)                     parameters *vmin*, *vmax* and *norm*. This makes sure that a *norm*
(8)                     will take precedence over *vmin*, *vmax*.
(8)                     Note that this method does not set the norm.
(8)                     """
(8)                     if vmin is not None or vmax is not None:
(12)                        self.set_clim(vmin, vmax)
(12)                        if isinstance(norm, colors.Normalize):
(16)                            raise ValueError(
(20)                                "Passing a Normalize instance simultaneously with "
(20)                                "vmin/vmax is not supported.  Please pass vmin/vmax "
(20)                                "directly to the norm when creating it.")
(8)                     self.autoscale_None()
(4)                 def to_rgba(self, x, alpha=None, bytes=False, norm=True):
(8)                     """
(8)                     Return a normalized RGBA array corresponding to *x*.
(8)                     In the normal case, *x* is a 1D or 2D sequence of scalars, and
(8)                     the corresponding `~numpy.ndarray` of RGBA values will be returned,
(8)                     based on the norm and colormap set for this ScalarMappable.
(8)                     There is one special case, for handling images that are already
(8)                     RGB or RGBA, such as might have been read from an image file.
(8)                     If *x* is an `~numpy.ndarray` with 3 dimensions,
(8)                     and the last dimension is either 3 or 4, then it will be
(8)                     treated as an RGB or RGBA array, and no mapping will be done.
(8)                     The array can be `~numpy.uint8`, or it can be floats with
(8)                     values in the 0-1 range; otherwise a ValueError will be raised.
(8)                     Any NaNs or masked elements will be set to 0 alpha.
(8)                     If the last dimension is 3, the *alpha* kwarg (defaulting to 1)
(8)                     will be used to fill in the transparency.  If the last dimension
(8)                     is 4, the *alpha* kwarg is ignored; it does not
(8)                     replace the preexisting alpha.  A ValueError will be raised
(8)                     if the third dimension is other than 3 or 4.
(8)                     In either case, if *bytes* is *False* (default), the RGBA
(8)                     array will be floats in the 0-1 range; if it is *True*,
(8)                     the returned RGBA array will be `~numpy.uint8` in the 0 to 255 range.
(8)                     If norm is False, no normalization of the input data is
(8)                     performed, and it is assumed to be in the range (0-1).
(8)                     """
(8)                     try:
(12)                        if x.ndim == 3:
(16)                            if x.shape[2] == 3:
(20)                                if alpha is None:
(24)                                    alpha = 1
(20)                                if x.dtype == np.uint8:
(24)                                    alpha = np.uint8(alpha * 255)
(20)                                m, n = x.shape[:2]
(20)                                xx = np.empty(shape=(m, n, 4), dtype=x.dtype)
(20)                                xx[:, :, :3] = x
(20)                                xx[:, :, 3] = alpha
(16)                            elif x.shape[2] == 4:
(20)                                xx = x
(16)                            else:
(20)                                raise ValueError("Third dimension must be 3 or 4")
(16)                            if xx.dtype.kind == 'f':
(20)                                if np.any(nans := np.isnan(x)):
(24)                                    if x.shape[2] == 4:
(28)                                        xx = xx.copy()
```

```
(24)                                xx[np.any(nans, axis=2), :] = 0
(20)                            if norm and (xx.max() > 1 or xx.min() < 0):
(24)                                raise ValueError("Floating point image RGB values "
(41)                                                 "must be in the 0..1 range.")
(20)                            if bytes:
(24)                                xx = (xx * 255).astype(np.uint8)
(16)                        elif xx.dtype == np.uint8:
(20)                            if not bytes:
(24)                                xx = xx.astype(np.float32) / 255
(16)                        else:
(20)                            raise ValueError("Image RGB array must be uint8 or "
(37)                                             "floating point; found %s" % xx.dtype)
(16)                        if np.ma.is_masked(x):
(20)                            xx[np.any(np.ma.getmaskarray(x), axis=2), 3] = 0
(16)                        return xx
(8)                 except AttributeError:
(12)                    pass
(8)                 x = ma.asarray(x)
(8)                 if norm:
(12)                    x = self.norm(x)
(8)                 rgba = self.cmap(x, alpha=alpha, bytes=bytes)
(8)                 return rgba
(4)             def set_array(self, A):
(8)                 """
(8)                 Set the value array from array-like *A*.
(8)                 Parameters
(8)                 ----------
(8)                 A : array-like or None
(12)                    The values that are mapped to colors.
(12)                    The base class `.ScalarMappable` does not make any assumptions on
(12)                    the dimensionality and shape of the value array *A*.
(8)                 """
(8)                 if A is None:
(12)                    self._A = None
(12)                    return
(8)                 A = cbook.safe_masked_invalid(A, copy=True)
(8)                 if not np.can_cast(A.dtype, float, "same_kind"):
(12)                    raise TypeError(f"Image data of dtype {A.dtype} cannot be "
(28)                                    "converted to float")
(8)                 self._A = A
(8)                 if not self.norm.scaled():
(12)                    self.norm.autoscale_None(A)
(4)             def get_array(self):
(8)                 """
(8)                 Return the array of values, that are mapped to colors.
(8)                 The base class `.ScalarMappable` does not make any assumptions on
(8)                 the dimensionality and shape of the array.
(8)                 """
(8)                 return self._A
(4)             def get_cmap(self):
(8)                 """Return the `.Colormap` instance."""
(8)                 return self.cmap
(4)             def get_clim(self):
(8)                 """
(8)                 Return the values (min, max) that are mapped to the colormap limits.
(8)                 """
(8)                 return self.norm.vmin, self.norm.vmax
(4)             def set_clim(self, vmin=None, vmax=None):
(8)                 """
(8)                 Set the norm limits for image scaling.
(8)                 Parameters
(8)                 ----------
(8)                 vmin, vmax : float
(13)                     The limits.
(13)                     The limits may also be passed as a tuple (*vmin*, *vmax*) as a
(13)                     single positional argument.
(13)                     .. ACCEPTS: (vmin: float, vmax: float)
(8)                 """
(8)                 if vmax is None:
```

```
(12)                        try:
(16)                            vmin, vmax = vmin
(12)                        except (TypeError, ValueError):
(16)                            pass
(8)                     if vmin is not None:
(12)                        self.norm.vmin = colors._sanitize_extrema(vmin)
(8)                     if vmax is not None:
(12)                        self.norm.vmax = colors._sanitize_extrema(vmax)
(4)                 def get_alpha(self):
(8)                     """
(8)                     Returns
(8)                     -------
(8)                     float
(12)                        Always returns 1.
(8)                     """
(8)                     return 1.
(4)                 def set_cmap(self, cmap):
(8)                     """
(8)                     Set the colormap for luminance data.
(8)                     Parameters
(8)                     ----------
(8)                     cmap : `.Colormap` or str or None
(8)                     """
(8)                     in_init = self.cmap is None
(8)                     self.cmap = _ensure_cmap(cmap)
(8)                     if not in_init:
(12)                        self.changed()  # Things are not set up properly yet.
(4)                 @property
(4)                 def norm(self):
(8)                     return self._norm
(4)                 @norm.setter
(4)                 def norm(self, norm):
(8)                     _api.check_isinstance((colors.Normalize, str, None), norm=norm)
(8)                     if norm is None:
(12)                        norm = colors.Normalize()
(8)                     elif isinstance(norm, str):
(12)                        try:
(16)                            scale_cls = scale._scale_mapping[norm]
(12)                        except KeyError:
(16)                            raise ValueError(
(20)                                "Invalid norm str name; the following values are "
(20)                                f"supported: {', '.join(scale._scale_mapping)}"
(16)                            ) from None
(12)                        norm = _auto_norm_from_scale(scale_cls)()
(8)                     if norm is self.norm:
(12)                        return
(8)                     in_init = self.norm is None
(8)                     if not in_init:
(12)                        self.norm.callbacks.disconnect(self._id_norm)
(8)                     self._norm = norm
(8)                     self._id_norm = self.norm.callbacks.connect('changed',
(52)                                                            self.changed)
(8)                     if not in_init:
(12)                        self.changed()
(4)                 def set_norm(self, norm):
(8)                     """
(8)                     Set the normalization instance.
(8)                     Parameters
(8)                     ----------
(8)                     norm : `.Normalize` or str or None
(8)                     Notes
(8)                     -----
(8)                     If there are any colorbars using the mappable for this norm, setting
(8)                     the norm of the mappable will reset the norm, locator, and formatters
(8)                     on the colorbar to default.
(8)                     """
(8)                     self.norm = norm
(4)                 def autoscale(self):
(8)                     """
```

```
442: (8)                    Autoscale the scalar limits on the norm instance using the
443: (8)                    current array
444: (8)                    """
445: (8)                    if self._A is None:
446: (12)                       raise TypeError('You must first set_array for mappable')
447: (8)                    self.norm.autoscale(self._A)
448: (4)                def autoscale_None(self):
449: (8)                    """
450: (8)                    Autoscale the scalar limits on the norm instance using the
451: (8)                    current array, changing only limits that are None
452: (8)                    """
453: (8)                    if self._A is None:
454: (12)                       raise TypeError('You must first set_array for mappable')
455: (8)                    self.norm.autoscale_None(self._A)
456: (4)                def changed(self):
457: (8)                    """
458: (8)                    Call this whenever the mappable is changed to notify all the
459: (8)                    callbackSM listeners to the 'changed' signal.
460: (8)                    """
461: (8)                    self.callbacks.process('changed', self)
462: (8)                    self.stale = True
463: (0)            mpl._docstring.interpd.update(
464: (4)                cmap_doc="""\
465: (0)            cmap : str or `~matplotlib.colors.Colormap`, default: :rc:`image.cmap`
466: (4)                The Colormap instance or registered colormap name used to map scalar data
467: (4)                to colors.""",
468: (4)                norm_doc="""\
469: (0)            norm : str or `~matplotlib.colors.Normalize`, optional
470: (4)                The normalization method used to scale scalar data to the [0, 1] range
471: (4)                before mapping to colors using *cmap*. By default, a linear scaling is
472: (4)                used, mapping the lowest value to 0 and the highest to 1.
473: (4)                If given, this can be one of the following:
474: (4)                - An instance of `.Normalize` or one of its subclasses
475: (6)                  (see :ref:`colormapnorms`).
476: (4)                - A scale name, i.e. one of "linear", "log", "symlog", "logit", etc.  For
a
477: (6)                  list of available scales, call `matplotlib.scale.get_scale_names()`.
478: (6)                  In that case, a suitable `.Normalize` subclass is dynamically generated
479: (6)                  and instantiated.""",
480: (4)                vmin_vmax_doc="""\
481: (0)            vmin, vmax : float, optional
482: (4)                When using scalar data and no explicit *norm*, *vmin* and *vmax* define
483: (4)                the data range that the colormap covers. By default, the colormap covers
484: (4)                the complete value range of the supplied data. It is an error to use
485: (4)                *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm*
486: (4)                name together with *vmin*/*vmax* is acceptable).""",
487: (0)            )
488: (0)            def _ensure_cmap(cmap):
489: (4)                """
490: (4)                Ensure that we have a `.Colormap` object.
491: (4)                For internal use to preserve type stability of errors.
492: (4)                Parameters
493: (4)                ----------
494: (4)                cmap : None, str, Colormap
495: (8)                    - if a `Colormap`, return it
496: (8)                    - if a string, look it up in mpl.colormaps
497: (8)                    - if None, look up the default color map in mpl.colormaps
498: (4)                Returns
499: (4)                -------
500: (4)                Colormap
501: (4)                """
502: (4)                if isinstance(cmap, colors.Colormap):
503: (8)                    return cmap
504: (4)                cmap_name = cmap if cmap is not None else mpl.rcParams["image.cmap"]
505: (4)                if cmap_name not in _colormaps:
506: (8)                    _api.check_in_list(sorted(_colormaps), cmap=cmap_name)
507: (4)                return mpl.colormaps[cmap_name]
```

File 31 - collections.py:

```
1: (0)              """
2: (0)              Classes for the efficient drawing of large collections of objects that
3: (0)              share most properties, e.g., a large number of line segments or
4: (0)              polygons.
5: (0)              The classes are not meant to be as flexible as their single element
6: (0)              counterparts (e.g., you may not be able to select all line styles) but
7: (0)              they are meant to be fast for common use cases (e.g., a large set of solid
8: (0)              line segments).
9: (0)              """
10: (0)             import itertools
11: (0)             import math
12: (0)             from numbers import Number, Real
13: (0)             import warnings
14: (0)             import numpy as np
15: (0)             import matplotlib as mpl
16: (0)             from . import (_api, _path, artist, cbook, cm, colors as mcolors, _docstring,
17: (15)                           hatch as mhatch, lines as mlines, path as mpath, transforms)
18: (0)             from ._enums import JoinStyle, CapStyle
19: (0)             @_api.define_aliases({
20: (4)                 "antialiased": ["antialiaseds", "aa"],
21: (4)                 "edgecolor": ["edgecolors", "ec"],
22: (4)                 "facecolor": ["facecolors", "fc"],
23: (4)                 "linestyle": ["linestyles", "dashes", "ls"],
24: (4)                 "linewidth": ["linewidths", "lw"],
25: (4)                 "offset_transform": ["transOffset"],
26: (0)             })
27: (0)             class Collection(artist.Artist, cm.ScalarMappable):
28: (4)                 r"""
29: (4)                 Base class for Collections. Must be subclassed to be usable.
30: (4)                 A Collection represents a sequence of `.Patch`\es that can be drawn
31: (4)                 more efficiently together than individually. For example, when a single
32: (4)                 path is being drawn repeatedly at different offsets, the renderer can
33: (4)                 typically execute a ``draw_marker()`` call much more efficiently than a
34: (4)                 series of repeated calls to ``draw_path()`` with the offsets put in
35: (4)                 one-by-one.
36: (4)                 Most properties of a collection can be configured per-element. Therefore,
37: (4)                 Collections have "plural" versions of many of the properties of a `.Patch`
38: (4)                 (e.g. `.Collection.get_paths` instead of `.Patch.get_path`). Exceptions
are
39: (4)                 the *zorder*, *hatch*, *pickradius*, *capstyle* and *joinstyle*
properties,
40: (4)                 which can only be set globally for the whole collection.
41: (4)                 Besides these exceptions, all properties can be specified as single values
42: (4)                 (applying to all elements) or sequences of values. The property of the
43: (4)                 ``i``\th element of the collection is::
44: (6)                   prop[i % len(prop)]
45: (4)                 Each Collection can optionally be used as its own `.ScalarMappable` by
46: (4)                 passing the *norm* and *cmap* parameters to its constructor. If the
47: (4)                 Collection's `.ScalarMappable` matrix ``_A`` has been set (via a call
48: (4)                 to `.Collection.set_array`), then at draw time this internal scalar
49: (4)                 mappable will be used to set the ``facecolors`` and ``edgecolors``,
50: (4)                 ignoring those that were manually passed in.
51: (4)                 """
52: (4)                 _transforms = np.empty((0, 3, 3))
53: (4)                 _edge_default = False
54: (4)                 @_docstring.interpd
55: (4)                 def __init__(self, *,
56: (17)                             edgecolors=None,
57: (17)                             facecolors=None,
58: (17)                             linewidths=None,
59: (17)                             linestyles='solid',
60: (17)                             capstyle=None,
61: (17)                             joinstyle=None,
62: (17)                             antialiaseds=None,
63: (17)                             offsets=None,
64: (17)                             offset_transform=None,
```

```
65: (17)                            norm=None,  # optional for ScalarMappable
66: (17)                            cmap=None,  # ditto
67: (17)                            pickradius=5.0,
68: (17)                            hatch=None,
69: (17)                            urls=None,
70: (17)                            zorder=1,
71: (17)                            **kwargs
72: (17)                            ):
73: (8)                     """
74: (8)                     Parameters
75: (8)                     ----------
76: (8)                     edgecolors : :mpltype:`color` or list of colors, default:
:rc:`patch.edgecolor`
77: (12)                        Edge color for each patch making up the collection. The special
78: (12)                        value 'face' can be passed to make the edgecolor match the
79: (12)                        facecolor.
80: (8)                     facecolors : :mpltype:`color` or list of colors, default:
:rc:`patch.facecolor`
81: (12)                        Face color for each patch making up the collection.
82: (8)                     linewidths : float or list of floats, default: :rc:`patch.linewidth`
83: (12)                        Line width for each patch making up the collection.
84: (8)                     linestyles : str or tuple or list thereof, default: 'solid'
85: (12)                        Valid strings are ['solid', 'dashed', 'dashdot', 'dotted', '-',
86: (12)                        '--', '-.', ':']. Dash tuples should be of the form::
87: (16)                            (offset, onoffseq),
88: (12)                        where *onoffseq* is an even length tuple of on and off ink lengths
89: (12)                        in points. For examples, see
90: (12)                        :doc:`/gallery/lines_bars_and_markers/linestyles`.
91: (8)                     capstyle : `.CapStyle`-like, default: :rc:`patch.capstyle`
92: (12)                        Style to use for capping lines for all paths in the collection.
93: (12)                        Allowed values are %(CapStyle)s.
94: (8)                     joinstyle : `.JoinStyle`-like, default: :rc:`patch.joinstyle`
95: (12)                        Style to use for joining lines for all paths in the collection.
96: (12)                        Allowed values are %(JoinStyle)s.
97: (8)                     antialiaseds : bool or list of bool, default: :rc:`patch.antialiased`
98: (12)                        Whether each patch in the collection should be drawn with
99: (12)                        antialiasing.
100: (8)                    offsets : (float, float) or list thereof, default: (0, 0)
101: (12)                       A vector by which to translate each patch after rendering (default
102: (12)                       is no translation). The translation is performed in screen (pixel)
103: (12)                       coordinates (i.e. after the Artist's transform is applied).
104: (8)                    offset_transform : `~.Transform`, default: `.IdentityTransform`
105: (12)                       A single transform which will be applied to each *offsets* vector
106: (12)                       before it is used.
107: (8)                    cmap, norm
108: (12)                       Data normalization and colormapping parameters. See
109: (12)                       `.ScalarMappable` for a detailed description.
110: (8)                    hatch : str, optional
111: (12)                       Hatching pattern to use in filled paths, if any. Valid strings are
112: (12)                       ['/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*']. See
113: (12)                       :doc:`/gallery/shapes_and_collections/hatch_style_reference` for
114: (12)                       the meaning of each hatch type.
115: (8)                    pickradius : float, default: 5.0
116: (12)                       If ``pickradius <= 0``, then `.Collection.contains` will return
117: (12)                       ``True`` whenever the test point is inside of one of the polygons
118: (12)                       formed by the control points of a Path in the Collection. On the
119: (12)                       other hand, if it is greater than 0, then we instead check if the
120: (12)                       test point is contained in a stroke of width ``2*pickradius``
121: (12)                       following any of the Paths in the Collection.
122: (8)                    urls : list of str, default: None
123: (12)                       A URL for each patch to link to once drawn. Currently only works
124: (12)                       for the SVG backend. See :doc:`/gallery/misc/hyperlinks_sgskip`
for
125: (12)                       examples.
126: (8)                    zorder : float, default: 1
127: (12)                       The drawing order, shared by all Patches in the Collection. See
128: (12)                       :doc:`/gallery/misc/zorder_demo` for all defaults and examples.
129: (8)                    **kwargs
130: (12)                       Remaining keyword arguments will be used to set properties as
```

```
131: (12)                        ``Collection.set_{key}(val)`` for each key-value pair in *kwargs*.
132: (8)                     """
133: (8)                     artist.Artist.__init__(self)
134: (8)                     cm.ScalarMappable.__init__(self, norm, cmap)
135: (8)                     self._us_linestyles = [(0, None)]
136: (8)                     self._linestyles = [(0, None)]
137: (8)                     self._us_lw = [0]
138: (8)                     self._linewidths = [0]
139: (8)                     self._gapcolor = None  # Currently only used by LineCollection.
140: (8)                     self._face_is_mapped = None
141: (8)                     self._edge_is_mapped = None
142: (8)                     self._mapped_colors = None  # calculated in update_scalarmappable
143: (8)                     self._hatch_color = mcolors.to_rgba(mpl.rcParams['hatch.color'])
144: (8)                     self.set_facecolor(facecolors)
145: (8)                     self.set_edgecolor(edgecolors)
146: (8)                     self.set_linewidth(linewidths)
147: (8)                     self.set_linestyle(linestyles)
148: (8)                     self.set_antialiased(antialiaseds)
149: (8)                     self.set_pickradius(pickradius)
150: (8)                     self.set_urls(urls)
151: (8)                     self.set_hatch(hatch)
152: (8)                     self.set_zorder(zorder)
153: (8)                     if capstyle:
154: (12)                        self.set_capstyle(capstyle)
155: (8)                     else:
156: (12)                        self._capstyle = None
157: (8)                     if joinstyle:
158: (12)                        self.set_joinstyle(joinstyle)
159: (8)                     else:
160: (12)                        self._joinstyle = None
161: (8)                     if offsets is not None:
162: (12)                        offsets = np.asanyarray(offsets, float)
163: (12)                        if offsets.shape == (2,):
164: (16)                            offsets = offsets[None, :]
165: (8)                     self._offsets = offsets
166: (8)                     self._offset_transform = offset_transform
167: (8)                     self._path_effects = None
168: (8)                     self._internal_update(kwargs)
169: (8)                     self._paths = None
170: (4)                 def get_paths(self):
171: (8)                     return self._paths
172: (4)                 def set_paths(self, paths):
173: (8)                     self._paths = paths
174: (8)                     self.stale = True
175: (4)                 def get_transforms(self):
176: (8)                     return self._transforms
177: (4)                 def get_offset_transform(self):
178: (8)                     """Return the `.Transform` instance used by this artist offset."""
179: (8)                     if self._offset_transform is None:
180: (12)                        self._offset_transform = transforms.IdentityTransform()
181: (8)                     elif (not isinstance(self._offset_transform, transforms.Transform)
182: (14)                           and hasattr(self._offset_transform, '_as_mpl_transform')):
183: (12)                        self._offset_transform = \
184: (16)                            self._offset_transform._as_mpl_transform(self.axes)
185: (8)                     return self._offset_transform
186: (4)                 def set_offset_transform(self, offset_transform):
187: (8)                     """
188: (8)                     Set the artist offset transform.
189: (8)                     Parameters
190: (8)                     ----------
191: (8)                     offset_transform : `.Transform`
192: (8)                     """
193: (8)                     self._offset_transform = offset_transform
194: (4)                 def get_datalim(self, transData):
195: (8)                     transform = self.get_transform()
196: (8)                     offset_trf = self.get_offset_transform()
197: (8)                     if not (isinstance(offset_trf, transforms.IdentityTransform)
198: (16)                            or offset_trf.contains_branch(transData)):
199: (12)                        return transforms.Bbox.null()
```

```
200: (8)                    paths = self.get_paths()
201: (8)                    if not len(paths):
202: (12)                       return transforms.Bbox.null()
203: (8)                    if not transform.is_affine:
204: (12)                       paths = [transform.transform_path_non_affine(p) for p in paths]
205: (8)                    offsets = self.get_offsets()
206: (8)                    if any(transform.contains_branch_seperately(transData)):
207: (12)                       if isinstance(offsets, np.ma.MaskedArray):
208: (16)                           offsets = offsets.filled(np.nan)
209: (12)                       return mpath.get_path_collection_extents(
210: (16)                           transform.get_affine() - transData, paths,
211: (16)                           self.get_transforms(),
212: (16)                           offset_trf.transform_non_affine(offsets),
213: (16)                           offset_trf.get_affine().frozen())
214: (8)                    if self._offsets is not None:
215: (12)                       offsets = (offset_trf - transData).transform(offsets)
216: (12)                       offsets = np.ma.masked_invalid(offsets)
217: (12)                       if not offsets.mask.all():
218: (16)                           bbox = transforms.Bbox.null()
219: (16)                           bbox.update_from_data_xy(offsets)
220: (16)                           return bbox
221: (8)                    return transforms.Bbox.null()
222: (4)                def get_window_extent(self, renderer=None):
223: (8)                    return self.get_datalim(transforms.IdentityTransform())
224: (4)                def _prepare_points(self):
225: (8)                    transform = self.get_transform()
226: (8)                    offset_trf = self.get_offset_transform()
227: (8)                    offsets = self.get_offsets()
228: (8)                    paths = self.get_paths()
229: (8)                    if self.have_units():
230: (12)                       paths = []
231: (12)                       for path in self.get_paths():
232: (16)                           vertices = path.vertices
233: (16)                           xs, ys = vertices[:, 0], vertices[:, 1]
234: (16)                           xs = self.convert_xunits(xs)
235: (16)                           ys = self.convert_yunits(ys)
236: (16)                           paths.append(mpath.Path(np.column_stack([xs, ys]),
path.codes))
237: (12)                       xs = self.convert_xunits(offsets[:, 0])
238: (12)                       ys = self.convert_yunits(offsets[:, 1])
239: (12)                       offsets = np.ma.column_stack([xs, ys])
240: (8)                    if not transform.is_affine:
241: (12)                       paths = [transform.transform_path_non_affine(path)
242: (21)                                for path in paths]
243: (12)                       transform = transform.get_affine()
244: (8)                    if not offset_trf.is_affine:
245: (12)                       offsets = offset_trf.transform_non_affine(offsets)
246: (12)                       offset_trf = offset_trf.get_affine()
247: (8)                    if isinstance(offsets, np.ma.MaskedArray):
248: (12)                       offsets = offsets.filled(np.nan)
249: (8)                    return transform, offset_trf, offsets, paths
250: (4)                @artist.allow_rasterization
251: (4)                def draw(self, renderer):
252: (8)                    if not self.get_visible():
253: (12)                       return
254: (8)                    renderer.open_group(self.__class__.__name__, self.get_gid())
255: (8)                    self.update_scalarmappable()
256: (8)                    transform, offset_trf, offsets, paths = self._prepare_points()
257: (8)                    gc = renderer.new_gc()
258: (8)                    self._set_gc_clip(gc)
259: (8)                    gc.set_snap(self.get_snap())
260: (8)                    if self._hatch:
261: (12)                       gc.set_hatch(self._hatch)
262: (12)                       gc.set_hatch_color(self._hatch_color)
263: (8)                    if self.get_sketch_params() is not None:
264: (12)                       gc.set_sketch_params(*self.get_sketch_params())
265: (8)                    if self.get_path_effects():
266: (12)                       from matplotlib.patheffects import PathEffectRenderer
267: (12)                       renderer = PathEffectRenderer(self.get_path_effects(), renderer)
```

```
268: (8)                    trans = self.get_transforms()
269: (8)                    facecolors = self.get_facecolor()
270: (8)                    edgecolors = self.get_edgecolor()
271: (8)                    do_single_path_optimization = False
272: (8)                    if (len(paths) == 1 and len(trans) <= 1 and
273: (16)                           len(facecolors) == 1 and len(edgecolors) == 1 and
274: (16)                           len(self._linewidths) == 1 and
275: (16)                           all(ls[1] is None for ls in self._linestyles) and
276: (16)                           len(self._antialiaseds) == 1 and len(self._urls) == 1 and
277: (16)                           self.get_hatch() is None):
278: (12)                        if len(trans):
279: (16)                            combined_transform = transforms.Affine2D(trans[0]) + transform
280: (12)                        else:
281: (16)                            combined_transform = transform
282: (12)                        extents = paths[0].get_extents(combined_transform)
283: (12)                        if (extents.width < self.figure.bbox.width
284: (20)                                and extents.height < self.figure.bbox.height):
285: (16)                            do_single_path_optimization = True
286: (8)                    if self._joinstyle:
287: (12)                        gc.set_joinstyle(self._joinstyle)
288: (8)                    if self._capstyle:
289: (12)                        gc.set_capstyle(self._capstyle)
290: (8)                    if do_single_path_optimization:
291: (12)                        gc.set_foreground(tuple(edgecolors[0]))
292: (12)                        gc.set_linewidth(self._linewidths[0])
293: (12)                        gc.set_dashes(*self._linestyles[0])
294: (12)                        gc.set_antialiased(self._antialiaseds[0])
295: (12)                        gc.set_url(self._urls[0])
296: (12)                        renderer.draw_markers(
297: (16)                            gc, paths[0], combined_transform.frozen(),
298: (16)                            mpath.Path(offsets), offset_trf, tuple(facecolors[0]))
299: (8)                    else:
300: (12)                        if self._gapcolor is not None:
301: (16)                            ipaths, ilinestyles = self._get_inverse_paths_linestyles()
302: (16)                            renderer.draw_path_collection(
303: (20)                                gc, transform.frozen(), ipaths,
304: (20)                                self.get_transforms(), offsets, offset_trf,
305: (20)                                [mcolors.to_rgba("none")], self._gapcolor,
306: (20)                                self._linewidths, ilinestyles,
307: (20)                                self._antialiaseds, self._urls,
308: (20)                                "screen")
309: (12)                        renderer.draw_path_collection(
310: (16)                            gc, transform.frozen(), paths,
311: (16)                            self.get_transforms(), offsets, offset_trf,
312: (16)                            self.get_facecolor(), self.get_edgecolor(),
313: (16)                            self._linewidths, self._linestyles,
314: (16)                            self._antialiaseds, self._urls,
315: (16)                            "screen")  # offset_position, kept for backcompat.
316: (8)                    gc.restore()
317: (8)                    renderer.close_group(self.__class__.__name__)
318: (8)                    self.stale = False
319: (4)                def set_pickradius(self, pickradius):
320: (8)                    """
321: (8)                    Set the pick radius used for containment tests.
322: (8)                    Parameters
323: (8)                    ----------
324: (8)                    pickradius : float
325: (12)                       Pick radius, in points.
326: (8)                    """
327: (8)                    if not isinstance(pickradius, Real):
328: (12)                       raise ValueError(
329: (16)                           f"pickradius must be a real-valued number, not
{pickradius!r}")
330: (8)                    self._pickradius = pickradius
331: (4)                def get_pickradius(self):
332: (8)                    return self._pickradius
333: (4)                def contains(self, mouseevent):
334: (8)                    """
335: (8)                    Test whether the mouse event occurred in the collection.
```

```
(8)                     Returns ``bool, dict(ind=itemlist)``, where every item in itemlist
(8)                     contains the event.
(8)                     """
(8)                     if self._different_canvas(mouseevent) or not self.get_visible():
(12)                        return False, {}
(8)                     pickradius = (
(12)                        float(self._picker)
(12)                        if isinstance(self._picker, Number) and
(15)                           self._picker is not True  # the bool, not just nonzero or 1
(12)                        else self._pickradius)
(8)                     if self.axes:
(12)                        self.axes._unstale_viewLim()
(8)                     transform, offset_trf, offsets, paths = self._prepare_points()
(8)                     ind = _path.point_in_path_collection(
(12)                        mouseevent.x, mouseevent.y, pickradius,
(12)                        transform.frozen(), paths, self.get_transforms(),
(12)                        offsets, offset_trf, pickradius <= 0)
(8)                     return len(ind) > 0, dict(ind=ind)
(4)                 def set_urls(self, urls):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     urls : list of str or None
(8)                     Notes
(8)                     -----
(8)                     URLs are currently only implemented by the SVG backend. They are
(8)                     ignored by all other backends.
(8)                     """
(8)                     self._urls = urls if urls is not None else [None]
(8)                     self.stale = True
(4)                 def get_urls(self):
(8)                     """
(8)                     Return a list of URLs, one for each element of the collection.
(8)                     The list contains *None* for elements without a URL. See
(8)                     :doc:`/gallery/misc/hyperlinks_sgskip` for an example.
(8)                     """
(8)                     return self._urls
(4)                 def set_hatch(self, hatch):
(8)                     r"""
(8)                     Set the hatching pattern
(8)                     *hatch* can be one of::
(10)                      /   - diagonal hatching
(10)                      \   - back diagonal
(10)                      |   - vertical
(10)                      -   - horizontal
(10)                      +   - crossed
(10)                      x   - crossed diagonal
(10)                      o   - small circle
(10)                      O   - large circle
(10)                      .   - dots
(10)                      *   - stars
(8)                     Letters can be combined, in which case all the specified
(8)                     hatchings are done.  If same letter repeats, it increases the
(8)                     density of hatching of that pattern.
(8)                     Unlike other properties such as linewidth and colors, hatching
(8)                     can only be specified for the collection as a whole, not separately
(8)                     for each member.
(8)                     Parameters
(8)                     ----------
(8)                     hatch : {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
(8)                     """
(8)                     mhatch._validate_hatch_pattern(hatch)
(8)                     self._hatch = hatch
(8)                     self.stale = True
(4)                 def get_hatch(self):
(8)                     """Return the current hatching pattern."""
(8)                     return self._hatch
(4)                 def set_offsets(self, offsets):
(8)                     """
```

```
(8)                     Set the offsets for the collection.
(8)                     Parameters
(8)                     ----------
(8)                     offsets : (N, 2) or (2,) array-like
(8)                     """
(8)                     offsets = np.asanyarray(offsets)
(8)                     if offsets.shape == (2,):  # Broadcast (2,) -> (1, 2) but nothing
else.
(12)                        offsets = offsets[None, :]
(8)                     cstack = (np.ma.column_stack if isinstance(offsets, np.ma.MaskedArray)
(18)                              else np.column_stack)
(8)                     self._offsets = cstack(
(12)                        (np.asanyarray(self.convert_xunits(offsets[:, 0]), float),
(13)                         np.asanyarray(self.convert_yunits(offsets[:, 1]), float)))
(8)                     self.stale = True
(4)                 def get_offsets(self):
(8)                     """Return the offsets for the collection."""
(8)                     return np.zeros((1, 2)) if self._offsets is None else self._offsets
(4)                 def _get_default_linewidth(self):
(8)                     return mpl.rcParams['patch.linewidth']  # validated as float
(4)                 def set_linewidth(self, lw):
(8)                     """
(8)                     Set the linewidth(s) for the collection.  *lw* can be a scalar
(8)                     or a sequence; if it is a sequence the patches will cycle
(8)                     through the sequence
(8)                     Parameters
(8)                     ----------
(8)                     lw : float or list of floats
(8)                     """
(8)                     if lw is None:
(12)                        lw = self._get_default_linewidth()
(8)                     self._us_lw = np.atleast_1d(lw)
(8)                     self._linewidths, self._linestyles = self._bcast_lwls(
(12)                        self._us_lw, self._us_linestyles)
(8)                     self.stale = True
(4)                 def set_linestyle(self, ls):
(8)                     """
(8)                     Set the linestyle(s) for the collection.
(8)                     ===========================   =================
(8)                     linestyle                     description
(8)                     ===========================   =================
(8)                     ``'-'`` or ``'solid'``        solid line
(8)                     ``'--'`` or  ``'dashed'``     dashed line
(8)                     ``'-.'`` or  ``'dashdot'``    dash-dotted line
(8)                     ``':'`` or ``'dotted'``       dotted line
(8)                     ===========================   =================
(8)                     Alternatively a dash tuple of the following form can be provided::
(12)                        (offset, onoffseq),
(8)                     where ``onoffseq`` is an even length tuple of on and off ink in
points.
(8)                     Parameters
(8)                     ----------
(8)                     ls : str or tuple or list thereof
(12)                        Valid values for individual linestyles include {'-', '--', '-.',
(12)                        ':', '', (offset, on-off-seq)}. See `.Line2D.set_linestyle` for a
(12)                        complete description.
(8)                     """
(8)                     try:
(12)                        dashes = [mlines._get_dash_pattern(ls)]
(8)                     except ValueError:
(12)                        try:
(16)                            dashes = [mlines._get_dash_pattern(x) for x in ls]
(12)                        except ValueError as err:
(16)                            emsg = f'Do not know how to convert {ls!r} to dashes'
(16)                            raise ValueError(emsg) from err
(8)                     self._us_linestyles = dashes
(8)                     self._linewidths, self._linestyles = self._bcast_lwls(
(12)                        self._us_lw, self._us_linestyles)
(4)                 @_docstring.interpd
```

```
    def set_capstyle(self, cs):
        """
        Set the `.CapStyle` for the collection (for all its elements).
        Parameters
        ----------
        cs : `.CapStyle` or %(CapStyle)s
        """
        self._capstyle = CapStyle(cs)
    @_docstring.interpd
    def get_capstyle(self):
        """
        Return the cap style for the collection (for all its elements).
        Returns
        -------
        %(CapStyle)s or None
        """
        return self._capstyle.name if self._capstyle else None
    @_docstring.interpd
    def set_joinstyle(self, js):
        """
        Set the `.JoinStyle` for the collection (for all its elements).
        Parameters
        ----------
        js : `.JoinStyle` or %(JoinStyle)s
        """
        self._joinstyle = JoinStyle(js)
    @_docstring.interpd
    def get_joinstyle(self):
        """
        Return the join style for the collection (for all its elements).
        Returns
        -------
        %(JoinStyle)s or None
        """
        return self._joinstyle.name if self._joinstyle else None
    @staticmethod
    def _bcast_lwls(linewidths, dashes):
        """
        Internal helper function to broadcast + scale ls/lw
        In the collection drawing code, the linewidth and linestyle are cycled
        through as circular buffers (via ``v[i % len(v)]``).  Thus, if we are
        going to scale the dash pattern at set time (not draw time) we need to
        do the broadcasting now and expand both lists to be the same length.
        Parameters
        ----------
        linewidths : list
            line widths of collection
        dashes : list
            dash specification (offset, (dash pattern tuple))
        Returns
        -------
        linewidths, dashes : list
            Will be the same length, dashes are scaled by paired linewidth
        """
        if mpl.rcParams['_internal.classic_mode']:
            return linewidths, dashes
        if len(dashes) != len(linewidths):
            l_dashes = len(dashes)
            l_lw = len(linewidths)
            gcd = math.gcd(l_dashes, l_lw)
            dashes = list(dashes) * (l_lw // gcd)
            linewidths = list(linewidths) * (l_dashes // gcd)
        dashes = [mlines._scale_dashes(o, d, lw)
                  for (o, d), lw in zip(dashes, linewidths)]
        return linewidths, dashes
    def get_antialiased(self):
        """
        Get the antialiasing state for rendering.
        Returns
```

```
(8)                     -------
(8)                     array of bools
(8)                     """
(8)                     return self._antialiaseds
(4)                 def set_antialiased(self, aa):
(8)                     """
(8)                     Set the antialiasing state for rendering.
(8)                     Parameters
(8)                     ----------
(8)                     aa : bool or list of bools
(8)                     """
(8)                     if aa is None:
(12)                        aa = self._get_default_antialiased()
(8)                     self._antialiaseds = np.atleast_1d(np.asarray(aa, bool))
(8)                     self.stale = True
(4)                 def _get_default_antialiased(self):
(8)                     return mpl.rcParams['patch.antialiased']
(4)                 def set_color(self, c):
(8)                     """
(8)                     Set both the edgecolor and the facecolor.
(8)                     Parameters
(8)                     ----------
(8)                     c : :mpltype:`color` or list of RGBA tuples
(8)                     See Also
(8)                     --------
(8)                     Collection.set_facecolor, Collection.set_edgecolor
(12)                        For setting the edge or face color individually.
(8)                     """
(8)                     self.set_facecolor(c)
(8)                     self.set_edgecolor(c)
(4)                 def _get_default_facecolor(self):
(8)                     return mpl.rcParams['patch.facecolor']
(4)                 def _set_facecolor(self, c):
(8)                     if c is None:
(12)                        c = self._get_default_facecolor()
(8)                     self._facecolors = mcolors.to_rgba_array(c, self._alpha)
(8)                     self.stale = True
(4)                 def set_facecolor(self, c):
(8)                     """
(8)                     Set the facecolor(s) of the collection. *c* can be a color (all
patches
(8)                     have same color), or a sequence of colors; if it is a sequence the
(8)                     patches will cycle through the sequence.
(8)                     If *c* is 'none', the patch will not be filled.
(8)                     Parameters
(8)                     ----------
(8)                     c : :mpltype:`color` or list of :mpltype:`color`
(8)                     """
(8)                     if isinstance(c, str) and c.lower() in ("none", "face"):
(12)                        c = c.lower()
(8)                     self._original_facecolor = c
(8)                     self._set_facecolor(c)
(4)                 def get_facecolor(self):
(8)                     return self._facecolors
(4)                 def get_edgecolor(self):
(8)                     if cbook._str_equal(self._edgecolors, 'face'):
(12)                        return self.get_facecolor()
(8)                     else:
(12)                        return self._edgecolors
(4)                 def _get_default_edgecolor(self):
(8)                     return mpl.rcParams['patch.edgecolor']
(4)                 def _set_edgecolor(self, c):
(8)                     set_hatch_color = True
(8)                     if c is None:
(12)                        if (mpl.rcParams['patch.force_edgecolor']
(20)                                or self._edge_default
(20)                                or cbook._str_equal(self._original_facecolor, 'none')):
(16)                            c = self._get_default_edgecolor()
(12)                        else:
```

```
(16)                            c = 'none'
(16)                            set_hatch_color = False
(8)                         if cbook._str_lower_equal(c, 'face'):
(12)                            self._edgecolors = 'face'
(12)                            self.stale = True
(12)                            return
(8)                         self._edgecolors = mcolors.to_rgba_array(c, self._alpha)
(8)                         if set_hatch_color and len(self._edgecolors):
(12)                            self._hatch_color = tuple(self._edgecolors[0])
(8)                         self.stale = True
(4)                     def set_edgecolor(self, c):
(8)                         """
(8)                         Set the edgecolor(s) of the collection.
(8)                         Parameters
(8)                         ----------
(8)                         c : :mpltype:`color` or list of :mpltype:`color` or 'face'
(12)                            The collection edgecolor(s).  If a sequence, the patches cycle
(12)                            through it.  If 'face', match the facecolor.
(8)                         """
(8)                         if isinstance(c, str) and c.lower() in ("none", "face"):
(12)                            c = c.lower()
(8)                         self._original_edgecolor = c
(8)                         self._set_edgecolor(c)
(4)                     def set_alpha(self, alpha):
(8)                         """
(8)                         Set the transparency of the collection.
(8)                         Parameters
(8)                         ----------
(8)                         alpha : float or array of float or None
(12)                            If not None, *alpha* values must be between 0 and 1, inclusive.
(12)                            If an array is provided, its length must match the number of
(12)                            elements in the collection.  Masked values and nans are not
(12)                            supported.
(8)                         """
(8)                         artist.Artist._set_alpha_for_array(self, alpha)
(8)                         self._set_facecolor(self._original_facecolor)
(8)                         self._set_edgecolor(self._original_edgecolor)
(4)                     set_alpha.__doc__ = artist.Artist._set_alpha_for_array.__doc__
(4)                     def get_linewidth(self):
(8)                         return self._linewidths
(4)                     def get_linestyle(self):
(8)                         return self._linestyles
(4)                     def _set_mappable_flags(self):
(8)                         """
(8)                         Determine whether edges and/or faces are color-mapped.
(8)                         This is a helper for update_scalarmappable.
(8)                         It sets Boolean flags '_edge_is_mapped' and '_face_is_mapped'.
(8)                         Returns
(8)                         -------
(8)                         mapping_change : bool
(12)                            True if either flag is True, or if a flag has changed.
(8)                         """
(8)                         edge0 = self._edge_is_mapped
(8)                         face0 = self._face_is_mapped
(8)                         self._edge_is_mapped = False
(8)                         self._face_is_mapped = False
(8)                         if self._A is not None:
(12)                            if not cbook._str_equal(self._original_facecolor, 'none'):
(16)                                self._face_is_mapped = True
(16)                                if cbook._str_equal(self._original_edgecolor, 'face'):
(20)                                    self._edge_is_mapped = True
(12)                            else:
(16)                                if self._original_edgecolor is None:
(20)                                    self._edge_is_mapped = True
(8)                         mapped = self._face_is_mapped or self._edge_is_mapped
(8)                         changed = (edge0 is None or face0 is None
(19)                                   or self._edge_is_mapped != edge0
(19)                                   or self._face_is_mapped != face0)
(8)                         return mapped or changed
```

```
(4)     def update_scalarmappable(self):
(8)         """
(8)         Update colors from the scalar mappable array, if any.
(8)         Assign colors to edges and faces based on the array and/or
(8)         colors that were directly set, as appropriate.
(8)         """
(8)         if not self._set_mappable_flags():
(12)            return
(8)         if self._A is not None:
(12)            if self._A.ndim > 1 and not isinstance(self, _MeshData):
(16)                raise ValueError('Collections can only map rank 1 arrays')
(12)            if np.iterable(self._alpha):
(16)                if self._alpha.size != self._A.size:
(20)                    raise ValueError(
(24)                        f'Data array shape, {self._A.shape} '
(24)                        'is incompatible with alpha array shape, '
(24)                        f'{self._alpha.shape}. '
(24)                        'This can occur with the deprecated '
(24)                        'behavior of the "flat" shading option, '
(24)                        'in which a row and/or column of the data '
(24)                        'array is dropped.')
(16)                self._alpha = self._alpha.reshape(self._A.shape)
(12)            self._mapped_colors = self.to_rgba(self._A, self._alpha)
(8)         if self._face_is_mapped:
(12)            self._facecolors = self._mapped_colors
(8)         else:
(12)            self._set_facecolor(self._original_facecolor)
(8)         if self._edge_is_mapped:
(12)            self._edgecolors = self._mapped_colors
(8)         else:
(12)            self._set_edgecolor(self._original_edgecolor)
(8)         self.stale = True
(4)     def get_fill(self):
(8)         """Return whether face is colored."""
(8)         return not cbook._str_lower_equal(self._original_facecolor, "none")
(4)     def update_from(self, other):
(8)         """Copy properties from other to self."""
(8)         artist.Artist.update_from(self, other)
(8)         self._antialiaseds = other._antialiaseds
(8)         self._mapped_colors = other._mapped_colors
(8)         self._edge_is_mapped = other._edge_is_mapped
(8)         self._original_edgecolor = other._original_edgecolor
(8)         self._edgecolors = other._edgecolors
(8)         self._face_is_mapped = other._face_is_mapped
(8)         self._original_facecolor = other._original_facecolor
(8)         self._facecolors = other._facecolors
(8)         self._linewidths = other._linewidths
(8)         self._linestyles = other._linestyles
(8)         self._us_linestyles = other._us_linestyles
(8)         self._pickradius = other._pickradius
(8)         self._hatch = other._hatch
(8)         self._A = other._A
(8)         self.norm = other.norm
(8)         self.cmap = other.cmap
(8)         self.stale = True
(0) class _CollectionWithSizes(Collection):
(4)     """
(4)     Base class for collections that have an array of sizes.
(4)     """
(4)     _factor = 1.0
(4)     def get_sizes(self):
(8)         """
(8)         Return the sizes ('areas') of the elements in the collection.
(8)         Returns
(8)         -------
(8)         array
(12)            The 'area' of each element.
(8)         """
(8)         return self._sizes
```

```
    def set_sizes(self, sizes, dpi=72.0):
        """
        Set the sizes of each member of the collection.
        Parameters
        ----------
        sizes : `numpy.ndarray` or None
            The size to set for each element of the collection.  The
            value is the 'area' of the element.
        dpi : float, default: 72
            The dpi of the canvas.
        """
        if sizes is None:
            self._sizes = np.array([])
            self._transforms = np.empty((0, 3, 3))
        else:
            self._sizes = np.asarray(sizes)
            self._transforms = np.zeros((len(self._sizes), 3, 3))
            scale = np.sqrt(self._sizes) * dpi / 72.0 * self._factor
            self._transforms[:, 0, 0] = scale
            self._transforms[:, 1, 1] = scale
            self._transforms[:, 2, 2] = 1.0
        self.stale = True
    @artist.allow_rasterization
    def draw(self, renderer):
        self.set_sizes(self._sizes, self.figure.dpi)
        super().draw(renderer)
class PathCollection(_CollectionWithSizes):
    r"""
    A collection of `~.path.Path`\s, as created by e.g. `~.Axes.scatter`.
    """
    def __init__(self, paths, sizes=None, **kwargs):
        """
        Parameters
        ----------
        paths : list of `.path.Path`
            The paths that will make up the `.Collection`.
        sizes : array-like
            The factor by which to scale each drawn `~.path.Path`. One unit
            squared in the Path's data space is scaled to be ``sizes**2``
            points when rendered.
        **kwargs
            Forwarded to `.Collection`.
        """
        super().__init__(**kwargs)
        self.set_paths(paths)
        self.set_sizes(sizes)
        self.stale = True
    def get_paths(self):
        return self._paths
    def legend_elements(self, prop="colors", num="auto",
                        fmt=None, func=lambda x: x, **kwargs):
        """
        Create legend handles and labels for a PathCollection.
        Each legend handle is a `.Line2D` representing the Path that was drawn,
        and each label is a string that represents the Path.
        This is useful for obtaining a legend for a `~.Axes.scatter` plot;
        e.g.::
            scatter = plt.scatter([1, 2, 3],  [4, 5, 6],  c=[7, 2, 3], num=None)
            plt.legend(*scatter.legend_elements())
        creates three legend elements, one for each color with the numerical
        values passed to *c* as the labels.
        Also see the :ref:`automatedlegendcreation` example.
        Parameters
        ----------
        prop : {"colors", "sizes"}, default: "colors"
            If "colors", the legend handles will show the different colors of
            the collection. If "sizes", the legend will show the different
```

```
(12)                        sizes. To set both, use *kwargs* to directly edit the `.Line2D`
(12)                        properties.
(8)                     num : int, None, "auto" (default), array-like, or `~.ticker.Locator`
(12)                        Target number of elements to create.
(12)                        If None, use all unique elements of the mappable array. If an
(12)                        integer, target to use *num* elements in the normed range.
(12)                        If *"auto"*, try to determine which option better suits the nature
(12)                        of the data.
(12)                        The number of created elements may slightly deviate from *num* due
(12)                        to a `~.ticker.Locator` being used to find useful locations.
(12)                        If a list or array, use exactly those elements for the legend.
(12)                        Finally, a `~.ticker.Locator` can be provided.
(8)                     fmt : str, `~matplotlib.ticker.Formatter`, or None (default)
(12)                        The format or formatter to use for the labels. If a string must be
(12)                        a valid input for a `.StrMethodFormatter`. If None (the default),
(12)                        use a `.ScalarFormatter`.
(8)                     func : function, default: ``lambda x: x``
(12)                        Function to calculate the labels.  Often the size (or color)
(12)                        argument to `~.Axes.scatter` will have been pre-processed by the
(12)                        user using a function ``s = f(x)`` to make the markers visible;
(12)                        e.g. ``size = np.log10(x)``.  Providing the inverse of this
(12)                        function here allows that pre-processing to be inverted, so that
(12)                        the legend labels have the correct values; e.g. ``func = lambda
(12)                        x: 10**x``.
(8)                     **kwargs
(12)                        Allowed keyword arguments are *color* and *size*. E.g. it may be
(12)                        useful to set the color of the markers if *prop="sizes"* is used;
(12)                        similarly to set the size of the markers if *prop="colors"* is
(12)                        used. Any further parameters are passed onto the `.Line2D`
(12)                        instance. This may be useful to e.g. specify a different
(12)                        *markeredgecolor* or *alpha* for the legend handles.
(8)                     Returns
(8)                     -------
(8)                     handles : list of `.Line2D`
(12)                        Visual representation of each element of the legend.
(8)                     labels : list of str
(12)                        The string labels for elements of the legend.
(8)                     """
(8)                     handles = []
(8)                     labels = []
(8)                     hasarray = self.get_array() is not None
(8)                     if fmt is None:
(12)                        fmt = mpl.ticker.ScalarFormatter(useOffset=False,
useMathText=True)
(8)                     elif isinstance(fmt, str):
(12)                        fmt = mpl.ticker.StrMethodFormatter(fmt)
(8)                     fmt.create_dummy_axis()
(8)                     if prop == "colors":
(12)                        if not hasarray:
(16)                            warnings.warn("Collection without array used. Make sure to "
(30)                                          "specify the values to be colormapped via the "
(30)                                          "`c` argument.")
(16)                            return handles, labels
(12)                        u = np.unique(self.get_array())
(12)                        size = kwargs.pop("size", mpl.rcParams["lines.markersize"])
(8)                     elif prop == "sizes":
(12)                        u = np.unique(self.get_sizes())
(12)                        color = kwargs.pop("color", "k")
(8)                     else:
(12)                        raise ValueError("Valid values for `prop` are 'colors' or "
(29)                                         f"'sizes'. You supplied '{prop}' instead.")
(8)                     fu = func(u)
(8)                     fmt.axis.set_view_interval(fu.min(), fu.max())
(8)                     fmt.axis.set_data_interval(fu.min(), fu.max())
(8)                     if num == "auto":
(12)                        num = 9
(12)                        if len(u) <= num:
(16)                            num = None
(8)                     if num is None:
```

```
(12)                        values = u
(12)                        label_values = func(values)
(8)                     else:
(12)                        if prop == "colors":
(16)                            arr = self.get_array()
(12)                        elif prop == "sizes":
(16)                            arr = self.get_sizes()
(12)                        if isinstance(num, mpl.ticker.Locator):
(16)                            loc = num
(12)                        elif np.iterable(num):
(16)                            loc = mpl.ticker.FixedLocator(num)
(12)                        else:
(16)                            num = int(num)
(16)                            loc = mpl.ticker.MaxNLocator(nbins=num, min_n_ticks=num-1,
(45)                                                         steps=[1, 2, 2.5, 3, 5, 6, 8,
10])
(12)                        label_values = loc.tick_values(func(arr).min(), func(arr).max())
(12)                        cond = ((label_values >= func(arr).min()) &
(20)                                (label_values <= func(arr).max()))
(12)                        label_values = label_values[cond]
(12)                        yarr = np.linspace(arr.min(), arr.max(), 256)
(12)                        xarr = func(yarr)
(12)                        ix = np.argsort(xarr)
(12)                        values = np.interp(label_values, xarr[ix], yarr[ix])
(8)                     kw = {"markeredgewidth": self.get_linewidths()[0],
(14)                          "alpha": self.get_alpha(),
(14)                          **kwargs}
(8)                     for val, lab in zip(values, label_values):
(12)                        if prop == "colors":
(16)                            color = self.cmap(self.norm(val))
(12)                        elif prop == "sizes":
(16)                            size = np.sqrt(val)
(16)                            if np.isclose(size, 0.0):
(20)                                continue
(12)                        h = mlines.Line2D([0], [0], ls="", color=color, ms=size,
(30)                                          marker=self.get_paths()[0], **kw)
(12)                        handles.append(h)
(12)                        if hasattr(fmt, "set_locs"):
(16)                            fmt.set_locs(label_values)
(12)                        l = fmt(lab)
(12)                        labels.append(l)
(8)                     return handles, labels
(0)             class PolyCollection(_CollectionWithSizes):
(4)                 def __init__(self, verts, sizes=None, *, closed=True, **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     verts : list of array-like
(12)                        The sequence of polygons [*verts0*, *verts1*, ...] where each
(12)                        element *verts_i* defines the vertices of polygon *i* as a 2D
(12)                        array-like of shape (M, 2).
(8)                     sizes : array-like, default: None
(12)                        Squared scaling factors for the polygons. The coordinates of each
(12)                        polygon *verts_i* are multiplied by the square-root of the
(12)                        corresponding entry in *sizes* (i.e., *sizes* specify the scaling
(12)                        of areas). The scaling is applied before the Artist master
(12)                        transform.
(8)                     closed : bool, default: True
(12)                        Whether the polygon should be closed by adding a CLOSEPOLY
(12)                        connection at the end.
(8)                     **kwargs
(12)                        Forwarded to `.Collection`.
(8)                     """
(8)                     super().__init__(**kwargs)
(8)                     self.set_sizes(sizes)
(8)                     self.set_verts(verts, closed)
(8)                     self.stale = True
(4)                 def set_verts(self, verts, closed=True):
(8)                     """
```

```
(8)                     Set the vertices of the polygons.
(8)                     Parameters
(8)                     ----------
(8)                     verts : list of array-like
(12)                        The sequence of polygons [*verts0*, *verts1*, ...] where each
(12)                        element *verts_i* defines the vertices of polygon *i* as a 2D
(12)                        array-like of shape (M, 2).
(8)                     closed : bool, default: True
(12)                        Whether the polygon should be closed by adding a CLOSEPOLY
(12)                        connection at the end.
(8)                     """
(8)                     self.stale = True
(8)                     if isinstance(verts, np.ma.MaskedArray):
(12)                        verts = verts.astype(float).filled(np.nan)
(8)                     if not closed:
(12)                        self._paths = [mpath.Path(xy) for xy in verts]
(12)                        return
(8)                     if isinstance(verts, np.ndarray) and len(verts.shape) == 3:
(12)                        verts_pad = np.concatenate((verts, verts[:, :1]), axis=1)
(12)                        codes = np.empty(verts_pad.shape[1], dtype=mpath.Path.code_type)
(12)                        codes[:] = mpath.Path.LINETO
(12)                        codes[0] = mpath.Path.MOVETO
(12)                        codes[-1] = mpath.Path.CLOSEPOLY
(12)                        self._paths = [mpath.Path(xy, codes) for xy in verts_pad]
(12)                        return
(8)                     self._paths = []
(8)                     for xy in verts:
(12)                        if len(xy):
(16)                            self._paths.append(mpath.Path._create_closed(xy))
(12)                        else:
(16)                            self._paths.append(mpath.Path(xy))
(4)                 set_paths = set_verts
(4)                 def set_verts_and_codes(self, verts, codes):
(8)                     """Initialize vertices with path codes."""
(8)                     if len(verts) != len(codes):
(12)                        raise ValueError("'codes' must be a 1D list or array "
(29)                                         "with the same length of 'verts'")
(8)                     self._paths = [mpath.Path(xy, cds) if len(xy) else mpath.Path(xy)
(23)                                   for xy, cds in zip(verts, codes)]
(8)                     self.stale = True
(0)             class RegularPolyCollection(_CollectionWithSizes):
(4)                 """A collection of n-sided regular polygons."""
(4)                 _path_generator = mpath.Path.unit_regular_polygon
(4)                 _factor = np.pi ** (-1/2)
(4)                 def __init__(self,
(17)                             numsides,
(17)                             *,
(17)                             rotation=0,
(17)                             sizes=(1,),
(17)                             **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     numsides : int
(12)                        The number of sides of the polygon.
(8)                     rotation : float
(12)                        The rotation of the polygon in radians.
(8)                     sizes : tuple of float
(12)                        The area of the circle circumscribing the polygon in points^2.
(8)                     **kwargs
(12)                        Forwarded to `.Collection`.
(8)                     Examples
(8)                     --------
(8)                     See :doc:`/gallery/event_handling/lasso_demo` for a complete example::
(12)                        offsets = np.random.rand(20, 2)
(12)                        facecolors = [cm.jet(x) for x in np.random.rand(20)]
(12)                        collection = RegularPolyCollection(
(16)                            numsides=5, # a pentagon
(16)                            rotation=0, sizes=(50,),
```

```
1019: (16)                          facecolors=facecolors,
1020: (16)                          edgecolors=("black",),
1021: (16)                          linewidths=(1,),
1022: (16)                          offsets=offsets,
1023: (16)                          offset_transform=ax.transData,
1024: (16)                          )
1025: (8)                   """
1026: (8)                   super().__init__(**kwargs)
1027: (8)                   self.set_sizes(sizes)
1028: (8)                   self._numsides = numsides
1029: (8)                   self._paths = [self._path_generator(numsides)]
1030: (8)                   self._rotation = rotation
1031: (8)                   self.set_transform(transforms.IdentityTransform())
1032: (4)               def get_numsides(self):
1033: (8)                   return self._numsides
1034: (4)               def get_rotation(self):
1035: (8)                   return self._rotation
1036: (4)               @artist.allow_rasterization
1037: (4)               def draw(self, renderer):
1038: (8)                   self.set_sizes(self._sizes, self.figure.dpi)
1039: (8)                   self._transforms = [
1040: (12)                      transforms.Affine2D(x).rotate(-self._rotation).get_matrix()
1041: (12)                      for x in self._transforms
1042: (8)                   ]
1043: (8)                   Collection.draw(self, renderer)
1044: (0)           class StarPolygonCollection(RegularPolyCollection):
1045: (4)               """Draw a collection of regular stars with *numsides* points."""
1046: (4)               _path_generator = mpath.Path.unit_regular_star
1047: (0)           class AsteriskPolygonCollection(RegularPolyCollection):
1048: (4)               """Draw a collection of regular asterisks with *numsides* points."""
1049: (4)               _path_generator = mpath.Path.unit_regular_asterisk
1050: (0)           class LineCollection(Collection):
1051: (4)               r"""
1052: (4)               Represents a sequence of `.Line2D`\s that should be drawn together.
1053: (4)               This class extends `.Collection` to represent a sequence of
1054: (4)               `.Line2D`\s instead of just a sequence of `.Patch`\s.
1055: (4)               Just as in `.Collection`, each property of a *LineCollection* may be
either
1056: (4)               a single value or a list of values. This list is then used cyclically for
1057: (4)               each element of the LineCollection, so the property of the ``i``\th
element
1058: (4)               of the collection is::
1059: (6)                 prop[i % len(prop)]
1060: (4)               The properties of each member of a *LineCollection* default to their
values
1061: (4)               in :rc:`lines.*` instead of :rc:`patch.*`, and the property *colors* is
1062: (4)               added in place of *edgecolors*.
1063: (4)               """
1064: (4)               _edge_default = True
1065: (4)               def __init__(self, segments,  # Can be None.
1066: (17)                           *,
1067: (17)                           zorder=2,        # Collection.zorder is 1
1068: (17)                           **kwargs
1069: (17)                           ):
1070: (8)                   """
1071: (8)                   Parameters
1072: (8)                   ----------
1073: (8)                   segments : list of array-like
1074: (12)                      A sequence (*line0*, *line1*, *line2*) of lines, where each line
is a list
1075: (12)                      of points::
1076: (16)                          lineN = [(x0, y0), (x1, y1), ... (xm, ym)]
1077: (12)                      or the equivalent Mx2 numpy array with two columns. Each line
1078: (12)                      can have a different number of segments.
1079: (8)                   linewidths : float or list of float, default: :rc:`lines.linewidth`
1080: (12)                      The width of each line in points.
1081: (8)                   colors : :mpltype:`color` or list of color, default: :rc:`lines.color`
1082: (12)                      A sequence of RGBA tuples (e.g., arbitrary color strings, etc, not
1083: (12)                      allowed).
```

```
(8)                 antialiaseds : bool or list of bool, default: :rc:`lines.antialiased`
(12)                    Whether to use antialiasing for each line.
(8)                 zorder : float, default: 2
(12)                    zorder of the lines once drawn.
(8)                 facecolors : :mpltype:`color` or list of :mpltype:`color`, default:
'none'
(12)                    When setting *facecolors*, each line is interpreted as a boundary
(12)                    for an area, implicitly closing the path from the last point to
the
(12)                    first point. The enclosed area is filled with *facecolor*.
(12)                    In order to manually specify what should count as the "interior"
of
(12)                    each line, please use `.PathCollection` instead, where the
(12)                    "interior" can be specified by appropriate usage of
(12)                    `~.path.Path.CLOSEPOLY`.
(8)                 **kwargs
(12)                    Forwarded to `.Collection`.
(8)                 """
(8)                 kwargs.setdefault('facecolors', 'none')
(8)                 super().__init__(
(12)                    zorder=zorder,
(12)                    **kwargs)
(8)                 self.set_segments(segments)
(4)             def set_segments(self, segments):
(8)                 if segments is None:
(12)                    return
(8)                 self._paths = [mpath.Path(seg) if isinstance(seg, np.ma.MaskedArray)
(23)                               else mpath.Path(np.asarray(seg, float))
(23)                               for seg in segments]
(8)                 self.stale = True
(4)             set_verts = set_segments  # for compatibility with PolyCollection
(4)             set_paths = set_segments
(4)             def get_segments(self):
(8)                 """
(8)                 Returns
(8)                 -------
(8)                 list
(12)                    List of segments in the LineCollection. Each list item contains an
(12)                    array of vertices.
(8)                 """
(8)                 segments = []
(8)                 for path in self._paths:
(12)                    vertices = [
(16)                        vertex
(16)                        for vertex, _
(16)                        in path.iter_segments(simplify=False)
(12)                    ]
(12)                    vertices = np.asarray(vertices)
(12)                    segments.append(vertices)
(8)                 return segments
(4)             def _get_default_linewidth(self):
(8)                 return mpl.rcParams['lines.linewidth']
(4)             def _get_default_antialiased(self):
(8)                 return mpl.rcParams['lines.antialiased']
(4)             def _get_default_edgecolor(self):
(8)                 return mpl.rcParams['lines.color']
(4)             def _get_default_facecolor(self):
(8)                 return 'none'
(4)             def set_alpha(self, alpha):
(8)                 super().set_alpha(alpha)
(8)                 if self._gapcolor is not None:
(12)                    self.set_gapcolor(self._original_gapcolor)
(4)             def set_color(self, c):
(8)                 """
(8)                 Set the edgecolor(s) of the LineCollection.
(8)                 Parameters
(8)                 ----------
(8)                 c : :mpltype:`color` or list of :mpltype:`color`
(12)                    Single color (all lines have same color), or a
```

```
(12)                        sequence of RGBA tuples; if it is a sequence the lines will
(12)                        cycle through the sequence.
(8)                     """
(8)                     self.set_edgecolor(c)
(4)                 set_colors = set_color
(4)                 def get_color(self):
(8)                     return self._edgecolors
(4)                 get_colors = get_color  # for compatibility with old versions
(4)                 def set_gapcolor(self, gapcolor):
(8)                     """
(8)                     Set a color to fill the gaps in the dashed line style.
(8)                     .. note::
(12)                        Striped lines are created by drawing two interleaved dashed lines.
(12)                        There can be overlaps between those two, which may result in
(12)                        artifacts when using transparency.
(12)                        This functionality is experimental and may change.
(8)                     Parameters
(8)                     ----------
(8)                     gapcolor : :mpltype:`color` or list of :mpltype:`color` or None
(12)                        The color with which to fill the gaps. If None, the gaps are
(12)                        unfilled.
(8)                     """
(8)                     self._original_gapcolor = gapcolor
(8)                     self._set_gapcolor(gapcolor)
(4)                 def _set_gapcolor(self, gapcolor):
(8)                     if gapcolor is not None:
(12)                        gapcolor = mcolors.to_rgba_array(gapcolor, self._alpha)
(8)                     self._gapcolor = gapcolor
(8)                     self.stale = True
(4)                 def get_gapcolor(self):
(8)                     return self._gapcolor
(4)                 def _get_inverse_paths_linestyles(self):
(8)                     """
(8)                     Returns the path and pattern for the gaps in the non-solid lines.
(8)                     This path and pattern is the inverse of the path and pattern used to
(8)                     construct the non-solid lines. For solid lines, we set the inverse
path
(8)                     to nans to prevent drawing an inverse line.
(8)                     """
(8)                     path_patterns = [
(12)                        (mpath.Path(np.full((1, 2), np.nan)), ls)
(12)                        if ls == (0, None) else
(12)                        (path, mlines._get_inverse_dash_pattern(*ls))
(12)                        for (path, ls) in
(12)                        zip(self._paths, itertools.cycle(self._linestyles))]
(8)                     return zip(*path_patterns)
(0)             class EventCollection(LineCollection):
(4)                 """
(4)                 A collection of locations along a single axis at which an "event"
occurred.
(4)                 The events are given by a 1-dimensional array. They do not have an
(4)                 amplitude and are displayed as parallel lines.
(4)                 """
(4)                 _edge_default = True
(4)                 def __init__(self,
(17)                             positions,  # Cannot be None.
(17)                             orientation='horizontal',
(17)                             *,
(17)                             lineoffset=0,
(17)                             linelength=1,
(17)                             linewidth=None,
(17)                             color=None,
(17)                             linestyle='solid',
(17)                             antialiased=None,
(17)                             **kwargs
(17)                             ):
(8)                     """
(8)                     Parameters
(8)                     ----------
```

```
(8)                 positions : 1D array-like
(12)                    Each value is an event.
(8)                 orientation : {'horizontal', 'vertical'}, default: 'horizontal'
(12)                    The sequence of events is plotted along this direction.
(12)                    The marker lines of the single events are along the orthogonal
(12)                    direction.
(8)                 lineoffset : float, default: 0
(12)                    The offset of the center of the markers from the origin, in the
(12)                    direction orthogonal to *orientation*.
(8)                 linelength : float, default: 1
(12)                    The total height of the marker (i.e. the marker stretches from
(12)                    ``lineoffset - linelength/2`` to ``lineoffset + linelength/2``).
(8)                 linewidth : float or list thereof, default: :rc:`lines.linewidth`
(12)                    The line width of the event lines, in points.
(8)                 color : :mpltype:`color` or list of :mpltype:`color`, default:
:rc:`lines.color`
(12)                    The color of the event lines.
(8)                 linestyle : str or tuple or list thereof, default: 'solid'
(12)                    Valid strings are ['solid', 'dashed', 'dashdot', 'dotted',
(12)                    '-', '--', '-.', ':']. Dash tuples should be of the form::
(16)                        (offset, onoffseq),
(12)                    where *onoffseq* is an even length tuple of on and off ink
(12)                    in points.
(8)                 antialiased : bool or list thereof, default: :rc:`lines.antialiased`
(12)                    Whether to use antialiasing for drawing the lines.
(8)                 **kwargs
(12)                    Forwarded to `.LineCollection`.
(8)                 Examples
(8)                 --------
(8)                 .. plot:: gallery/lines_bars_and_markers/eventcollection_demo.py
(8)                 """
(8)                 super().__init__([],
(25)                                linewidths=linewidth, linestyles=linestyle,
(25)                                colors=color, antialiaseds=antialiased,
(25)                                **kwargs)
(8)                 self._is_horizontal = True  # Initial value, may be switched below.
(8)                 self._linelength = linelength
(8)                 self._lineoffset = lineoffset
(8)                 self.set_orientation(orientation)
(8)                 self.set_positions(positions)
(4)             def get_positions(self):
(8)                 """
(8)                 Return an array containing the floating-point values of the positions.
(8)                 """
(8)                 pos = 0 if self.is_horizontal() else 1
(8)                 return [segment[0, pos] for segment in self.get_segments()]
(4)             def set_positions(self, positions):
(8)                 """Set the positions of the events."""
(8)                 if positions is None:
(12)                    positions = []
(8)                 if np.ndim(positions) != 1:
(12)                    raise ValueError('positions must be one-dimensional')
(8)                 lineoffset = self.get_lineoffset()
(8)                 linelength = self.get_linelength()
(8)                 pos_idx = 0 if self.is_horizontal() else 1
(8)                 segments = np.empty((len(positions), 2, 2))
(8)                 segments[:, :, pos_idx] = np.sort(positions)[:, None]
(8)                 segments[:, 0, 1 - pos_idx] = lineoffset + linelength / 2
(8)                 segments[:, 1, 1 - pos_idx] = lineoffset - linelength / 2
(8)                 self.set_segments(segments)
(4)             def add_positions(self, position):
(8)                 """Add one or more events at the specified positions."""
(8)                 if position is None or (hasattr(position, 'len') and
(32)                                        len(position) == 0):
(12)                    return
(8)                 positions = self.get_positions()
(8)                 positions = np.hstack([positions, np.asanyarray(position)])
(8)                 self.set_positions(positions)
(4)             extend_positions = append_positions = add_positions
```

```
(4)     def is_horizontal(self):
(8)         """True if the eventcollection is horizontal, False if vertical."""
(8)         return self._is_horizontal
(4)     def get_orientation(self):
(8)         """
(8)         Return the orientation of the event line ('horizontal' or 'vertical').
(8)         """
(8)         return 'horizontal' if self.is_horizontal() else 'vertical'
(4)     def switch_orientation(self):
(8)         """
(8)         Switch the orientation of the event line, either from vertical to
(8)         horizontal or vice versus.
(8)         """
(8)         segments = self.get_segments()
(8)         for i, segment in enumerate(segments):
(12)            segments[i] = np.fliplr(segment)
(8)         self.set_segments(segments)
(8)         self._is_horizontal = not self.is_horizontal()
(8)         self.stale = True
(4)     def set_orientation(self, orientation):
(8)         """
(8)         Set the orientation of the event line.
(8)         Parameters
(8)         ----------
(8)         orientation : {'horizontal', 'vertical'}
(8)         """
(8)         is_horizontal = _api.check_getitem(
(12)            {"horizontal": True, "vertical": False},
(12)            orientation=orientation)
(8)         if is_horizontal == self.is_horizontal():
(12)            return
(8)         self.switch_orientation()
(4)     def get_linelength(self):
(8)         """Return the length of the lines used to mark each event."""
(8)         return self._linelength
(4)     def set_linelength(self, linelength):
(8)         """Set the length of the lines used to mark each event."""
(8)         if linelength == self.get_linelength():
(12)            return
(8)         lineoffset = self.get_lineoffset()
(8)         segments = self.get_segments()
(8)         pos = 1 if self.is_horizontal() else 0
(8)         for segment in segments:
(12)            segment[0, pos] = lineoffset + linelength / 2.
(12)            segment[1, pos] = lineoffset - linelength / 2.
(8)         self.set_segments(segments)
(8)         self._linelength = linelength
(4)     def get_lineoffset(self):
(8)         """Return the offset of the lines used to mark each event."""
(8)         return self._lineoffset
(4)     def set_lineoffset(self, lineoffset):
(8)         """Set the offset of the lines used to mark each event."""
(8)         if lineoffset == self.get_lineoffset():
(12)            return
(8)         linelength = self.get_linelength()
(8)         segments = self.get_segments()
(8)         pos = 1 if self.is_horizontal() else 0
(8)         for segment in segments:
(12)            segment[0, pos] = lineoffset + linelength / 2.
(12)            segment[1, pos] = lineoffset - linelength / 2.
(8)         self.set_segments(segments)
(8)         self._lineoffset = lineoffset
(4)     def get_linewidth(self):
(8)         """Get the width of the lines used to mark each event."""
(8)         return super().get_linewidth()[0]
(4)     def get_linewidths(self):
(8)         return super().get_linewidth()
(4)     def get_color(self):
(8)         """Return the color of the lines used to mark each event."""
```

```
(8)                 return self.get_colors()[0]
(0)         class CircleCollection(_CollectionWithSizes):
(4)             """A collection of circles, drawn using splines."""
(4)             _factor = np.pi ** (-1/2)
(4)             def __init__(self, sizes, **kwargs):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 sizes : float or array-like
(12)                    The area of each circle in points^2.
(8)                 **kwargs
(12)                    Forwarded to `.Collection`.
(8)                 """
(8)                 super().__init__(**kwargs)
(8)                 self.set_sizes(sizes)
(8)                 self.set_transform(transforms.IdentityTransform())
(8)                 self._paths = [mpath.Path.unit_circle()]
(0)         class EllipseCollection(Collection):
(4)             """A collection of ellipses, drawn using splines."""
(4)             def __init__(self, widths, heights, angles, *, units='points', **kwargs):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 widths : array-like
(12)                    The lengths of the first axes (e.g., major axis lengths).
(8)                 heights : array-like
(12)                    The lengths of second axes.
(8)                 angles : array-like
(12)                    The angles of the first axes, degrees CCW from the x-axis.
(8)                 units : {'points', 'inches', 'dots', 'width', 'height', 'x', 'y',
'xy'}
(12)                    The units in which majors and minors are given; 'width' and
(12)                    'height' refer to the dimensions of the axes, while 'x' and 'y'
(12)                    refer to the *offsets* data units. 'xy' differs from all others in
(12)                    that the angle as plotted varies with the aspect ratio, and equals
(12)                    the specified angle only when the aspect ratio is unity.  Hence
(12)                    it behaves the same as the `~.patches.Ellipse` with
(12)                    ``axes.transData`` as its transform.
(8)                 **kwargs
(12)                    Forwarded to `Collection`.
(8)                 """
(8)                 super().__init__(**kwargs)
(8)                 self.set_widths(widths)
(8)                 self.set_heights(heights)
(8)                 self.set_angles(angles)
(8)                 self._units = units
(8)                 self.set_transform(transforms.IdentityTransform())
(8)                 self._transforms = np.empty((0, 3, 3))
(8)                 self._paths = [mpath.Path.unit_circle()]
(4)             def _set_transforms(self):
(8)                 """Calculate transforms immediately before drawing."""
(8)                 ax = self.axes
(8)                 fig = self.figure
(8)                 if self._units == 'xy':
(12)                    sc = 1
(8)                 elif self._units == 'x':
(12)                    sc = ax.bbox.width / ax.viewLim.width
(8)                 elif self._units == 'y':
(12)                    sc = ax.bbox.height / ax.viewLim.height
(8)                 elif self._units == 'inches':
(12)                    sc = fig.dpi
(8)                 elif self._units == 'points':
(12)                    sc = fig.dpi / 72.0
(8)                 elif self._units == 'width':
(12)                    sc = ax.bbox.width
(8)                 elif self._units == 'height':
(12)                    sc = ax.bbox.height
(8)                 elif self._units == 'dots':
(12)                    sc = 1.0
```

```
1422: (8)                    else:
1423: (12)                       raise ValueError(f'Unrecognized units: {self._units!r}')
1424: (8)                    self._transforms = np.zeros((len(self._widths), 3, 3))
1425: (8)                    widths = self._widths * sc
1426: (8)                    heights = self._heights * sc
1427: (8)                    sin_angle = np.sin(self._angles)
1428: (8)                    cos_angle = np.cos(self._angles)
1429: (8)                    self._transforms[:, 0, 0] = widths * cos_angle
1430: (8)                    self._transforms[:, 0, 1] = heights * -sin_angle
1431: (8)                    self._transforms[:, 1, 0] = widths * sin_angle
1432: (8)                    self._transforms[:, 1, 1] = heights * cos_angle
1433: (8)                    self._transforms[:, 2, 2] = 1.0
1434: (8)                    _affine = transforms.Affine2D
1435: (8)                    if self._units == 'xy':
1436: (12)                       m = ax.transData.get_affine().get_matrix().copy()
1437: (12)                       m[:2, 2:] = 0
1438: (12)                       self.set_transform(_affine(m))
1439: (4)                def set_widths(self, widths):
1440: (8)                    """Set the lengths of the first axes (e.g., major axis)."""
1441: (8)                    self._widths = 0.5 * np.asarray(widths).ravel()
1442: (8)                    self.stale = True
1443: (4)                def set_heights(self, heights):
1444: (8)                    """Set the lengths of second axes (e.g., minor axes)."""
1445: (8)                    self._heights = 0.5 * np.asarray(heights).ravel()
1446: (8)                    self.stale = True
1447: (4)                def set_angles(self, angles):
1448: (8)                    """Set the angles of the first axes, degrees CCW from the x-axis."""
1449: (8)                    self._angles = np.deg2rad(angles).ravel()
1450: (8)                    self.stale = True
1451: (4)                def get_widths(self):
1452: (8)                    """Get the lengths of the first axes (e.g., major axis)."""
1453: (8)                    return self._widths * 2
1454: (4)                def get_heights(self):
1455: (8)                    """Set the lengths of second axes (e.g., minor axes)."""
1456: (8)                    return self._heights * 2
1457: (4)                def get_angles(self):
1458: (8)                    """Get the angles of the first axes, degrees CCW from the x-axis."""
1459: (8)                    return np.rad2deg(self._angles)
1460: (4)                @artist.allow_rasterization
1461: (4)                def draw(self, renderer):
1462: (8)                    self._set_transforms()
1463: (8)                    super().draw(renderer)
1464: (0)            class PatchCollection(Collection):
1465: (4)                """
1466: (4)                A generic collection of patches.
1467: (4)                PatchCollection draws faster than a large number of equivalent individual
1468: (4)                Patches. It also makes it easier to assign a colormap to a heterogeneous
1469: (4)                collection of patches.
1470: (4)                """
1471: (4)                def __init__(self, patches, *, match_original=False, **kwargs):
1472: (8)                    """
1473: (8)                    Parameters
1474: (8)                    ----------
1475: (8)                    patches : list of `.Patch`
1476: (12)                       A sequence of Patch objects.  This list may include
1477: (12)                       a heterogeneous assortment of different patch types.
1478: (8)                    match_original : bool, default: False
1479: (12)                       If True, use the colors and linewidths of the original
1480: (12)                       patches.  If False, new colors may be assigned by
1481: (12)                       providing the standard collection arguments, facecolor,
1482: (12)                       edgecolor, linewidths, norm or cmap.
1483: (8)                    **kwargs
1484: (12)                       All other parameters are forwarded to `.Collection`.
1485: (12)                       If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds*
1486: (12)                       are None, they default to their `.rcParams` patch setting, in
1487: (12)                       sequence form.
1488: (8)                    Notes
1489: (8)                    -----
1490: (8)                    The use of `~matplotlib.cm.ScalarMappable` functionality is optional.
```

```
(8)                 If the `~matplotlib.cm.ScalarMappable` matrix ``_A`` has been set (via
(8)                 a call to `~.ScalarMappable.set_array`), at draw time a call to scalar
(8)                 mappable will be made to set the face colors.
(8)                 """
(8)                 if match_original:
(12)                    def determine_facecolor(patch):
(16)                        if patch.get_fill():
(20)                            return patch.get_facecolor()
(16)                        return [0, 0, 0, 0]
(12)                    kwargs['facecolors'] = [determine_facecolor(p) for p in patches]
(12)                    kwargs['edgecolors'] = [p.get_edgecolor() for p in patches]
(12)                    kwargs['linewidths'] = [p.get_linewidth() for p in patches]
(12)                    kwargs['linestyles'] = [p.get_linestyle() for p in patches]
(12)                    kwargs['antialiaseds'] = [p.get_antialiased() for p in patches]
(8)                 super().__init__(**kwargs)
(8)                 self.set_paths(patches)
(4)             def set_paths(self, patches):
(8)                 paths = [p.get_transform().transform_path(p.get_path())
(17)                         for p in patches]
(8)                 self._paths = paths
(0)         class TriMesh(Collection):
(4)             """
(4)             Class for the efficient drawing of a triangular mesh using Gouraud
shading.
(4)             A triangular mesh is a `~matplotlib.tri.Triangulation` object.
(4)             """
(4)             def __init__(self, triangulation, **kwargs):
(8)                 super().__init__(**kwargs)
(8)                 self._triangulation = triangulation
(8)                 self._shading = 'gouraud'
(8)                 self._bbox = transforms.Bbox.unit()
(8)                 xy = np.hstack((triangulation.x.reshape(-1, 1),
(24)                                triangulation.y.reshape(-1, 1)))
(8)                 self._bbox.update_from_data_xy(xy)
(4)             def get_paths(self):
(8)                 if self._paths is None:
(12)                    self.set_paths()
(8)                 return self._paths
(4)             def set_paths(self):
(8)                 self._paths = self.convert_mesh_to_paths(self._triangulation)
(4)             @staticmethod
(4)             def convert_mesh_to_paths(tri):
(8)                 """
(8)                 Convert a given mesh into a sequence of `.Path` objects.
(8)                 This function is primarily of use to implementers of backends that do
(8)                 not directly support meshes.
(8)                 """
(8)                 triangles = tri.get_masked_triangles()
(8)                 verts = np.stack((tri.x[triangles], tri.y[triangles]), axis=-1)
(8)                 return [mpath.Path(x) for x in verts]
(4)             @artist.allow_rasterization
(4)             def draw(self, renderer):
(8)                 if not self.get_visible():
(12)                    return
(8)                 renderer.open_group(self.__class__.__name__, gid=self.get_gid())
(8)                 transform = self.get_transform()
(8)                 tri = self._triangulation
(8)                 triangles = tri.get_masked_triangles()
(8)                 verts = np.stack((tri.x[triangles], tri.y[triangles]), axis=-1)
(8)                 self.update_scalarmappable()
(8)                 colors = self._facecolors[triangles]
(8)                 gc = renderer.new_gc()
(8)                 self._set_gc_clip(gc)
(8)                 gc.set_linewidth(self.get_linewidth()[0])
(8)                 renderer.draw_gouraud_triangles(gc, verts, colors, transform.frozen())
(8)                 gc.restore()
(8)                 renderer.close_group(self.__class__.__name__)
(0)         class _MeshData:
(4)             r"""
```

```
    Class for managing the two dimensional coordinates of Quadrilateral meshes
    and the associated data with them. This class is a mixin and is intended
to
    be used with another collection that will implement the draw separately.
    A quadrilateral mesh is a grid of M by N adjacent quadrilaterals that are
    defined via a (M+1, N+1) grid of vertices. The quadrilateral (m, n) is
    defined by the vertices ::
                (m+1, n) ----------- (m+1, n+1)
                   /                  /
                  /                  /
                 /                  /
                (m, n) -------- (m, n+1)
    The mesh need not be regular and the polygons need not be convex.
    Parameters
    ----------
    coordinates : (M+1, N+1, 2) array-like
        The vertices. ``coordinates[m, n]`` specifies the (x, y) coordinates
        of vertex (m, n).
    shading : {'flat', 'gouraud'}, default: 'flat'
    """
    def __init__(self, coordinates, *, shading='flat'):
        _api.check_shape((None, None, 2), coordinates=coordinates)
        self._coordinates = coordinates
        self._shading = shading
    def set_array(self, A):
        """
        Set the data values.
        Parameters
        ----------
        A : array-like
            The mesh data. Supported array shapes are:
            - (M, N) or (M*N,): a mesh with scalar data. The values are mapped
              to colors using normalization and a colormap. See parameters
              *norm*, *cmap*, *vmin*, *vmax*.
            - (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
            - (M, N, 4): an image with RGBA values (0-1 float or 0-255 int),
              i.e. including transparency.
            If the values are provided as a 2D grid, the shape must match the
            coordinates grid. If the values are 1D, they are reshaped to 2D.
            M, N follow from the coordinates grid, where the coordinates grid
            shape is (M, N) for 'gouraud' *shading* and (M+1, N+1) for 'flat'
            shading.
        """
        height, width = self._coordinates.shape[0:-1]
        if self._shading == 'flat':
            h, w = height - 1, width - 1
        else:
            h, w = height, width
        ok_shapes = [(h, w, 3), (h, w, 4), (h, w), (h * w,)]
        if A is not None:
            shape = np.shape(A)
            if shape not in ok_shapes:
                raise ValueError(
                    f"For X ({width}) and Y ({height}) with {self._shading} "
                    f"shading, A should have shape "
                    f"{' or '.join(map(str, ok_shapes))}, not {A.shape}")
        return super().set_array(A)
    def get_coordinates(self):
        """
        Return the vertices of the mesh as an (M+1, N+1, 2) array.
        M, N are the number of quadrilaterals in the rows / columns of the
        mesh, corresponding to (M+1, N+1) vertices.
        The last dimension specifies the components (x, y).
        """
        return self._coordinates
    def get_edgecolor(self):
        return super().get_edgecolor().reshape(-1, 4)
    def get_facecolor(self):
        return super().get_facecolor().reshape(-1, 4)
```

```
(4)                 @staticmethod
(4)                 def _convert_mesh_to_paths(coordinates):
(8)                     """
(8)                     Convert a given mesh into a sequence of `.Path` objects.
(8)                     This function is primarily of use to implementers of backends that do
(8)                     not directly support quadmeshes.
(8)                     """
(8)                     if isinstance(coordinates, np.ma.MaskedArray):
(12)                        c = coordinates.data
(8)                     else:
(12)                        c = coordinates
(8)                     points = np.concatenate([
(12)                        c[:-1, :-1],
(12)                        c[:-1, 1:],
(12)                        c[1:, 1:],
(12)                        c[1:, :-1],
(12)                        c[:-1, :-1]
(8)                     ], axis=2).reshape((-1, 5, 2))
(8)                     return [mpath.Path(x) for x in points]
(4)                 def _convert_mesh_to_triangles(self, coordinates):
(8)                     """
(8)                     Convert a given mesh into a sequence of triangles, each point
(8)                     with its own color.  The result can be used to construct a call to
(8)                     `~.RendererBase.draw_gouraud_triangles`.
(8)                     """
(8)                     if isinstance(coordinates, np.ma.MaskedArray):
(12)                        p = coordinates.data
(8)                     else:
(12)                        p = coordinates
(8)                     p_a = p[:-1, :-1]
(8)                     p_b = p[:-1, 1:]
(8)                     p_c = p[1:, 1:]
(8)                     p_d = p[1:, :-1]
(8)                     p_center = (p_a + p_b + p_c + p_d) / 4.0
(8)                     triangles = np.concatenate([
(12)                        p_a, p_b, p_center,
(12)                        p_b, p_c, p_center,
(12)                        p_c, p_d, p_center,
(12)                        p_d, p_a, p_center,
(8)                     ], axis=2).reshape((-1, 3, 2))
(8)                     c = self.get_facecolor().reshape((*coordinates.shape[:2], 4))
(8)                     z = self.get_array()
(8)                     mask = z.mask if np.ma.is_masked(z) else None
(8)                     if mask is not None:
(12)                        c[mask, 3] = np.nan
(8)                     c_a = c[:-1, :-1]
(8)                     c_b = c[:-1, 1:]
(8)                     c_c = c[1:, 1:]
(8)                     c_d = c[1:, :-1]
(8)                     c_center = (c_a + c_b + c_c + c_d) / 4.0
(8)                     colors = np.concatenate([
(12)                        c_a, c_b, c_center,
(12)                        c_b, c_c, c_center,
(12)                        c_c, c_d, c_center,
(12)                        c_d, c_a, c_center,
(8)                     ], axis=2).reshape((-1, 3, 4))
(8)                     tmask = np.isnan(colors[..., 2, 3])
(8)                     return triangles[~tmask], colors[~tmask]
(0)             class QuadMesh(_MeshData, Collection):
(4)                 r"""
(4)                 Class for the efficient drawing of a quadrilateral mesh.
(4)                 A quadrilateral mesh is a grid of M by N adjacent quadrilaterals that are
(4)                 defined via a (M+1, N+1) grid of vertices. The quadrilateral (m, n) is
(4)                 defined by the vertices ::
(15)                            (m+1, n) ----------- (m+1, n+1)
(18)                               /                   /
(17)                              /                   /
(16)                             /                   /
(12)                         (m, n) -------- (m, n+1)
```

```
    The mesh need not be regular and the polygons need not be convex.
    Parameters
    ----------
    coordinates : (M+1, N+1, 2) array-like
        The vertices. ``coordinates[m, n]`` specifies the (x, y) coordinates
        of vertex (m, n).
    antialiased : bool, default: True
    shading : {'flat', 'gouraud'}, default: 'flat'
    Notes
    -----
    Unlike other `.Collection`\s, the default *pickradius* of `.QuadMesh` is 0,
    i.e. `~.Artist.contains` checks whether the test point is within any of the
    mesh quadrilaterals.
    """
    def __init__(self, coordinates, *, antialiased=True, shading='flat',
                 **kwargs):
        kwargs.setdefault("pickradius", 0)
        super().__init__(coordinates=coordinates, shading=shading)
        Collection.__init__(self, **kwargs)
        self._antialiased = antialiased
        self._bbox = transforms.Bbox.unit()
        self._bbox.update_from_data_xy(self._coordinates.reshape(-1, 2))
        self.set_mouseover(False)
    def get_paths(self):
        if self._paths is None:
            self.set_paths()
        return self._paths
    def set_paths(self):
        self._paths = self._convert_mesh_to_paths(self._coordinates)
        self.stale = True
    def get_datalim(self, transData):
        return (self.get_transform() - transData).transform_bbox(self._bbox)
    @artist.allow_rasterization
    def draw(self, renderer):
        if not self.get_visible():
            return
        renderer.open_group(self.__class__.__name__, self.get_gid())
        transform = self.get_transform()
        offset_trf = self.get_offset_transform()
        offsets = self.get_offsets()
        if self.have_units():
            xs = self.convert_xunits(offsets[:, 0])
            ys = self.convert_yunits(offsets[:, 1])
            offsets = np.column_stack([xs, ys])
        self.update_scalarmappable()
        if not transform.is_affine:
            coordinates = self._coordinates.reshape((-1, 2))
            coordinates = transform.transform(coordinates)
            coordinates = coordinates.reshape(self._coordinates.shape)
            transform = transforms.IdentityTransform()
        else:
            coordinates = self._coordinates
        if not offset_trf.is_affine:
            offsets = offset_trf.transform_non_affine(offsets)
            offset_trf = offset_trf.get_affine()
        gc = renderer.new_gc()
        gc.set_snap(self.get_snap())
        self._set_gc_clip(gc)
        gc.set_linewidth(self.get_linewidth()[0])
        if self._shading == 'gouraud':
            triangles, colors = self._convert_mesh_to_triangles(coordinates)
            renderer.draw_gouraud_triangles(
                gc, triangles, colors, transform.frozen())
        else:
            renderer.draw_quad_mesh(
                gc, transform.frozen(),
                coordinates.shape[1] - 1, coordinates.shape[0] - 1,
```

```
(16)                        coordinates, offsets, offset_trf,
(16)                        self.get_facecolor().reshape((-1, 4)),
(16)                        self._antialiased, self.get_edgecolors().reshape((-1, 4)))
(8)                 gc.restore()
(8)                 renderer.close_group(self.__class__.__name__)
(8)                 self.stale = False
(4)             def get_cursor_data(self, event):
(8)                 contained, info = self.contains(event)
(8)                 if contained and self.get_array() is not None:
(12)                    return self.get_array().ravel()[info["ind"]]
(8)                 return None
(0)         class PolyQuadMesh(_MeshData, PolyCollection):
(4)             """
(4)             Class for drawing a quadrilateral mesh as individual Polygons.
(4)             A quadrilateral mesh is a grid of M by N adjacent quadrilaterals that are
(4)             defined via a (M+1, N+1) grid of vertices. The quadrilateral (m, n) is
(4)             defined by the vertices ::
(15)                      (m+1, n) ----------- (m+1, n+1)
(18)                         /                   /
(17)                        /                 /
(16)                       /               /
(12)                    (m, n) -------- (m, n+1)
(4)             The mesh need not be regular and the polygons need not be convex.
(4)             Parameters
(4)             ----------
(4)             coordinates : (M+1, N+1, 2) array-like
(8)                 The vertices. ``coordinates[m, n]`` specifies the (x, y) coordinates
(8)                 of vertex (m, n).
(4)             Notes
(4)             -----
(4)             Unlike `.QuadMesh`, this class will draw each cell as an individual
Polygon.
(4)             This is significantly slower, but allows for more flexibility when wanting
(4)             to add additional properties to the cells, such as hatching.
(4)             Another difference from `.QuadMesh` is that if any of the vertices or data
(4)             of a cell are masked, that Polygon will **not** be drawn and it won't be
in
(4)             the list of paths returned.
(4)             """
(4)             def __init__(self, coordinates, **kwargs):
(8)                 self._deprecated_compression = False
(8)                 super().__init__(coordinates=coordinates)
(8)                 PolyCollection.__init__(self, verts=[], **kwargs)
(8)                 self._original_mask = ~self._get_unmasked_polys()
(8)                 self._deprecated_compression = np.any(self._original_mask)
(8)                 self._set_unmasked_verts()
(4)             def _get_unmasked_polys(self):
(8)                 """Get the unmasked regions using the coordinates and array"""
(8)                 mask = np.any(np.ma.getmaskarray(self._coordinates), axis=-1)
(8)                 mask = (mask[0:-1, 0:-1] | mask[1:, 1:] | mask[0:-1, 1:] | mask[1:,
0:-1])
(8)                 if (getattr(self, "_deprecated_compression", False) and
(16)                        np.any(self._original_mask)):
(12)                    return ~(mask | self._original_mask)
(8)                 with cbook._setattr_cm(self, _deprecated_compression=False):
(12)                    arr = self.get_array()
(8)                 if arr is not None:
(12)                    arr = np.ma.getmaskarray(arr)
(12)                    if arr.ndim == 3:
(16)                        mask |= np.any(arr, axis=-1)
(12)                    elif arr.ndim == 2:
(16)                        mask |= arr
(12)                    else:
(16)                        mask |= arr.reshape(self._coordinates[:-1, :-1, :].shape[:2])
(8)                 return ~mask
(4)             def _set_unmasked_verts(self):
(8)                 X = self._coordinates[..., 0]
(8)                 Y = self._coordinates[..., 1]
(8)                 unmask = self._get_unmasked_polys()
```

```
1829: (8)                   X1 = np.ma.filled(X[:-1, :-1])[unmask]
1830: (8)                   Y1 = np.ma.filled(Y[:-1, :-1])[unmask]
1831: (8)                   X2 = np.ma.filled(X[1:, :-1])[unmask]
1832: (8)                   Y2 = np.ma.filled(Y[1:, :-1])[unmask]
1833: (8)                   X3 = np.ma.filled(X[1:, 1:])[unmask]
1834: (8)                   Y3 = np.ma.filled(Y[1:, 1:])[unmask]
1835: (8)                   X4 = np.ma.filled(X[:-1, 1:])[unmask]
1836: (8)                   Y4 = np.ma.filled(Y[:-1, 1:])[unmask]
1837: (8)                   npoly = len(X1)
1838: (8)                   xy = np.ma.stack([X1, Y1, X2, Y2, X3, Y3, X4, Y4, X1, Y1], axis=-1)
1839: (8)                   verts = xy.reshape((npoly, 5, 2))
1840: (8)                   self.set_verts(verts)
1841: (4)               def get_edgecolor(self):
1842: (8)                   ec = super().get_edgecolor()
1843: (8)                   unmasked_polys = self._get_unmasked_polys().ravel()
1844: (8)                   if len(ec) != len(unmasked_polys):
1845: (12)                      return ec
1846: (8)                   return ec[unmasked_polys, :]
1847: (4)               def get_facecolor(self):
1848: (8)                   fc = super().get_facecolor()
1849: (8)                   unmasked_polys = self._get_unmasked_polys().ravel()
1850: (8)                   if len(fc) != len(unmasked_polys):
1851: (12)                      return fc
1852: (8)                   return fc[unmasked_polys, :]
1853: (4)               def set_array(self, A):
1854: (8)                   prev_unmask = self._get_unmasked_polys()
1855: (8)                   if self._deprecated_compression and np.ndim(A) == 1:
1856: (12)                      _api.warn_deprecated("3.8", message="Setting a PolyQuadMesh array
using "
1857: (33)                                           "the compressed values is deprecated. "
1858: (33)                                           "Pass the full 2D shape of the original array
"
1859: (33)                                           f"{prev_unmask.shape} including the masked
elements.")
1860: (12)                      Afull = np.empty(self._original_mask.shape)
1861: (12)                      Afull[~self._original_mask] = A
1862: (12)                      mask = self._original_mask.copy()
1863: (12)                      mask[~self._original_mask] |= np.ma.getmask(A)
1864: (12)                      A = np.ma.array(Afull, mask=mask)
1865: (12)                      return super().set_array(A)
1866: (8)                   self._deprecated_compression = False
1867: (8)                   super().set_array(A)
1868: (8)                   if not np.array_equal(prev_unmask, self._get_unmasked_polys()):
1869: (12)                      self._set_unmasked_verts()
1870: (4)               def get_array(self):
1871: (8)                   A = super().get_array()
1872: (8)                   if A is None:
1873: (12)                      return
1874: (8)                   if self._deprecated_compression and np.any(np.ma.getmask(A)):
1875: (12)                      _api.warn_deprecated("3.8", message=(
1876: (16)                          "Getting the array from a PolyQuadMesh will return the full "
1877: (16)                          "array in the future (uncompressed). To get this behavior now
"
1878: (16)                          "set the PolyQuadMesh with a 2D array .set_array(data2d)."))
1879: (12)                      return np.ma.compressed(A)
1880: (8)                   return A
```

File 32 - colorbar.py:

```
1: (0)              """
2: (0)              Colorbars are a visualization of the mapping from scalar values to colors.
3: (0)              In Matplotlib they are drawn into a dedicated `~.axes.Axes`.
4: (0)              .. note::
5: (3)                 Colorbars are typically created through `.Figure.colorbar` or its pyplot
6: (3)                 wrapper `.pyplot.colorbar`, which internally use `.Colorbar` together with
7: (3)                 `.make_axes_gridspec` (for `.GridSpec`-positioned Axes) or `.make_axes`
(for
```

```
8: (3)                 non-`.GridSpec`-positioned Axes).
9: (3)                 End-users most likely won't need to directly use this module's API.
10: (0)             """
11: (0)             import logging
12: (0)             import numpy as np
13: (0)             import matplotlib as mpl
14: (0)             from matplotlib import _api, cbook, collections, cm, colors, contour, ticker
15: (0)             import matplotlib.artist as martist
16: (0)             import matplotlib.patches as mpatches
17: (0)             import matplotlib.path as mpath
18: (0)             import matplotlib.spines as mspines
19: (0)             import matplotlib.transforms as mtransforms
20: (0)             from matplotlib import _docstring
21: (0)             _log = logging.getLogger(__name__)
22: (0)             _docstring.interpd.update(
23: (4)                 _make_axes_kw_doc="""
24: (0)             location : None or {'left', 'right', 'top', 'bottom'}
25: (4)                 The location, relative to the parent Axes, where the colorbar Axes
26: (4)                 is created.  It also determines the *orientation* of the colorbar
27: (4)                 (colorbars on the left and right are vertical, colorbars at the top
28: (4)                 and bottom are horizontal).  If None, the location will come from the
29: (4)                 *orientation* if it is set (vertical colorbars on the right, horizontal
30: (4)                 ones at the bottom), or default to 'right' if *orientation* is unset.
31: (0)             orientation : None or {'vertical', 'horizontal'}
32: (4)                 The orientation of the colorbar.  It is preferable to set the *location*
33: (4)                 of the colorbar, as that also determines the *orientation*; passing
34: (4)                 incompatible values for *location* and *orientation* raises an exception.
35: (0)             fraction : float, default: 0.15
36: (4)                 Fraction of original Axes to use for colorbar.
37: (0)             shrink : float, default: 1.0
38: (4)                 Fraction by which to multiply the size of the colorbar.
39: (0)             aspect : float, default: 20
40: (4)                 Ratio of long to short dimensions.
41: (0)             pad : float, default: 0.05 if vertical, 0.15 if horizontal
42: (4)                 Fraction of original Axes between colorbar and new image Axes.
43: (0)             anchor : (float, float), optional
44: (4)                 The anchor point of the colorbar Axes.
45: (4)                 Defaults to (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal.
46: (0)             panchor : (float, float), or *False*, optional
47: (4)                 The anchor point of the colorbar parent Axes. If *False*, the parent
48: (4)                 axes' anchor will be unchanged.
49: (4)                 Defaults to (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal.""",
50: (4)                 _colormap_kw_doc="""
51: (0)             extend : {'neither', 'both', 'min', 'max'}
52: (4)                 Make pointed end(s) for out-of-range values (unless 'neither').  These are
53: (4)                 set for a given colormap using the colormap set_under and set_over
methods.
54: (0)             extendfrac : {*None*, 'auto', length, lengths}
55: (4)                 If set to *None*, both the minimum and maximum triangular colorbar
56: (4)                 extensions will have a length of 5% of the interior colorbar length (this
57: (4)                 is the default setting).
58: (4)                 If set to 'auto', makes the triangular colorbar extensions the same
lengths
59: (4)                 as the interior boxes (when *spacing* is set to 'uniform') or the same
60: (4)                 lengths as the respective adjacent interior boxes (when *spacing* is set
to
61: (4)                 'proportional').
62: (4)                 If a scalar, indicates the length of both the minimum and maximum
63: (4)                 triangular colorbar extensions as a fraction of the interior colorbar
64: (4)                 length.  A two-element sequence of fractions may also be given, indicating
65: (4)                 the lengths of the minimum and maximum colorbar extensions respectively as
66: (4)                 a fraction of the interior colorbar length.
67: (0)             extendrect : bool
68: (4)                 If *False* the minimum and maximum colorbar extensions will be triangular
69: (4)                 (the default).  If *True* the extensions will be rectangular.
70: (0)             spacing : {'uniform', 'proportional'}
71: (4)                 For discrete colorbars (`.BoundaryNorm` or contours), 'uniform' gives each
72: (4)                 color the same space; 'proportional' makes the space proportional to the
73: (4)                 data interval.
```

```
 74: (0)             ticks : None or list of ticks or Locator
 75: (4)                 If None, ticks are determined automatically from the input.
 76: (0)             format : None or str or Formatter
 77: (4)                 If None, `~.ticker.ScalarFormatter` is used.
 78: (4)                 Format strings, e.g., ``"%4.2e"`` or ``"{x:.2e}"``, are supported.
 79: (4)                 An alternative `~.ticker.Formatter` may be given instead.
 80: (0)             drawedges : bool
 81: (4)                 Whether to draw lines at color boundaries.
 82: (0)             label : str
 83: (4)                 The label on the colorbar's long axis.
 84: (0)             boundaries, values : None or a sequence
 85: (4)                 If unset, the colormap will be displayed on a 0-1 scale.
 86: (4)                 If sequences, *values* must have a length 1 less than *boundaries*.  For
 87: (4)                 each region delimited by adjacent entries in *boundaries*, the color
mapped
 88: (4)                 to the corresponding value in values will be used.
 89: (4)                 Normally only useful for indexed colors (i.e. ``norm=NoNorm()``) or other
 90: (4)                 unusual circumstances.""")
 91: (0)             def _set_ticks_on_axis_warn(*args, **kwargs):
 92: (4)                 _api.warn_external("Use the colorbar set_ticks() method instead.")
 93: (0)             class _ColorbarSpine(mspines.Spine):
 94: (4)                 def __init__(self, axes):
 95: (8)                     self._ax = axes
 96: (8)                     super().__init__(axes, 'colorbar', mpath.Path(np.empty((0, 2))))
 97: (8)                     mpatches.Patch.set_transform(self, axes.transAxes)
 98: (4)                 def get_window_extent(self, renderer=None):
 99: (8)                     return mpatches.Patch.get_window_extent(self, renderer=renderer)
100: (4)                 def set_xy(self, xy):
101: (8)                     self._path = mpath.Path(xy, closed=True)
102: (8)                     self._xy = xy
103: (8)                     self.stale = True
104: (4)                 def draw(self, renderer):
105: (8)                     ret = mpatches.Patch.draw(self, renderer)
106: (8)                     self.stale = False
107: (8)                     return ret
108: (0)             class _ColorbarAxesLocator:
109: (4)                 """
110: (4)                 Shrink the Axes if there are triangular or rectangular extends.
111: (4)                 """
112: (4)                 def __init__(self, cbar):
113: (8)                     self._cbar = cbar
114: (8)                     self._orig_locator = cbar.ax._axes_locator
115: (4)                 def __call__(self, ax, renderer):
116: (8)                     if self._orig_locator is not None:
117: (12)                        pos = self._orig_locator(ax, renderer)
118: (8)                     else:
119: (12)                        pos = ax.get_position(original=True)
120: (8)                     if self._cbar.extend == 'neither':
121: (12)                        return pos
122: (8)                     y, extendlen = self._cbar._proportional_y()
123: (8)                     if not self._cbar._extend_lower():
124: (12)                        extendlen[0] = 0
125: (8)                     if not self._cbar._extend_upper():
126: (12)                        extendlen[1] = 0
127: (8)                     len = sum(extendlen) + 1
128: (8)                     shrink = 1 / len
129: (8)                     offset = extendlen[0] / len
130: (8)                     if hasattr(ax, '_colorbar_info'):
131: (12)                        aspect = ax._colorbar_info['aspect']
132: (8)                     else:
133: (12)                        aspect = False
134: (8)                     if self._cbar.orientation == 'vertical':
135: (12)                        if aspect:
136: (16)                            self._cbar.ax.set_box_aspect(aspect*shrink)
137: (12)                        pos = pos.shrunk(1, shrink).translated(0, offset * pos.height)
138: (8)                     else:
139: (12)                        if aspect:
140: (16)                            self._cbar.ax.set_box_aspect(1/(aspect * shrink))
141: (12)                        pos = pos.shrunk(shrink, 1).translated(offset * pos.width, 0)
```

```
(8)                 return pos
(4)             def get_subplotspec(self):
(8)                 return (
(12)                    self._cbar.ax.get_subplotspec()
(12)                    or getattr(self._orig_locator, "get_subplotspec", lambda: None)())
(0)         @_docstring.interpd
(0)         class Colorbar:
(4)             r"""
(4)             Draw a colorbar in an existing Axes.
(4)             Typically, colorbars are created using `.Figure.colorbar` or
(4)             `.pyplot.colorbar` and associated with `.ScalarMappable`\s (such as an
(4)             `.AxesImage` generated via `~.axes.Axes.imshow`).
(4)             In order to draw a colorbar not associated with other elements in the
(4)             figure, e.g. when showing a colormap by itself, one can create an empty
(4)             `.ScalarMappable`, or directly pass *cmap* and *norm* instead of
*mappable*
(4)             to `Colorbar`.
(4)             Useful public methods are :meth:`set_label` and :meth:`add_lines`.
(4)             Attributes
(4)             ----------
(4)             ax : `~matplotlib.axes.Axes`
(8)                 The `~.axes.Axes` instance in which the colorbar is drawn.
(4)             lines : list
(8)                 A list of `.LineCollection` (empty if no lines were drawn).
(4)             dividers : `.LineCollection`
(8)                 A LineCollection (empty if *drawedges* is ``False``).
(4)             Parameters
(4)             ----------
(4)             ax : `~matplotlib.axes.Axes`
(8)                 The `~.axes.Axes` instance in which the colorbar is drawn.
(4)             mappable : `.ScalarMappable`
(8)                 The mappable whose colormap and norm will be used.
(8)                 To show the under- and over- value colors, the mappable's norm should
(8)                 be specified as ::
(12)                    norm = colors.Normalize(clip=False)
(8)                 To show the colors versus index instead of on a 0-1 scale, use::
(12)                    norm=colors.NoNorm()
(4)             cmap : `~matplotlib.colors.Colormap`, default: :rc:`image.cmap`
(8)                 The colormap to use.  This parameter is ignored, unless *mappable* is
(8)                 None.
(4)             norm : `~matplotlib.colors.Normalize`
(8)                 The normalization to use.  This parameter is ignored, unless
*mappable*
(8)                 is None.
(4)             alpha : float
(8)                 The colorbar transparency between 0 (transparent) and 1 (opaque).
(4)             orientation : None or {'vertical', 'horizontal'}
(8)                 If None, use the value determined by *location*. If both
(8)                 *orientation* and *location* are None then defaults to 'vertical'.
(4)             ticklocation : {'auto', 'left', 'right', 'top', 'bottom'}
(8)                 The location of the colorbar ticks. The *ticklocation* must match
(8)                 *orientation*. For example, a horizontal colorbar can only have ticks
(8)                 at the top or the bottom. If 'auto', the ticks will be the same as
(8)                 *location*, so a colorbar to the left will have ticks to the left. If
(8)                 *location* is None, the ticks will be at the bottom for a horizontal
(8)                 colorbar and at the right for a vertical.
(4)             %(_colormap_kw_doc)s
(4)             location : None or {'left', 'right', 'top', 'bottom'}
(8)                 Set the *orientation* and *ticklocation* of the colorbar using a
(8)                 single argument. Colorbars on the left and right are vertical,
(8)                 colorbars at the top and bottom are horizontal. The *ticklocation* is
(8)                 the same as *location*, so if *location* is 'top', the ticks are on
(8)                 the top. *orientation* and/or *ticklocation* can be provided as well
(8)                 and overrides the value set by *location*, but there will be an error
(8)                 for incompatible combinations.
(8)                 .. versionadded:: 3.7
(4)             """
(4)             n_rasterize = 50  # rasterize solids if number of colors >= n_rasterize
(4)             def __init__(self, ax, mappable=None, *, cmap=None,
```

```
209: (17)                             norm=None,
210: (17)                             alpha=None,
211: (17)                             values=None,
212: (17)                             boundaries=None,
213: (17)                             orientation=None,
214: (17)                             ticklocation='auto',
215: (17)                             extend=None,
216: (17)                             spacing='uniform',  # uniform or proportional
217: (17)                             ticks=None,
218: (17)                             format=None,
219: (17)                             drawedges=False,
220: (17)                             extendfrac=None,
221: (17)                             extendrect=False,
222: (17)                             label='',
223: (17)                             location=None,
224: (17)                             ):
225: (8)                     if mappable is None:
226: (12)                         mappable = cm.ScalarMappable(norm=norm, cmap=cmap)
227: (8)                     self.mappable = mappable
228: (8)                     cmap = mappable.cmap
229: (8)                     norm = mappable.norm
230: (8)                     filled = True
231: (8)                     if isinstance(mappable, contour.ContourSet):
232: (12)                        cs = mappable
233: (12)                        alpha = cs.get_alpha()
234: (12)                        boundaries = cs._levels
235: (12)                        values = cs.cvalues
236: (12)                        extend = cs.extend
237: (12)                        filled = cs.filled
238: (12)                        if ticks is None:
239: (16)                            ticks = ticker.FixedLocator(cs.levels, nbins=10)
240: (8)                     elif isinstance(mappable, martist.Artist):
241: (12)                        alpha = mappable.get_alpha()
242: (8)                     mappable.colorbar = self
243: (8)                     mappable.colorbar_cid = mappable.callbacks.connect(
244: (12)                        'changed', self.update_normal)
245: (8)                     location_orientation = _get_orientation_from_location(location)
246: (8)                     _api.check_in_list(
247: (12)                        [None, 'vertical', 'horizontal'], orientation=orientation)
248: (8)                     _api.check_in_list(
249: (12)                        ['auto', 'left', 'right', 'top', 'bottom'],
250: (12)                        ticklocation=ticklocation)
251: (8)                     _api.check_in_list(
252: (12)                        ['uniform', 'proportional'], spacing=spacing)
253: (8)                     if location_orientation is not None and orientation is not None:
254: (12)                        if location_orientation != orientation:
255: (16)                            raise TypeError(
256: (20)                                "location and orientation are mutually exclusive")
257: (8)                     else:
258: (12)                        orientation = orientation or location_orientation or "vertical"
259: (8)                     self.ax = ax
260: (8)                     self.ax._axes_locator = _ColorbarAxesLocator(self)
261: (8)                     if extend is None:
262: (12)                        if (not isinstance(mappable, contour.ContourSet)
263: (20)                                and getattr(cmap, 'colorbar_extend', False) is not False):
264: (16)                            extend = cmap.colorbar_extend
265: (12)                        elif hasattr(norm, 'extend'):
266: (16)                            extend = norm.extend
267: (12)                        else:
268: (16)                            extend = 'neither'
269: (8)                     self.alpha = None
270: (8)                     self.set_alpha(alpha)
271: (8)                     self.cmap = cmap
272: (8)                     self.norm = norm
273: (8)                     self.values = values
274: (8)                     self.boundaries = boundaries
275: (8)                     self.extend = extend
276: (8)                     self._inside = _api.check_getitem(
277: (12)                        {'neither': slice(0, None), 'both': slice(1, -1),
```

```
(13)                         'min': slice(1, None), 'max': slice(0, -1)},
(12)                         extend=extend)
(8)                     self.spacing = spacing
(8)                     self.orientation = orientation
(8)                     self.drawedges = drawedges
(8)                     self._filled = filled
(8)                     self.extendfrac = extendfrac
(8)                     self.extendrect = extendrect
(8)                     self._extend_patches = []
(8)                     self.solids = None
(8)                     self.solids_patches = []
(8)                     self.lines = []
(8)                     for spine in self.ax.spines.values():
(12)                         spine.set_visible(False)
(8)                     self.outline = self.ax.spines['outline'] = _ColorbarSpine(self.ax)
(8)                     self.dividers = collections.LineCollection(
(12)                         [],
(12)                         colors=[mpl.rcParams['axes.edgecolor']],
(12)                         linewidths=[0.5 * mpl.rcParams['axes.linewidth']],
(12)                         clip_on=False)
(8)                     self.ax.add_collection(self.dividers)
(8)                     self._locator = None
(8)                     self._minorlocator = None
(8)                     self._formatter = None
(8)                     self._minorformatter = None
(8)                     if ticklocation == 'auto':
(12)                         ticklocation = _get_ticklocation_from_orientation(
(16)                             orientation) if location is None else location
(8)                     self.ticklocation = ticklocation
(8)                     self.set_label(label)
(8)                     self._reset_locator_formatter_scale()
(8)                     if np.iterable(ticks):
(12)                         self._locator = ticker.FixedLocator(ticks, nbins=len(ticks))
(8)                     else:
(12)                         self._locator = ticks
(8)                     if isinstance(format, str):
(12)                         try:
(16)                             self._formatter = ticker.FormatStrFormatter(format)
(16)                             _ = self._formatter(0)
(12)                         except (TypeError, ValueError):
(16)                             self._formatter = ticker.StrMethodFormatter(format)
(8)                     else:
(12)                         self._formatter = format  # Assume it is a Formatter or None
(8)                     self._draw_all()
(8)                     if isinstance(mappable, contour.ContourSet) and not mappable.filled:
(12)                         self.add_lines(mappable)
(8)                     self.ax._colorbar = self
(8)                     if (isinstance(self.norm, (colors.BoundaryNorm, colors.NoNorm)) or
(16)                             isinstance(self.mappable, contour.ContourSet)):
(12)                         self.ax.set_navigate(False)
(8)                     self._interactive_funcs = ["_get_view", "_set_view",
(35)                                            "_set_view_from_bbox", "drag_pan"]
(8)                     for x in self._interactive_funcs:
(12)                         setattr(self.ax, x, getattr(self, x))
(8)                     self.ax.cla = self._cbar_cla
(8)                     self._extend_cid1 = self.ax.callbacks.connect(
(12)                         "xlim_changed", self._do_extends)
(8)                     self._extend_cid2 = self.ax.callbacks.connect(
(12)                         "ylim_changed", self._do_extends)
(4)                 @property
(4)                 def locator(self):
(8)                     """Major tick `.Locator` for the colorbar."""
(8)                     return self._long_axis().get_major_locator()
(4)                 @locator.setter
(4)                 def locator(self, loc):
(8)                     self._long_axis().set_major_locator(loc)
(8)                     self._locator = loc
(4)                 @property
(4)                 def minorlocator(self):
```

```
(8)         """Minor tick `.Locator` for the colorbar."""
(8)         return self._long_axis().get_minor_locator()
(4)     @minorlocator.setter
(4)     def minorlocator(self, loc):
(8)         self._long_axis().set_minor_locator(loc)
(8)         self._minorlocator = loc
(4)     @property
(4)     def formatter(self):
(8)         """Major tick label `.Formatter` for the colorbar."""
(8)         return self._long_axis().get_major_formatter()
(4)     @formatter.setter
(4)     def formatter(self, fmt):
(8)         self._long_axis().set_major_formatter(fmt)
(8)         self._formatter = fmt
(4)     @property
(4)     def minorformatter(self):
(8)         """Minor tick `.Formatter` for the colorbar."""
(8)         return self._long_axis().get_minor_formatter()
(4)     @minorformatter.setter
(4)     def minorformatter(self, fmt):
(8)         self._long_axis().set_minor_formatter(fmt)
(8)         self._minorformatter = fmt
(4)     def _cbar_cla(self):
(8)         """Function to clear the interactive colorbar state."""
(8)         for x in self._interactive_funcs:
(12)            delattr(self.ax, x)
(8)         del self.ax.cla
(8)         self.ax.cla()
(4)     def update_normal(self, mappable):
(8)         """
(8)         Update solid patches, lines, etc.
(8)         This is meant to be called when the norm of the image or contour plot
(8)         to which this colorbar belongs changes.
(8)         If the norm on the mappable is different than before, this resets the
(8)         locator and formatter for the axis, so if these have been customized,
(8)         they will need to be customized again.  However, if the norm only
(8)         changes values of *vmin*, *vmax* or *cmap* then the old formatter
(8)         and locator will be preserved.
(8)         """
(8)         _log.debug('colorbar update normal %r %r', mappable.norm, self.norm)
(8)         self.mappable = mappable
(8)         self.set_alpha(mappable.get_alpha())
(8)         self.cmap = mappable.cmap
(8)         if mappable.norm != self.norm:
(12)            self.norm = mappable.norm
(12)            self._reset_locator_formatter_scale()
(8)         self._draw_all()
(8)         if isinstance(self.mappable, contour.ContourSet):
(12)            CS = self.mappable
(12)            if not CS.filled:
(16)                self.add_lines(CS)
(8)         self.stale = True
(4)     def _draw_all(self):
(8)         """
(8)         Calculate any free parameters based on the current cmap and norm,
(8)         and do all the drawing.
(8)         """
(8)         if self.orientation == 'vertical':
(12)            if mpl.rcParams['ytick.minor.visible']:
(16)                self.minorticks_on()
(8)         else:
(12)            if mpl.rcParams['xtick.minor.visible']:
(16)                self.minorticks_on()
(8)         self._long_axis().set(label_position=self.ticklocation,
(30)                              ticks_position=self.ticklocation)
(8)         self._short_axis().set_ticks([])
(8)         self._short_axis().set_ticks([], minor=True)
(8)         self._process_values()
(8)         self.vmin, self.vmax = self._boundaries[self._inside][[0, -1]]
```

```
(8)                     X, Y = self._mesh()
(8)                     self._do_extends()
(8)                     lower, upper = self.vmin, self.vmax
(8)                     if self._long_axis().get_inverted():
(12)                        lower, upper = upper, lower
(8)                     if self.orientation == 'vertical':
(12)                        self.ax.set_xlim(0, 1)
(12)                        self.ax.set_ylim(lower, upper)
(8)                     else:
(12)                        self.ax.set_ylim(0, 1)
(12)                        self.ax.set_xlim(lower, upper)
(8)                     self.update_ticks()
(8)                     if self._filled:
(12)                        ind = np.arange(len(self._values))
(12)                        if self._extend_lower():
(16)                            ind = ind[1:]
(12)                        if self._extend_upper():
(16)                            ind = ind[:-1]
(12)                        self._add_solids(X, Y, self._values[ind, np.newaxis])
(4)                 def _add_solids(self, X, Y, C):
(8)                     """Draw the colors; optionally add separators."""
(8)                     if self.solids is not None:
(12)                        self.solids.remove()
(8)                     for solid in self.solids_patches:
(12)                        solid.remove()
(8)                     mappable = getattr(self, 'mappable', None)
(8)                     if (isinstance(mappable, contour.ContourSet)
(16)                            and any(hatch is not None for hatch in mappable.hatches)):
(12)                        self._add_solids_patches(X, Y, C, mappable)
(8)                     else:
(12)                        self.solids = self.ax.pcolormesh(
(16)                            X, Y, C, cmap=self.cmap, norm=self.norm, alpha=self.alpha,
(16)                            edgecolors='none', shading='flat')
(12)                        if not self.drawedges:
(16)                            if len(self._y) >= self.n_rasterize:
(20)                                self.solids.set_rasterized(True)
(8)                     self._update_dividers()
(4)                 def _update_dividers(self):
(8)                     if not self.drawedges:
(12)                        self.dividers.set_segments([])
(12)                        return
(8)                     if self.orientation == 'vertical':
(12)                        lims = self.ax.get_ylim()
(12)                        bounds = (lims[0] < self._y) & (self._y < lims[1])
(8)                     else:
(12)                        lims = self.ax.get_xlim()
(12)                        bounds = (lims[0] < self._y) & (self._y < lims[1])
(8)                     y = self._y[bounds]
(8)                     if self._extend_lower():
(12)                        y = np.insert(y, 0, lims[0])
(8)                     if self._extend_upper():
(12)                        y = np.append(y, lims[1])
(8)                     X, Y = np.meshgrid([0, 1], y)
(8)                     if self.orientation == 'vertical':
(12)                        segments = np.dstack([X, Y])
(8)                     else:
(12)                        segments = np.dstack([Y, X])
(8)                     self.dividers.set_segments(segments)
(4)                 def _add_solids_patches(self, X, Y, C, mappable):
(8)                     hatches = mappable.hatches * (len(C) + 1)  # Have enough hatches.
(8)                     if self._extend_lower():
(12)                        hatches = hatches[1:]
(8)                     patches = []
(8)                     for i in range(len(X) - 1):
(12)                        xy = np.array([[X[i, 0], Y[i, 1]],
(27)                                       [X[i, 1], Y[i, 0]],
(27)                                       [X[i + 1, 1], Y[i + 1, 0]],
(27)                                       [X[i + 1, 0], Y[i + 1, 1]]])
(12)                        patch = mpatches.PathPatch(mpath.Path(xy),
```

```
485: (39)                                                   facecolor=self.cmap(self.norm(C[i]
[0])),
486: (39)                                                   hatch=hatches[i], linewidth=0,
487: (39)                                                   antialiased=False, alpha=self.alpha)
488: (12)                        self.ax.add_patch(patch)
489: (12)                        patches.append(patch)
490: (8)                    self.solids_patches = patches
491: (4)                def _do_extends(self, ax=None):
492: (8)                    """
493: (8)                    Add the extend tri/rectangles on the outside of the Axes.
494: (8)                    ax is unused, but required due to the callbacks on xlim/ylim changed
495: (8)                    """
496: (8)                    for patch in self._extend_patches:
497: (12)                        patch.remove()
498: (8)                    self._extend_patches = []
499: (8)                    _, extendlen = self._proportional_y()
500: (8)                    bot = 0 - (extendlen[0] if self._extend_lower() else 0)
501: (8)                    top = 1 + (extendlen[1] if self._extend_upper() else 0)
502: (8)                    if not self.extendrect:
503: (12)                        xyout = np.array([[0, 0], [0.5, bot], [1, 0],
504: (30)                                          [1, 1], [0.5, top], [0, 1], [0, 0]])
505: (8)                    else:
506: (12)                        xyout = np.array([[0, 0], [0, bot], [1, bot], [1, 0],
507: (30)                                          [1, 1], [1, top], [0, top], [0, 1],
508: (30)                                          [0, 0]])
509: (8)                    if self.orientation == 'horizontal':
510: (12)                        xyout = xyout[:, ::-1]
511: (8)                    self.outline.set_xy(xyout)
512: (8)                    if not self._filled:
513: (12)                        return
514: (8)                    mappable = getattr(self, 'mappable', None)
515: (8)                    if (isinstance(mappable, contour.ContourSet)
516: (16)                            and any(hatch is not None for hatch in mappable.hatches)):
517: (12)                        hatches = mappable.hatches * (len(self._y) + 1)
518: (8)                    else:
519: (12)                        hatches = [None] * (len(self._y) + 1)
520: (8)                    if self._extend_lower():
521: (12)                        if not self.extendrect:
522: (16)                            xy = np.array([[0, 0], [0.5, bot], [1, 0]])
523: (12)                        else:
524: (16)                            xy = np.array([[0, 0], [0, bot], [1., bot], [1, 0]])
525: (12)                        if self.orientation == 'horizontal':
526: (16)                            xy = xy[:, ::-1]
527: (12)                        val = -1 if self._long_axis().get_inverted() else 0
528: (12)                        color = self.cmap(self.norm(self._values[val]))
529: (12)                        patch = mpatches.PathPatch(
530: (16)                            mpath.Path(xy), facecolor=color, alpha=self.alpha,
531: (16)                            linewidth=0, antialiased=False,
532: (16)                            transform=self.ax.transAxes,
533: (16)                            hatch=hatches[0], clip_on=False,
534: (16)                            zorder=np.nextafter(self.ax.patch.zorder, -np.inf))
535: (12)                        self.ax.add_patch(patch)
536: (12)                        self._extend_patches.append(patch)
537: (12)                        hatches = hatches[1:]
538: (8)                    if self._extend_upper():
539: (12)                        if not self.extendrect:
540: (16)                            xy = np.array([[0, 1], [0.5, top], [1, 1]])
541: (12)                        else:
542: (16)                            xy = np.array([[0, 1], [0, top], [1, top], [1, 1]])
543: (12)                        if self.orientation == 'horizontal':
544: (16)                            xy = xy[:, ::-1]
545: (12)                        val = 0 if self._long_axis().get_inverted() else -1
546: (12)                        color = self.cmap(self.norm(self._values[val]))
547: (12)                        hatch_idx = len(self._y) - 1
548: (12)                        patch = mpatches.PathPatch(
549: (16)                            mpath.Path(xy), facecolor=color, alpha=self.alpha,
550: (16)                            linewidth=0, antialiased=False,
551: (16)                            transform=self.ax.transAxes, hatch=hatches[hatch_idx],
552: (16)                            clip_on=False,
```

```
(16)                            zorder=np.nextafter(self.ax.patch.zorder, -np.inf))
(12)                        self.ax.add_patch(patch)
(12)                        self._extend_patches.append(patch)
(8)                     self._update_dividers()
(4)                 def add_lines(self, *args, **kwargs):
(8)                     """
(8)                     Draw lines on the colorbar.
(8)                     The lines are appended to the list :attr:`lines`.
(8)                     Parameters
(8)                     ----------
(8)                     levels : array-like
(12)                        The positions of the lines.
(8)                     colors : :mpltype:`color` or list of :mpltype:`color`
(12)                        Either a single color applying to all lines or one color value for
(12)                        each line.
(8)                     linewidths : float or array-like
(12)                        Either a single linewidth applying to all lines or one linewidth
(12)                        for each line.
(8)                     erase : bool, default: True
(12)                        Whether to remove any previously added lines.
(8)                     Notes
(8)                     -----
(8)                     Alternatively, this method can also be called with the signature
(8)                     ``colorbar.add_lines(contour_set, erase=True)``, in which case
(8)                     *levels*, *colors*, and *linewidths* are taken from *contour_set*.
(8)                     """
(8)                     params = _api.select_matching_signature(
(12)                        [lambda self, CS, erase=True: locals(),
(13)                         lambda self, levels, colors, linewidths, erase=True: locals()],
(12)                        self, *args, **kwargs)
(8)                     if "CS" in params:
(12)                        self, cs, erase = params.values()
(12)                        if not isinstance(cs, contour.ContourSet) or cs.filled:
(16)                            raise ValueError("If a single artist is passed to add_lines, "
(33)                                             "it must be a ContourSet of lines")
(12)                        return self.add_lines(
(16)                            cs.levels,
(16)                            cs.to_rgba(cs.cvalues, cs.alpha),
(16)                            cs.get_linewidths(),
(16)                            erase=erase)
(8)                     else:
(12)                        self, levels, colors, linewidths, erase = params.values()
(8)                     y = self._locate(levels)
(8)                     rtol = (self._y[-1] - self._y[0]) * 1e-10
(8)                     igood = (y < self._y[-1] + rtol) & (y > self._y[0] - rtol)
(8)                     y = y[igood]
(8)                     if np.iterable(colors):
(12)                        colors = np.asarray(colors)[igood]
(8)                     if np.iterable(linewidths):
(12)                        linewidths = np.asarray(linewidths)[igood]
(8)                     X, Y = np.meshgrid([0, 1], y)
(8)                     if self.orientation == 'vertical':
(12)                        xy = np.stack([X, Y], axis=-1)
(8)                     else:
(12)                        xy = np.stack([Y, X], axis=-1)
(8)                     col = collections.LineCollection(xy, linewidths=linewidths,
(41)                                                     colors=colors)
(8)                     if erase and self.lines:
(12)                        for lc in self.lines:
(16)                            lc.remove()
(12)                        self.lines = []
(8)                     self.lines.append(col)
(8)                     fac = np.max(linewidths) / 72
(8)                     xy = np.array([[0, 0], [1, 0], [1, 1], [0, 1], [0, 0]])
(8)                     inches = self.ax.get_figure().dpi_scale_trans
(8)                     xy = inches.inverted().transform(self.ax.transAxes.transform(xy))
(8)                     xy[[0, 1, 4], 1] -= fac
(8)                     xy[[2, 3], 1] += fac
(8)                     xy = self.ax.transAxes.inverted().transform(inches.transform(xy))
```

```
(8)                     col.set_clip_path(mpath.Path(xy, closed=True),
(26)                                      self.ax.transAxes)
(8)                     self.ax.add_collection(col)
(8)                     self.stale = True
(4)                 def update_ticks(self):
(8)                     """
(8)                     Set up the ticks and ticklabels. This should not be needed by users.
(8)                     """
(8)                     self._get_ticker_locator_formatter()
(8)                     self._long_axis().set_major_locator(self._locator)
(8)                     self._long_axis().set_minor_locator(self._minorlocator)
(8)                     self._long_axis().set_major_formatter(self._formatter)
(4)                 def _get_ticker_locator_formatter(self):
(8)                     """
(8)                     Return the ``locator`` and ``formatter`` of the colorbar.
(8)                     If they have not been defined (i.e. are *None*), the formatter and
(8)                     locator are retrieved from the axis, or from the value of the
(8)                     boundaries for a boundary norm.
(8)                     Called by update_ticks...
(8)                     """
(8)                     locator = self._locator
(8)                     formatter = self._formatter
(8)                     minorlocator = self._minorlocator
(8)                     if isinstance(self.norm, colors.BoundaryNorm):
(12)                        b = self.norm.boundaries
(12)                        if locator is None:
(16)                            locator = ticker.FixedLocator(b, nbins=10)
(12)                        if minorlocator is None:
(16)                            minorlocator = ticker.FixedLocator(b)
(8)                     elif isinstance(self.norm, colors.NoNorm):
(12)                        if locator is None:
(16)                            nv = len(self._values)
(16)                            base = 1 + int(nv / 10)
(16)                            locator = ticker.IndexLocator(base=base, offset=.5)
(8)                     elif self.boundaries is not None:
(12)                        b = self._boundaries[self._inside]
(12)                        if locator is None:
(16)                            locator = ticker.FixedLocator(b, nbins=10)
(8)                     else:  # most cases:
(12)                        if locator is None:
(16)                            locator = self._long_axis().get_major_locator()
(12)                        if minorlocator is None:
(16)                            minorlocator = self._long_axis().get_minor_locator()
(8)                     if minorlocator is None:
(12)                        minorlocator = ticker.NullLocator()
(8)                     if formatter is None:
(12)                        formatter = self._long_axis().get_major_formatter()
(8)                     self._locator = locator
(8)                     self._formatter = formatter
(8)                     self._minorlocator = minorlocator
(8)                     _log.debug('locator: %r', locator)
(4)                 def set_ticks(self, ticks, *, labels=None, minor=False, **kwargs):
(8)                     """
(8)                     Set tick locations.
(8)                     Parameters
(8)                     ----------
(8)                     ticks : 1D array-like
(12)                        List of tick locations.
(8)                     labels : list of str, optional
(12)                        List of tick labels. If not set, the labels show the data value.
(8)                     minor : bool, default: False
(12)                        If ``False``, set the major ticks; if ``True``, the minor ticks.
(8)                     **kwargs
(12)                        `.Text` properties for the labels. These take effect only if you
(12)                        pass *labels*. In other cases, please use `~.Axes.tick_params`.
(8)                     """
(8)                     if np.iterable(ticks):
(12)                        self._long_axis().set_ticks(ticks, labels=labels, minor=minor,
(40)                                                    **kwargs)
```

```
(12)                        self._locator = self._long_axis().get_major_locator()
(8)                     else:
(12)                        self._locator = ticks
(12)                        self._long_axis().set_major_locator(self._locator)
(8)                     self.stale = True
(4)                 def get_ticks(self, minor=False):
(8)                     """
(8)                     Return the ticks as a list of locations.
(8)                     Parameters
(8)                     ----------
(8)                     minor : boolean, default: False
(12)                        if True return the minor ticks.
(8)                     """
(8)                     if minor:
(12)                        return self._long_axis().get_minorticklocs()
(8)                     else:
(12)                        return self._long_axis().get_majorticklocs()
(4)                 def set_ticklabels(self, ticklabels, *, minor=False, **kwargs):
(8)                     """
(8)                     [*Discouraged*] Set tick labels.
(8)                     .. admonition:: Discouraged
(12)                        The use of this method is discouraged, because of the dependency
(12)                        on tick positions. In most cases, you'll want to use
(12)                        ``set_ticks(positions, labels=labels)`` instead.
(12)                        If you are using this method, you should always fix the tick
(12)                        positions before, e.g. by using `.Colorbar.set_ticks` or by
(12)                        explicitly setting a `~.ticker.FixedLocator` on the long axis
(12)                        of the colorbar. Otherwise, ticks are free to move and the
(12)                        labels may end up in unexpected positions.
(8)                     Parameters
(8)                     ----------
(8)                     ticklabels : sequence of str or of `.Text`
(12)                        Texts for labeling each tick location in the sequence set by
(12)                        `.Colorbar.set_ticks`; the number of labels must match the number
(12)                        of locations.
(8)                     update_ticks : bool, default: True
(12)                        This keyword argument is ignored and will be removed.
(12)                        Deprecated
(8)                     minor : bool
(12)                        If True, set minor ticks instead of major ticks.
(8)                     **kwargs
(12)                        `.Text` properties for the labels.
(8)                     """
(8)                     self._long_axis().set_ticklabels(ticklabels, minor=minor, **kwargs)
(4)                 def minorticks_on(self):
(8)                     """
(8)                     Turn on colorbar minor ticks.
(8)                     """
(8)                     self.ax.minorticks_on()
(8)                     self._short_axis().set_minor_locator(ticker.NullLocator())
(4)                 def minorticks_off(self):
(8)                     """Turn the minor ticks of the colorbar off."""
(8)                     self._minorlocator = ticker.NullLocator()
(8)                     self._long_axis().set_minor_locator(self._minorlocator)
(4)                 def set_label(self, label, *, loc=None, **kwargs):
(8)                     """
(8)                     Add a label to the long axis of the colorbar.
(8)                     Parameters
(8)                     ----------
(8)                     label : str
(12)                        The label text.
(8)                     loc : str, optional
(12)                        The location of the label.
(12)                        - For horizontal orientation one of {'left', 'center', 'right'}
(12)                        - For vertical orientation one of {'bottom', 'center', 'top'}
(12)                        Defaults to :rc:`xaxis.labellocation` or :rc:`yaxis.labellocation`
(12)                        depending on the orientation.
(8)                     **kwargs
(12)                        Keyword arguments are passed to `~.Axes.set_xlabel` /
```

```
760: (12)                        `~.Axes.set_ylabel`.
761: (12)                        Supported keywords are *labelpad* and `.Text` properties.
762: (8)                    """
763: (8)                    if self.orientation == "vertical":
764: (12)                        self.ax.set_ylabel(label, loc=loc, **kwargs)
765: (8)                    else:
766: (12)                        self.ax.set_xlabel(label, loc=loc, **kwargs)
767: (8)                    self.stale = True
768: (4)                def set_alpha(self, alpha):
769: (8)                    """
770: (8)                    Set the transparency between 0 (transparent) and 1 (opaque).
771: (8)                    If an array is provided, *alpha* will be set to None to use the
772: (8)                    transparency values associated with the colormap.
773: (8)                    """
774: (8)                    self.alpha = None if isinstance(alpha, np.ndarray) else alpha
775: (4)                def _set_scale(self, scale, **kwargs):
776: (8)                    """
777: (8)                    Set the colorbar long axis scale.
778: (8)                    Parameters
779: (8)                    ----------
780: (8)                    scale : {"linear", "log", "symlog", "logit", ...} or `.ScaleBase`
781: (12)                        The axis scale type to apply.
782: (8)                    **kwargs
783: (12)                        Different keyword arguments are accepted, depending on the scale.
784: (12)                        See the respective class keyword arguments:
785: (12)                        - `matplotlib.scale.LinearScale`
786: (12)                        - `matplotlib.scale.LogScale`
787: (12)                        - `matplotlib.scale.SymmetricalLogScale`
788: (12)                        - `matplotlib.scale.LogitScale`
789: (12)                        - `matplotlib.scale.FuncScale`
790: (12)                        - `matplotlib.scale.AsinhScale`
791: (8)                    Notes
792: (8)                    -----
793: (8)                    By default, Matplotlib supports the above-mentioned scales.
794: (8)                    Additionally, custom scales may be registered using
795: (8)                    `matplotlib.scale.register_scale`. These scales can then also
796: (8)                    be used here.
797: (8)                    """
798: (8)                    self._long_axis()._set_axes_scale(scale, **kwargs)
799: (4)                def remove(self):
800: (8)                    """
801: (8)                    Remove this colorbar from the figure.
802: (8)                    If the colorbar was created with ``use_gridspec=True`` the previous
803: (8)                    gridspec is restored.
804: (8)                    """
805: (8)                    if hasattr(self.ax, '_colorbar_info'):
806: (12)                        parents = self.ax._colorbar_info['parents']
807: (12)                        for a in parents:
808: (16)                            if self.ax in a._colorbars:
809: (20)                                a._colorbars.remove(self.ax)
810: (8)                    self.ax.remove()
811: (8)                    self.mappable.callbacks.disconnect(self.mappable.colorbar_cid)
812: (8)                    self.mappable.colorbar = None
813: (8)                    self.mappable.colorbar_cid = None
814: (8)                    self.ax.callbacks.disconnect(self._extend_cid1)
815: (8)                    self.ax.callbacks.disconnect(self._extend_cid2)
816: (8)                    try:
817: (12)                        ax = self.mappable.axes
818: (8)                    except AttributeError:
819: (12)                        return
820: (8)                    try:
821: (12)                        subplotspec =
self.ax.get_subplotspec().get_gridspec()._subplot_spec
822: (8)                    except AttributeError:  # use_gridspec was False
823: (12)                        pos = ax.get_position(original=True)
824: (12)                        ax._set_position(pos)
825: (8)                    else:  # use_gridspec was True
826: (12)                        ax.set_subplotspec(subplotspec)
827: (4)                def _process_values(self):
```

```
(8)                 """
(8)                 Set `_boundaries` and `_values` based on the self.boundaries and
(8)                 self.values if not None, or based on the size of the colormap and
(8)                 the vmin/vmax of the norm.
(8)                 """
(8)                 if self.values is not None:
(12)                    self._values = np.array(self.values)
(12)                    if self.boundaries is None:
(16)                        b = np.zeros(len(self.values) + 1)
(16)                        b[1:-1] = 0.5 * (self._values[:-1] + self._values[1:])
(16)                        b[0] = 2.0 * b[1] - b[2]
(16)                        b[-1] = 2.0 * b[-2] - b[-3]
(16)                        self._boundaries = b
(16)                        return
(12)                    self._boundaries = np.array(self.boundaries)
(12)                    return
(8)                 if isinstance(self.norm, colors.BoundaryNorm):
(12)                    b = self.norm.boundaries
(8)                 elif isinstance(self.norm, colors.NoNorm):
(12)                    b = np.arange(self.cmap.N + 1) - .5
(8)                 elif self.boundaries is not None:
(12)                    b = self.boundaries
(8)                 else:
(12)                    N = self.cmap.N + 1
(12)                    b, _ = self._uniform_y(N)
(8)                 if self._extend_lower():
(12)                    b = np.hstack((b[0] - 1, b))
(8)                 if self._extend_upper():
(12)                    b = np.hstack((b, b[-1] + 1))
(8)                 if self.mappable.get_array() is not None:
(12)                    self.mappable.autoscale_None()
(8)                 if not self.norm.scaled():
(12)                    self.norm.vmin = 0
(12)                    self.norm.vmax = 1
(8)                 self.norm.vmin, self.norm.vmax = mtransforms.nonsingular(
(12)                    self.norm.vmin, self.norm.vmax, expander=0.1)
(8)                 if (not isinstance(self.norm, colors.BoundaryNorm) and
(16)                        (self.boundaries is None)):
(12)                    b = self.norm.inverse(b)
(8)                 self._boundaries = np.asarray(b, dtype=float)
(8)                 self._values = 0.5 * (self._boundaries[:-1] + self._boundaries[1:])
(8)                 if isinstance(self.norm, colors.NoNorm):
(12)                    self._values = (self._values + 0.00001).astype(np.int16)
(4)             def _mesh(self):
(8)                 """
(8)                 Return the coordinate arrays for the colorbar pcolormesh/patches.
(8)                 These are scaled between vmin and vmax, and already handle colorbar
(8)                 orientation.
(8)                 """
(8)                 y, _ = self._proportional_y()
(8)                 if (isinstance(self.norm, (colors.BoundaryNorm, colors.NoNorm))
(16)                        or self.boundaries is not None):
(12)                    y = y * (self.vmax - self.vmin) + self.vmin
(8)                 else:
(12)                    with self.norm.callbacks.blocked(), \
(20)                            cbook._setattr_cm(self.norm,
(38)                                              vmin=self.vmin,
(38)                                              vmax=self.vmax):
(16)                        y = self.norm.inverse(y)
(8)                 self._y = y
(8)                 X, Y = np.meshgrid([0., 1.], y)
(8)                 if self.orientation == 'vertical':
(12)                    return (X, Y)
(8)                 else:
(12)                    return (Y, X)
(4)             def _forward_boundaries(self, x):
(8)                 b = self._boundaries
(8)                 y = np.interp(x, b, np.linspace(0, 1, len(b)))
(8)                 eps = (b[-1] - b[0]) * 1e-6
```

```
(8)                     y[x < b[0]-eps] = -1
(8)                     y[x > b[-1]+eps] = 2
(8)                     return y
(4)                 def _inverse_boundaries(self, x):
(8)                     b = self._boundaries
(8)                     return np.interp(x, np.linspace(0, 1, len(b)), b)
(4)                 def _reset_locator_formatter_scale(self):
(8)                     """
(8)                     Reset the locator et al to defaults.  Any user-hardcoded changes
(8)                     need to be re-entered if this gets called (either at init, or when
(8)                     the mappable normal gets changed: Colorbar.update_normal)
(8)                     """
(8)                     self._process_values()
(8)                     self._locator = None
(8)                     self._minorlocator = None
(8)                     self._formatter = None
(8)                     self._minorformatter = None
(8)                     if (isinstance(self.mappable, contour.ContourSet) and
(16)                            isinstance(self.norm, colors.LogNorm)):
(12)                        self._set_scale('log')
(8)                     elif (self.boundaries is not None or
(16)                            isinstance(self.norm, colors.BoundaryNorm)):
(12)                        if self.spacing == 'uniform':
(16)                            funcs = (self._forward_boundaries, self._inverse_boundaries)
(16)                            self._set_scale('function', functions=funcs)
(12)                        elif self.spacing == 'proportional':
(16)                            self._set_scale('linear')
(8)                     elif getattr(self.norm, '_scale', None):
(12)                        self._set_scale(self.norm._scale)
(8)                     elif type(self.norm) is colors.Normalize:
(12)                        self._set_scale('linear')
(8)                     else:
(12)                        funcs = (self.norm, self.norm.inverse)
(12)                        self._set_scale('function', functions=funcs)
(4)                 def _locate(self, x):
(8)                     """
(8)                     Given a set of color data values, return their
(8)                     corresponding colorbar data coordinates.
(8)                     """
(8)                     if isinstance(self.norm, (colors.NoNorm, colors.BoundaryNorm)):
(12)                        b = self._boundaries
(12)                        xn = x
(8)                     else:
(12)                        b = self.norm(self._boundaries, clip=False).filled()
(12)                        xn = self.norm(x, clip=False).filled()
(8)                     bunique = b[self._inside]
(8)                     yunique = self._y
(8)                     z = np.interp(xn, bunique, yunique)
(8)                     return z
(4)                 def _uniform_y(self, N):
(8)                     """
(8)                     Return colorbar data coordinates for *N* uniformly
(8)                     spaced boundaries, plus extension lengths if required.
(8)                     """
(8)                     automin = automax = 1. / (N - 1.)
(8)                     extendlength = self._get_extension_lengths(self.extendfrac,
(51)                                                               automin, automax,
(51)                                                               default=0.05)
(8)                     y = np.linspace(0, 1, N)
(8)                     return y, extendlength
(4)                 def _proportional_y(self):
(8)                     """
(8)                     Return colorbar data coordinates for the boundaries of
(8)                     a proportional colorbar, plus extension lengths if required:
(8)                     """
(8)                     if (isinstance(self.norm, colors.BoundaryNorm) or
(16)                            self.boundaries is not None):
(12)                        y = (self._boundaries - self._boundaries[self._inside][0])
(12)                        y = y / (self._boundaries[self._inside][-1] -
```

```
(21)                                self._boundaries[self._inside][0])
(12)                        if self.spacing == 'uniform':
(16)                            yscaled = self._forward_boundaries(self._boundaries)
(12)                        else:
(16)                            yscaled = y
(8)                     else:
(12)                        y = self.norm(self._boundaries.copy())
(12)                        y = np.ma.filled(y, np.nan)
(12)                        yscaled = y
(8)                     y = y[self._inside]
(8)                     yscaled = yscaled[self._inside]
(8)                     norm = colors.Normalize(y[0], y[-1])
(8)                     y = np.ma.filled(norm(y), np.nan)
(8)                     norm = colors.Normalize(yscaled[0], yscaled[-1])
(8)                     yscaled = np.ma.filled(norm(yscaled), np.nan)
(8)                     automin = yscaled[1] - yscaled[0]
(8)                     automax = yscaled[-1] - yscaled[-2]
(8)                     extendlength = [0, 0]
(8)                     if self._extend_lower() or self._extend_upper():
(12)                        extendlength = self._get_extension_lengths(
(20)                                self.extendfrac, automin, automax, default=0.05)
(8)                     return y, extendlength
(4)                 def _get_extension_lengths(self, frac, automin, automax, default=0.05):
(8)                     """
(8)                     Return the lengths of colorbar extensions.
(8)                     This is a helper method for _uniform_y and _proportional_y.
(8)                     """
(8)                     extendlength = np.array([default, default])
(8)                     if isinstance(frac, str):
(12)                        _api.check_in_list(['auto'], extendfrac=frac.lower())
(12)                        extendlength[:] = [automin, automax]
(8)                     elif frac is not None:
(12)                        try:
(16)                            extendlength[:] = frac
(16)                            if np.isnan(extendlength).any():
(20)                                raise ValueError()
(12)                        except (TypeError, ValueError) as err:
(16)                            raise ValueError('invalid value for extendfrac') from err
(8)                     return extendlength
(4)                 def _extend_lower(self):
(8)                     """Return whether the lower limit is open ended."""
(8)                     minmax = "max" if self._long_axis().get_inverted() else "min"
(8)                     return self.extend in ('both', minmax)
(4)                 def _extend_upper(self):
(8)                     """Return whether the upper limit is open ended."""
(8)                     minmax = "min" if self._long_axis().get_inverted() else "max"
(8)                     return self.extend in ('both', minmax)
(4)                 def _long_axis(self):
(8)                     """Return the long axis"""
(8)                     if self.orientation == 'vertical':
(12)                        return self.ax.yaxis
(8)                     return self.ax.xaxis
(4)                 def _short_axis(self):
(8)                     """Return the short axis"""
(8)                     if self.orientation == 'vertical':
(12)                        return self.ax.xaxis
(8)                     return self.ax.yaxis
(4)                 def _get_view(self):
(8)                     return self.norm.vmin, self.norm.vmax
(4)                 def _set_view(self, view):
(8)                     self.norm.vmin, self.norm.vmax = view
(4)                 def _set_view_from_bbox(self, bbox, direction='in',
(28)                                        mode=None, twinx=False, twiny=False):
(8)                     new_xbound, new_ybound = self.ax._prepare_view_from_bbox(
(12)                        bbox, direction=direction, mode=mode, twinx=twinx, twiny=twiny)
(8)                     if self.orientation == 'horizontal':
(12)                        self.norm.vmin, self.norm.vmax = new_xbound
(8)                     elif self.orientation == 'vertical':
(12)                        self.norm.vmin, self.norm.vmax = new_ybound
```

```
(4)                 def drag_pan(self, button, key, x, y):
(8)                     points = self.ax._get_pan_points(button, key, x, y)
(8)                     if points is not None:
(12)                        if self.orientation == 'horizontal':
(16)                            self.norm.vmin, self.norm.vmax = points[:, 0]
(12)                        elif self.orientation == 'vertical':
(16)                            self.norm.vmin, self.norm.vmax = points[:, 1]
(0)             ColorbarBase = Colorbar  # Backcompat API
(0)             def _normalize_location_orientation(location, orientation):
(4)                 if location is None:
(8)                     location = _get_ticklocation_from_orientation(orientation)
(4)                 loc_settings = _api.check_getitem({
(8)                     "left":   {"location": "left", "anchor": (1.0, 0.5),
(19)                               "panchor": (0.0, 0.5), "pad": 0.10},
(8)                     "right":  {"location": "right", "anchor": (0.0, 0.5),
(19)                               "panchor": (1.0, 0.5), "pad": 0.05},
(8)                     "top":    {"location": "top", "anchor": (0.5, 0.0),
(19)                               "panchor": (0.5, 1.0), "pad": 0.05},
(8)                     "bottom": {"location": "bottom", "anchor": (0.5, 1.0),
(19)                               "panchor": (0.5, 0.0), "pad": 0.15},
(4)                 }, location=location)
(4)                 loc_settings["orientation"] = _get_orientation_from_location(location)
(4)                 if orientation is not None and orientation != loc_settings["orientation"]:
(8)                     raise TypeError("location and orientation are mutually exclusive")
(4)                 return loc_settings
(0)             def _get_orientation_from_location(location):
(4)                 return _api.check_getitem(
(8)                     {None: None, "left": "vertical", "right": "vertical",
(9)                      "top": "horizontal", "bottom": "horizontal"}, location=location)
(0)             def _get_ticklocation_from_orientation(orientation):
(4)                 return _api.check_getitem(
(8)                     {None: "right", "vertical": "right", "horizontal": "bottom"},
(8)                     orientation=orientation)
(0)             @_docstring.interpd
(0)             def make_axes(parents, location=None, orientation=None, fraction=0.15,
(14)                          shrink=1.0, aspect=20, **kwargs):
(4)                 """
(4)                 Create an `~.axes.Axes` suitable for a colorbar.
(4)                 The Axes is placed in the figure of the *parents* Axes, by resizing and
(4)                 repositioning *parents*.
(4)                 Parameters
(4)                 ----------
(4)                 parents : `~matplotlib.axes.Axes` or iterable or `numpy.ndarray` of
`~.axes.Axes`
(8)                     The Axes to use as parents for placing the colorbar.
(4)                 %(_make_axes_kw_doc)s
(4)                 Returns
(4)                 -------
(4)                 cax : `~matplotlib.axes.Axes`
(8)                     The child Axes.
(4)                 kwargs : dict
(8)                     The reduced keyword dictionary to be passed when creating the colorbar
(8)                     instance.
(4)                 """
(4)                 loc_settings = _normalize_location_orientation(location, orientation)
(4)                 kwargs['orientation'] = loc_settings['orientation']
(4)                 location = kwargs['ticklocation'] = loc_settings['location']
(4)                 anchor = kwargs.pop('anchor', loc_settings['anchor'])
(4)                 panchor = kwargs.pop('panchor', loc_settings['panchor'])
(4)                 aspect0 = aspect
(4)                 if isinstance(parents, np.ndarray):
(8)                     parents = list(parents.flat)
(4)                 elif np.iterable(parents):
(8)                     parents = list(parents)
(4)                 else:
(8)                     parents = [parents]
(4)                 fig = parents[0].get_figure()
(4)                 pad0 = 0.05 if fig.get_constrained_layout() else loc_settings['pad']
(4)                 pad = kwargs.pop('pad', pad0)
```

```
(4)             if not all(fig is ax.get_figure() for ax in parents):
(8)                 raise ValueError('Unable to create a colorbar Axes as not all '
(25)                                 'parents share the same figure.')
(4)             parents_bbox = mtransforms.Bbox.union(
(8)                 [ax.get_position(original=True).frozen() for ax in parents])
(4)             pb = parents_bbox
(4)             if location in ('left', 'right'):
(8)                 if location == 'left':
(12)                    pbcb, _, pb1 = pb.splitx(fraction, fraction + pad)
(8)                 else:
(12)                    pb1, _, pbcb = pb.splitx(1 - fraction - pad, 1 - fraction)
(8)                 pbcb = pbcb.shrunk(1.0, shrink).anchored(anchor, pbcb)
(4)             else:
(8)                 if location == 'bottom':
(12)                    pbcb, _, pb1 = pb.splity(fraction, fraction + pad)
(8)                 else:
(12)                    pb1, _, pbcb = pb.splity(1 - fraction - pad, 1 - fraction)
(8)                 pbcb = pbcb.shrunk(shrink, 1.0).anchored(anchor, pbcb)
(8)                 aspect = 1.0 / aspect
(4)             shrinking_trans = mtransforms.BboxTransform(parents_bbox, pb1)
(4)             for ax in parents:
(8)                 new_posn = shrinking_trans.transform(ax.get_position(original=True))
(8)                 new_posn = mtransforms.Bbox(new_posn)
(8)                 ax._set_position(new_posn)
(8)                 if panchor is not False:
(12)                    ax.set_anchor(panchor)
(4)             cax = fig.add_axes(pbcb, label="<colorbar>")
(4)             for a in parents:
(8)                 a._colorbars += [cax]
(4)             cax._colorbar_info = dict(
(8)                 parents=parents,
(8)                 location=location,
(8)                 shrink=shrink,
(8)                 anchor=anchor,
(8)                 panchor=panchor,
(8)                 fraction=fraction,
(8)                 aspect=aspect0,
(8)                 pad=pad)
(4)             cax.set_anchor(anchor)
(4)             cax.set_box_aspect(aspect)
(4)             cax.set_aspect('auto')
(4)             return cax, kwargs
(0)         @_docstring.interpd
(0)         def make_axes_gridspec(parent, *, location=None, orientation=None,
(23)                               fraction=0.15, shrink=1.0, aspect=20, **kwargs):
(4)             """
(4)             Create an `~.axes.Axes` suitable for a colorbar.
(4)             The Axes is placed in the figure of the *parent* Axes, by resizing and
(4)             repositioning *parent*.
(4)             This function is similar to `.make_axes` and mostly compatible with it.
(4)             Primary differences are
(4)             - `.make_axes_gridspec` requires the *parent* to have a subplotspec.
(4)             - `.make_axes` positions the Axes in figure coordinates;
(6)               `.make_axes_gridspec` positions it using a subplotspec.
(4)             - `.make_axes` updates the position of the parent.  `.make_axes_gridspec`
(6)               replaces the parent gridspec with a new one.
(4)             Parameters
(4)             ----------
(4)             parent : `~matplotlib.axes.Axes`
(8)                 The Axes to use as parent for placing the colorbar.
(4)             %(_make_axes_kw_doc)s
(4)             Returns
(4)             -------
(4)             cax : `~matplotlib.axes.Axes`
(8)                 The child Axes.
(4)             kwargs : dict
(8)                 The reduced keyword dictionary to be passed when creating the colorbar
(8)                 instance.
(4)             """
```

```
1172: (4)               loc_settings = _normalize_location_orientation(location, orientation)
1173: (4)               kwargs['orientation'] = loc_settings['orientation']
1174: (4)               location = kwargs['ticklocation'] = loc_settings['location']
1175: (4)               aspect0 = aspect
1176: (4)               anchor = kwargs.pop('anchor', loc_settings['anchor'])
1177: (4)               panchor = kwargs.pop('panchor', loc_settings['panchor'])
1178: (4)               pad = kwargs.pop('pad', loc_settings["pad"])
1179: (4)               wh_space = 2 * pad / (1 - pad)
1180: (4)               if location in ('left', 'right'):
1181: (8)                   gs = parent.get_subplotspec().subgridspec(
1182: (12)                      3, 2, wspace=wh_space, hspace=0,
1183: (12)                      height_ratios=[(1-anchor[1])*(1-shrink), shrink, anchor[1]*(1-
shrink)])
1184: (8)                   if location == 'left':
1185: (12)                      gs.set_width_ratios([fraction, 1 - fraction - pad])
1186: (12)                      ss_main = gs[:, 1]
1187: (12)                      ss_cb = gs[1, 0]
1188: (8)                   else:
1189: (12)                      gs.set_width_ratios([1 - fraction - pad, fraction])
1190: (12)                      ss_main = gs[:, 0]
1191: (12)                      ss_cb = gs[1, 1]
1192: (4)               else:
1193: (8)                   gs = parent.get_subplotspec().subgridspec(
1194: (12)                      2, 3, hspace=wh_space, wspace=0,
1195: (12)                      width_ratios=[anchor[0]*(1-shrink), shrink, (1-anchor[0])*(1-
shrink)])
1196: (8)                   if location == 'top':
1197: (12)                      gs.set_height_ratios([fraction, 1 - fraction - pad])
1198: (12)                      ss_main = gs[1, :]
1199: (12)                      ss_cb = gs[0, 1]
1200: (8)                   else:
1201: (12)                      gs.set_height_ratios([1 - fraction - pad, fraction])
1202: (12)                      ss_main = gs[0, :]
1203: (12)                      ss_cb = gs[1, 1]
1204: (8)                   aspect = 1 / aspect
1205: (4)               parent.set_subplotspec(ss_main)
1206: (4)               if panchor is not False:
1207: (8)                   parent.set_anchor(panchor)
1208: (4)               fig = parent.get_figure()
1209: (4)               cax = fig.add_subplot(ss_cb, label="<colorbar>")
1210: (4)               cax.set_anchor(anchor)
1211: (4)               cax.set_box_aspect(aspect)
1212: (4)               cax.set_aspect('auto')
1213: (4)               cax._colorbar_info = dict(
1214: (8)                   location=location,
1215: (8)                   parents=[parent],
1216: (8)                   shrink=shrink,
1217: (8)                   anchor=anchor,
1218: (8)                   panchor=panchor,
1219: (8)                   fraction=fraction,
1220: (8)                   aspect=aspect0,
1221: (8)                   pad=pad)
1222: (4)               return cax, kwargs
```

File 33 - colors.py:

```
1: (0)              """
2: (0)              A module for converting numbers or color arguments to *RGB* or *RGBA*.
3: (0)              *RGB* and *RGBA* are sequences of, respectively, 3 or 4 floats in the
4: (0)              range 0-1.
5: (0)              This module includes functions and classes for color specification
conversions,
6: (0)              and for mapping numbers to colors in a 1-D array of colors called a colormap.
7: (0)              Mapping data onto colors using a colormap typically involves two steps: a data
8: (0)              array is first mapped onto the range 0-1 using a subclass of `Normalize`,
9: (0)              then this number is mapped to a color using a subclass of `Colormap`.  Two
10: (0)             subclasses of `Colormap` provided here:  `LinearSegmentedColormap`, which uses
```

```
 11: (0)             piecewise-linear interpolation to define colormaps, and `ListedColormap`,
 which
 12: (0)             makes a colormap from a list of colors.
 13: (0)             .. seealso::
 14: (2)               :ref:`colormap-manipulation` for examples of how to
 15: (2)               make colormaps and
 16: (2)               :ref:`colormaps` for a list of built-in colormaps.
 17: (2)               :ref:`colormapnorms` for more details about data
 18: (2)               normalization
 19: (2)               More colormaps are available at palettable_.
 20: (0)             The module also provides functions for checking whether an object can be
 21: (0)             interpreted as a color (`is_color_like`), for converting such an object
 22: (0)             to an RGBA tuple (`to_rgba`) or to an HTML-like hex string in the
 23: (0)             "#rrggbb" format (`to_hex`), and a sequence of colors to an (n, 4)
 24: (0)             RGBA array (`to_rgba_array`).  Caching is used for efficiency.
 25: (0)             Colors that Matplotlib recognizes are listed at
 26: (0)             :ref:`colors_def`.
 27: (0)             .. _palettable: https://jiffyclub.github.io/palettable/
 28: (0)             .. _xkcd color survey: https://xkcd.com/color/rgb/
 29: (0)             """
 30: (0)             import base64
 31: (0)             from collections.abc import Sized, Sequence, Mapping
 32: (0)             import functools
 33: (0)             import importlib
 34: (0)             import inspect
 35: (0)             import io
 36: (0)             import itertools
 37: (0)             from numbers import Real
 38: (0)             import re
 39: (0)             from PIL import Image
 40: (0)             from PIL.PngImagePlugin import PngInfo
 41: (0)             import matplotlib as mpl
 42: (0)             import numpy as np
 43: (0)             from matplotlib import _api, _cm, cbook, scale
 44: (0)             from ._color_data import BASE_COLORS, TABLEAU_COLORS, CSS4_COLORS, XKCD_COLORS
 45: (0)             class _ColorMapping(dict):
 46: (4)                 def __init__(self, mapping):
 47: (8)                     super().__init__(mapping)
 48: (8)                     self.cache = {}
 49: (4)                 def __setitem__(self, key, value):
 50: (8)                     super().__setitem__(key, value)
 51: (8)                     self.cache.clear()
 52: (4)                 def __delitem__(self, key):
 53: (8)                     super().__delitem__(key)
 54: (8)                     self.cache.clear()
 55: (0)             _colors_full_map = {}
 56: (0)             _colors_full_map.update(XKCD_COLORS)
 57: (0)             _colors_full_map.update({k.replace('grey', 'gray'): v
 58: (25)                                     for k, v in XKCD_COLORS.items()
 59: (25)                                     if 'grey' in k})
 60: (0)             _colors_full_map.update(CSS4_COLORS)
 61: (0)             _colors_full_map.update(TABLEAU_COLORS)
 62: (0)             _colors_full_map.update({k.replace('gray', 'grey'): v
 63: (25)                                     for k, v in TABLEAU_COLORS.items()
 64: (25)                                     if 'gray' in k})
 65: (0)             _colors_full_map.update(BASE_COLORS)
 66: (0)             _colors_full_map = _ColorMapping(_colors_full_map)
 67: (0)             _REPR_PNG_SIZE = (512, 64)
 68: (0)             def get_named_colors_mapping():
 69: (4)                 """Return the global mapping of names to named colors."""
 70: (4)                 return _colors_full_map
 71: (0)             class ColorSequenceRegistry(Mapping):
 72: (4)                 r"""
 73: (4)                 Container for sequences of colors that are known to Matplotlib by name.
 74: (4)                 The universal registry instance is `matplotlib.color_sequences`. There
 75: (4)                 should be no need for users to instantiate `.ColorSequenceRegistry`
 76: (4)                 themselves.
 77: (4)                 Read access uses a dict-like interface mapping names to lists of colors::
 78: (8)                     import matplotlib as mpl
```

```
79: (8)                     cmap = mpl.color_sequences['tab10']
80: (4)                 The returned lists are copies, so that their modification does not change
81: (4)                 the global definition of the color sequence.
82: (4)                 Additional color sequences can be added via
83: (4)                 `.ColorSequenceRegistry.register`::
84: (8)                     mpl.color_sequences.register('rgb', ['r', 'g', 'b'])
85: (4)                 """
86: (4)                 _BUILTIN_COLOR_SEQUENCES = {
87: (8)                     'tab10': _cm._tab10_data,
88: (8)                     'tab20': _cm._tab20_data,
89: (8)                     'tab20b': _cm._tab20b_data,
90: (8)                     'tab20c': _cm._tab20c_data,
91: (8)                     'Pastel1': _cm._Pastel1_data,
92: (8)                     'Pastel2': _cm._Pastel2_data,
93: (8)                     'Paired': _cm._Paired_data,
94: (8)                     'Accent': _cm._Accent_data,
95: (8)                     'Dark2': _cm._Dark2_data,
96: (8)                     'Set1': _cm._Set1_data,
97: (8)                     'Set2': _cm._Set2_data,
98: (8)                     'Set3': _cm._Set3_data,
99: (4)                 }
100: (4)                def __init__(self):
101: (8)                    self._color_sequences = {**self._BUILTIN_COLOR_SEQUENCES}
102: (4)                def __getitem__(self, item):
103: (8)                    try:
104: (12)                       return list(self._color_sequences[item])
105: (8)                    except KeyError:
106: (12)                       raise KeyError(f"{item!r} is not a known color sequence name")
107: (4)                def __iter__(self):
108: (8)                    return iter(self._color_sequences)
109: (4)                def __len__(self):
110: (8)                    return len(self._color_sequences)
111: (4)                def __str__(self):
112: (8)                    return ('ColorSequenceRegistry; available colormaps:\n' +
113: (16)                           ', '.join(f"'{name}'" for name in self))
114: (4)                def register(self, name, color_list):
115: (8)                    """
116: (8)                    Register a new color sequence.
117: (8)                    The color sequence registry stores a copy of the given *color_list*,
so
118: (8)                    that future changes to the original list do not affect the registered
119: (8)                    color sequence. Think of this as the registry taking a snapshot
120: (8)                    of *color_list* at registration.
121: (8)                    Parameters
122: (8)                    ----------
123: (8)                    name : str
124: (12)                       The name for the color sequence.
125: (8)                    color_list : list of :mpltype:`color`
126: (12)                       An iterable returning valid Matplotlib colors when iterating over.
127: (12)                       Note however that the returned color sequence will always be a
128: (12)                       list regardless of the input type.
129: (8)                    """
130: (8)                    if name in self._BUILTIN_COLOR_SEQUENCES:
131: (12)                       raise ValueError(f"{name!r} is a reserved name for a builtin "
132: (29)                                        "color sequence")
133: (8)                    color_list = list(color_list)  # force copy and coerce type to list
134: (8)                    for color in color_list:
135: (12)                       try:
136: (16)                           to_rgba(color)
137: (12)                       except ValueError:
138: (16)                           raise ValueError(
139: (20)                               f"{color!r} is not a valid color specification")
140: (8)                    self._color_sequences[name] = color_list
141: (4)                def unregister(self, name):
142: (8)                    """
143: (8)                    Remove a sequence from the registry.
144: (8)                    You cannot remove built-in color sequences.
145: (8)                    If the name is not registered, returns with no error.
146: (8)                    """
```

```
(8)                     if name in self._BUILTIN_COLOR_SEQUENCES:
(12)                        raise ValueError(
(16)                            f"Cannot unregister builtin color sequence {name!r}")
(8)                     self._color_sequences.pop(name, None)
(0)             _color_sequences = ColorSequenceRegistry()
(0)             def _sanitize_extrema(ex):
(4)                 if ex is None:
(8)                     return ex
(4)                 try:
(8)                     ret = ex.item()
(4)                 except AttributeError:
(8)                     ret = float(ex)
(4)                 return ret
(0)             _nth_color_re = re.compile(r"\AC[0-9]+\Z")
(0)             def _is_nth_color(c):
(4)                 """Return whether *c* can be interpreted as an item in the color cycle."""
(4)                 return isinstance(c, str) and _nth_color_re.match(c)
(0)             def is_color_like(c):
(4)                 """Return whether *c* can be interpreted as an RGB(A) color."""
(4)                 if _is_nth_color(c):
(8)                     return True
(4)                 try:
(8)                     to_rgba(c)
(4)                 except (TypeError, ValueError):
(8)                     return False
(4)                 else:
(8)                     return True
(0)             def _has_alpha_channel(c):
(4)                 """Return whether *c* is a color with an alpha channel."""
(4)                 return not isinstance(c, str) and len(c) == 4
(0)             def _check_color_like(**kwargs):
(4)                 """
(4)                 For each *key, value* pair in *kwargs*, check that *value* is color-like.
(4)                 """
(4)                 for k, v in kwargs.items():
(8)                     if not is_color_like(v):
(12)                        raise ValueError(
(16)                            f"{v!r} is not a valid value for {k}: supported inputs are "
(16)                            f"(r, g, b) and (r, g, b, a) 0-1 float tuples; "
(16)                            f"'#rrggbb', '#rrggbbaa', '#rgb', '#rgba' strings; "
(16)                            f"named color strings; "
(16)                            f"string reprs of 0-1 floats for grayscale values; "
(16)                            f"'C0', 'C1', ... strings for colors of the color cycle; "
(16)                            f"and pairs combining one of the above with an alpha value")
(0)             def same_color(c1, c2):
(4)                 """
(4)                 Return whether the colors *c1* and *c2* are the same.
(4)                 *c1*, *c2* can be single colors or lists/arrays of colors.
(4)                 """
(4)                 c1 = to_rgba_array(c1)
(4)                 c2 = to_rgba_array(c2)
(4)                 n1 = max(c1.shape[0], 1)  # 'none' results in shape (0, 4), but is 1-elem
(4)                 n2 = max(c2.shape[0], 1)  # 'none' results in shape (0, 4), but is 1-elem
(4)                 if n1 != n2:
(8)                     raise ValueError('Different number of elements passed.')
(4)                 return c1.shape == c2.shape and (c1 == c2).all()
(0)             def to_rgba(c, alpha=None):
(4)                 """
(4)                 Convert *c* to an RGBA color.
(4)                 Parameters
(4)                 ----------
(4)                 c : Matplotlib color or ``np.ma.masked``
(4)                 alpha : float, optional
(8)                     If *alpha* is given, force the alpha value of the returned RGBA tuple
(8)                     to *alpha*.
(8)                     If None, the alpha value from *c* is used. If *c* does not have an
(8)                     alpha channel, then alpha defaults to 1.
(8)                     *alpha* is ignored for the color value ``"none"`` (case-insensitive),
(8)                     which always maps to ``(0, 0, 0, 0)``.
```

```
(4)                 Returns
(4)                 -------
(4)                 tuple
(8)                     Tuple of floats ``(r, g, b, a)``, where each channel (red, green,
blue,
(8)                     alpha) can assume values between 0 and 1.
(4)                 """
(4)                 if isinstance(c, tuple) and len(c) == 2:
(8)                     if alpha is None:
(12)                        c, alpha = c
(8)                     else:
(12)                        c = c[0]
(4)                 if _is_nth_color(c):
(8)                     prop_cycler = mpl.rcParams['axes.prop_cycle']
(8)                     colors = prop_cycler.by_key().get('color', ['k'])
(8)                     c = colors[int(c[1:]) % len(colors)]
(4)                 try:
(8)                     rgba = _colors_full_map.cache[c, alpha]
(4)                 except (KeyError, TypeError):  # Not in cache, or unhashable.
(8)                     rgba = None
(4)                 if rgba is None:  # Suppress exception chaining of cache lookup failure.
(8)                     rgba = _to_rgba_no_colorcycle(c, alpha)
(8)                     try:
(12)                        _colors_full_map.cache[c, alpha] = rgba
(8)                     except TypeError:
(12)                        pass
(4)                 return rgba
(0)             def _to_rgba_no_colorcycle(c, alpha=None):
(4)                 """
(4)                 Convert *c* to an RGBA color, with no support for color-cycle syntax.
(4)                 If *alpha* is given, force the alpha value of the returned RGBA tuple
(4)                 to *alpha*. Otherwise, the alpha value from *c* is used, if it has alpha
(4)                 information, or defaults to 1.
(4)                 *alpha* is ignored for the color value ``"none"`` (case-insensitive),
(4)                 which always maps to ``(0, 0, 0, 0)``.
(4)                 """
(4)                 if alpha is not None and not 0 <= alpha <= 1:
(8)                     raise ValueError("'alpha' must be between 0 and 1, inclusive")
(4)                 orig_c = c
(4)                 if c is np.ma.masked:
(8)                     return (0., 0., 0., 0.)
(4)                 if isinstance(c, str):
(8)                     if c.lower() == "none":
(12)                        return (0., 0., 0., 0.)
(8)                     try:
(12)                        c = _colors_full_map[c]
(8)                     except KeyError:
(12)                        if len(orig_c) != 1:
(16)                            try:
(20)                                c = _colors_full_map[c.lower()]
(16)                            except KeyError:
(20)                                pass
(4)                 if isinstance(c, str):
(8)                     match = re.match(r"\A#[a-fA-F0-9]{6}\Z", c)
(8)                     if match:
(12)                        return (tuple(int(n, 16) / 255
(26)                                      for n in [c[1:3], c[3:5], c[5:7]])
(20)                                + (alpha if alpha is not None else 1.,))
(8)                     match = re.match(r"\A#[a-fA-F0-9]{3}\Z", c)
(8)                     if match:
(12)                        return (tuple(int(n, 16) / 255
(26)                                      for n in [c[1]*2, c[2]*2, c[3]*2])
(20)                                + (alpha if alpha is not None else 1.,))
(8)                     match = re.match(r"\A#[a-fA-F0-9]{8}\Z", c)
(8)                     if match:
(12)                        color = [int(n, 16) / 255
(21)                                 for n in [c[1:3], c[3:5], c[5:7], c[7:9]]]
(12)                        if alpha is not None:
(16)                            color[-1] = alpha
```

```
(12)                        return tuple(color)
(8)                     match = re.match(r"\A#[a-fA-F0-9]{4}\Z", c)
(8)                     if match:
(12)                        color = [int(n, 16) / 255
(21)                                 for n in [c[1]*2, c[2]*2, c[3]*2, c[4]*2]]
(12)                        if alpha is not None:
(16)                            color[-1] = alpha
(12)                        return tuple(color)
(8)                     try:
(12)                        c = float(c)
(8)                     except ValueError:
(12)                        pass
(8)                     else:
(12)                        if not (0 <= c <= 1):
(16)                            raise ValueError(
(20)                                f"Invalid string grayscale value {orig_c!r}. "
(20)                                f"Value must be within 0-1 range")
(12)                        return c, c, c, alpha if alpha is not None else 1.
(8)                     raise ValueError(f"Invalid RGBA argument: {orig_c!r}")
(4)                 if isinstance(c, np.ndarray):
(8)                     if c.ndim == 2 and c.shape[0] == 1:
(12)                        c = c.reshape(-1)
(4)                 if not np.iterable(c):
(8)                     raise ValueError(f"Invalid RGBA argument: {orig_c!r}")
(4)                 if len(c) not in [3, 4]:
(8)                     raise ValueError("RGBA sequence should have length 3 or 4")
(4)                 if not all(isinstance(x, Real) for x in c):
(8)                     raise ValueError(f"Invalid RGBA argument: {orig_c!r}")
(4)                 c = tuple(map(float, c))
(4)                 if len(c) == 3 and alpha is None:
(8)                     alpha = 1
(4)                 if alpha is not None:
(8)                     c = c[:3] + (alpha,)
(4)                 if any(elem < 0 or elem > 1 for elem in c):
(8)                     raise ValueError("RGBA values should be within 0-1 range")
(4)                 return c
(0)             def to_rgba_array(c, alpha=None):
(4)                 """
(4)                 Convert *c* to a (n, 4) array of RGBA colors.
(4)                 Parameters
(4)                 ----------
(4)                 c : Matplotlib color or array of colors
(8)                     If *c* is a masked array, an `~numpy.ndarray` is returned with a
(8)                     (0, 0, 0, 0) row for each masked value or row in *c*.
(4)                 alpha : float or sequence of floats, optional
(8)                     If *alpha* is given, force the alpha value of the returned RGBA tuple
(8)                     to *alpha*.
(8)                     If None, the alpha value from *c* is used. If *c* does not have an
(8)                     alpha channel, then alpha defaults to 1.
(8)                     *alpha* is ignored for the color value ``"none"`` (case-insensitive),
(8)                     which always maps to ``(0, 0, 0, 0)``.
(8)                     If *alpha* is a sequence and *c* is a single color, *c* will be
(8)                     repeated to match the length of *alpha*.
(4)                 Returns
(4)                 -------
(4)                 array
(8)                     (n, 4) array of RGBA colors,  where each channel (red, green, blue,
(8)                     alpha) can assume values between 0 and 1.
(4)                 """
(4)                 if isinstance(c, tuple) and len(c) == 2 and isinstance(c[1], Real):
(8)                     if alpha is None:
(12)                        c, alpha = c
(8)                     else:
(12)                        c = c[0]
(4)                 if np.iterable(alpha):
(8)                     alpha = np.asarray(alpha).ravel()
(4)                 if (isinstance(c, np.ndarray) and c.dtype.kind in "if"
(12)                        and c.ndim == 2 and c.shape[1] in [3, 4]):
(8)                     mask = c.mask.any(axis=1) if np.ma.is_masked(c) else None
```

```
(8)                     c = np.ma.getdata(c)
(8)                     if np.iterable(alpha):
(12)                        if c.shape[0] == 1 and alpha.shape[0] > 1:
(16)                            c = np.tile(c, (alpha.shape[0], 1))
(12)                        elif c.shape[0] != alpha.shape[0]:
(16)                            raise ValueError("The number of colors must match the number"
(33)                                             " of alpha values if there are more than one"
(33)                                             " of each.")
(8)                     if c.shape[1] == 3:
(12)                        result = np.column_stack([c, np.zeros(len(c))])
(12)                        result[:, -1] = alpha if alpha is not None else 1.
(8)                     elif c.shape[1] == 4:
(12)                        result = c.copy()
(12)                        if alpha is not None:
(16)                            result[:, -1] = alpha
(8)                     if mask is not None:
(12)                        result[mask] = 0
(8)                     if np.any((result < 0) | (result > 1)):
(12)                        raise ValueError("RGBA values should be within 0-1 range")
(8)                     return result
(4)                 if cbook._str_lower_equal(c, "none"):
(8)                     return np.zeros((0, 4), float)
(4)                 try:
(8)                     if np.iterable(alpha):
(12)                        return np.array([to_rgba(c, a) for a in alpha], float)
(8)                     else:
(12)                        return np.array([to_rgba(c, alpha)], float)
(4)                 except TypeError:
(8)                     pass
(4)                 except ValueError as e:
(8)                     if e.args == ("'alpha' must be between 0 and 1, inclusive", ):
(12)                        raise e
(4)                 if isinstance(c, str):
(8)                     raise ValueError(f"{c!r} is not a valid color value.")
(4)                 if len(c) == 0:
(8)                     return np.zeros((0, 4), float)
(4)                 if isinstance(c, Sequence):
(8)                     lens = {len(cc) if isinstance(cc, (list, tuple)) else -1 for cc in c}
(8)                     if lens == {3}:
(12)                        rgba = np.column_stack([c, np.ones(len(c))])
(8)                     elif lens == {4}:
(12)                        rgba = np.array(c)
(8)                     else:
(12)                        rgba = np.array([to_rgba(cc) for cc in c])
(4)                 else:
(8)                     rgba = np.array([to_rgba(cc) for cc in c])
(4)                 if alpha is not None:
(8)                     rgba[:, 3] = alpha
(8)                     if isinstance(c, Sequence):
(12)                        none_mask = [cbook._str_equal(cc, "none") for cc in c]
(12)                        rgba[:, 3][none_mask] = 0
(4)                 return rgba
(0)             def to_rgb(c):
(4)                 """Convert *c* to an RGB color, silently dropping the alpha channel."""
(4)                 return to_rgba(c)[:3]
(0)             def to_hex(c, keep_alpha=False):
(4)                 """
(4)                 Convert *c* to a hex color.
(4)                 Parameters
(4)                 ----------
(4)                 c : :ref:`color <colors_def>` or `numpy.ma.masked`
(4)                 keep_alpha : bool, default: False
(6)                   If False, use the ``#rrggbb`` format, otherwise use ``#rrggbbaa``.
(4)                 Returns
(4)                 -------
(4)                 str
(6)                   ``#rrggbb`` or ``#rrggbbaa`` hex color string
(4)                 """
(4)                 c = to_rgba(c)
```

```
(4)                 if not keep_alpha:
(8)                     c = c[:3]
(4)                 return "#" + "".join(format(round(val * 255), "02x") for val in c)
(0)             cnames = CSS4_COLORS
(0)             hexColorPattern = re.compile(r"\A#[a-fA-F0-9]{6}\Z")
(0)             rgb2hex = to_hex
(0)             hex2color = to_rgb
(0)             class ColorConverter:
(4)                 """
(4)                 A class only kept for backwards compatibility.
(4)                 Its functionality is entirely provided by module-level functions.
(4)                 """
(4)                 colors = _colors_full_map
(4)                 cache = _colors_full_map.cache
(4)                 to_rgb = staticmethod(to_rgb)
(4)                 to_rgba = staticmethod(to_rgba)
(4)                 to_rgba_array = staticmethod(to_rgba_array)
(0)             colorConverter = ColorConverter()
(0)             def _create_lookup_table(N, data, gamma=1.0):
(4)                 r"""
(4)                 Create an *N* -element 1D lookup table.
(4)                 This assumes a mapping :math:`f : [0, 1] \rightarrow [0, 1]`. The returned
(4)                 data is an array of N values :math:`y = f(x)` where x is sampled from
(4)                 [0, 1].
(4)                 By default (*gamma* = 1) x is equidistantly sampled from [0, 1]. The
(4)                 *gamma* correction factor :math:`\gamma` distorts this equidistant
(4)                 sampling by :math:`x \rightarrow x^\gamma`.
(4)                 Parameters
(4)                 ----------
(4)                 N : int
(8)                     The number of elements of the created lookup table; at least 1.
(4)                 data : (M, 3) array-like or callable
(8)                     Defines the mapping :math:`f`.
(8)                     If a (M, 3) array-like, the rows define values (x, y0, y1).  The x
(8)                     values must start with x=0, end with x=1, and all x values be in
(8)                     increasing order.
(8)                     A value between :math:`x_i` and :math:`x_{i+1}` is mapped to the range
(8)                     :math:`y^1_{i-1} \ldots y^0_i` by linear interpolation.
(8)                     For the simple case of a y-continuous mapping, y0 and y1 are
identical.
(8)                     The two values of y are to allow for discontinuous mapping functions.
(8)                     E.g. a sawtooth with a period of 0.2 and an amplitude of 1 would be::
(12)                        [(0, 1, 0), (0.2, 1, 0), (0.4, 1, 0), ..., [(1, 1, 0)]
(8)                     In the special case of ``N == 1``, by convention the returned value
(8)                     is y0 for x == 1.
(8)                     If *data* is a callable, it must accept and return numpy arrays::
(11)                       data(x : ndarray) -> ndarray
(8)                     and map values between 0 - 1 to 0 - 1.
(4)                 gamma : float
(8)                     Gamma correction factor for input distribution x of the mapping.
(8)                     See also https://en.wikipedia.org/wiki/Gamma_correction.
(4)                 Returns
(4)                 -------
(4)                 array
(8)                     The lookup table where ``lut[x * (N-1)]`` gives the closest value
(8)                     for values of x between 0 and 1.
(4)                 Notes
(4)                 -----
(4)                 This function is internally used for `.LinearSegmentedColormap`.
(4)                 """
(4)                 if callable(data):
(8)                     xind = np.linspace(0, 1, N) ** gamma
(8)                     lut = np.clip(np.array(data(xind), dtype=float), 0, 1)
(8)                     return lut
(4)                 try:
(8)                     adata = np.array(data)
(4)                 except Exception as err:
(8)                     raise TypeError("data must be convertible to an array") from err
(4)                 _api.check_shape((None, 3), data=adata)
```

```
(4)                 x = adata[:, 0]
(4)                 y0 = adata[:, 1]
(4)                 y1 = adata[:, 2]
(4)                 if x[0] != 0. or x[-1] != 1.0:
(8)                     raise ValueError(
(12)                        "data mapping points must start with x=0 and end with x=1")
(4)                 if (np.diff(x) < 0).any():
(8)                     raise ValueError("data mapping points must have x in increasing order")
(4)                 if N == 1:
(8)                     lut = np.array(y0[-1])
(4)                 else:
(8)                     x = x * (N - 1)
(8)                     xind = (N - 1) * np.linspace(0, 1, N) ** gamma
(8)                     ind = np.searchsorted(x, xind)[1:-1]
(8)                     distance = (xind[1:-1] - x[ind - 1]) / (x[ind] - x[ind - 1])
(8)                     lut = np.concatenate([
(12)                        [y1[0]],
(12)                        distance * (y0[ind] - y1[ind - 1]) + y1[ind - 1],
(12)                        [y0[-1]],
(8)                     ])
(4)                 return np.clip(lut, 0.0, 1.0)
(0)             class Colormap:
(4)                 """
(4)                 Baseclass for all scalar to RGBA mappings.
(4)                 Typically, Colormap instances are used to convert data values (floats)
(4)                 from the interval ``[0, 1]`` to the RGBA color that the respective
(4)                 Colormap represents. For scaling of data into the ``[0, 1]`` interval see
(4)                 `matplotlib.colors.Normalize`. Subclasses of `matplotlib.cm.ScalarMappable`
(4)                 make heavy use of this ``data -> normalize -> map-to-color`` processing
(4)                 chain.
(4)                 """
(4)                 def __init__(self, name, N=256):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     name : str
(12)                        The name of the colormap.
(8)                     N : int
(12)                        The number of RGB quantization levels.
(8)                     """
(8)                     self.name = name
(8)                     self.N = int(N)  # ensure that N is always int
(8)                     self._rgba_bad = (0.0, 0.0, 0.0, 0.0)  # If bad, don't paint anything.
(8)                     self._rgba_under = None
(8)                     self._rgba_over = None
(8)                     self._i_under = self.N
(8)                     self._i_over = self.N + 1
(8)                     self._i_bad = self.N + 2
(8)                     self._isinit = False
(8)                     self.colorbar_extend = False
(4)                 def __call__(self, X, alpha=None, bytes=False):
(8)                     r"""
(8)                     Parameters
(8)                     ----------
(8)                     X : float or int, `~numpy.ndarray` or scalar
(12)                        The data value(s) to convert to RGBA.
(12)                        For floats, *X* should be in the interval ``[0.0, 1.0]`` to
(12)                        return the RGBA values ``X*100`` percent along the Colormap line.
(12)                        For integers, *X* should be in the interval ``[0, Colormap.N)`` to
(12)                        return RGBA values *indexed* from the Colormap with index ``X``.
(8)                     alpha : float or array-like or None
(12)                        Alpha must be a scalar between 0 and 1, a sequence of such
(12)                        floats with shape matching X, or None.
(8)                     bytes : bool
(12)                        If False (default), the returned RGBA values will be floats in the
(12)                        interval ``[0, 1]`` otherwise they will be `numpy.uint8`\s in the
(12)                        interval ``[0, 255]``.
```

```
(8)                     Returns
(8)                     -------
(8)                     Tuple of RGBA values if X is scalar, otherwise an array of
(8)                     RGBA values with a shape of ``X.shape + (4, )``.
(8)                     """
(8)                     if not self._isinit:
(12)                        self._init()
(8)                     xa = np.array(X, copy=True)
(8)                     if not xa.dtype.isnative:
(12)                        xa = xa.byteswap().view(xa.dtype.newbyteorder())
(8)                     if xa.dtype.kind == "f":
(12)                        xa *= self.N
(12)                        xa[xa == self.N] = self.N - 1
(8)                     mask_under = xa < 0
(8)                     mask_over = xa >= self.N
(8)                     mask_bad = X.mask if np.ma.is_masked(X) else np.isnan(xa)
(8)                     with np.errstate(invalid="ignore"):
(12)                        xa = xa.astype(int)
(8)                     xa[mask_under] = self._i_under
(8)                     xa[mask_over] = self._i_over
(8)                     xa[mask_bad] = self._i_bad
(8)                     lut = self._lut
(8)                     if bytes:
(12)                        lut = (lut * 255).astype(np.uint8)
(8)                     rgba = lut.take(xa, axis=0, mode='clip')
(8)                     if alpha is not None:
(12)                        alpha = np.clip(alpha, 0, 1)
(12)                        if bytes:
(16)                            alpha *= 255  # Will be cast to uint8 upon assignment.
(12)                        if alpha.shape not in [(), xa.shape]:
(16)                            raise ValueError(
(20)                                f"alpha is array-like but its shape {alpha.shape} does "
(20)                                f"not match that of X {xa.shape}")
(12)                        rgba[..., -1] = alpha
(12)                        if (lut[-1] == 0).all():
(16)                            rgba[mask_bad] = (0, 0, 0, 0)
(8)                     if not np.iterable(X):
(12)                        rgba = tuple(rgba)
(8)                     return rgba
(4)                 def __copy__(self):
(8)                     cls = self.__class__
(8)                     cmapobject = cls.__new__(cls)
(8)                     cmapobject.__dict__.update(self.__dict__)
(8)                     if self._isinit:
(12)                        cmapobject._lut = np.copy(self._lut)
(8)                     return cmapobject
(4)                 def __eq__(self, other):
(8)                     if (not isinstance(other, Colormap) or
(16)                            self.colorbar_extend != other.colorbar_extend):
(12)                        return False
(8)                     if not self._isinit:
(12)                        self._init()
(8)                     if not other._isinit:
(12)                        other._init()
(8)                     return np.array_equal(self._lut, other._lut)
(4)                 def get_bad(self):
(8)                     """Get the color for masked values."""
(8)                     if not self._isinit:
(12)                        self._init()
(8)                     return np.array(self._lut[self._i_bad])
(4)                 def set_bad(self, color='k', alpha=None):
(8)                     """Set the color for masked values."""
(8)                     self._rgba_bad = to_rgba(color, alpha)
(8)                     if self._isinit:
(12)                        self._set_extremes()
(4)                 def get_under(self):
(8)                     """Get the color for low out-of-range values."""
(8)                     if not self._isinit:
(12)                        self._init()
```

```
(8)                 return np.array(self._lut[self._i_under])
(4)             def set_under(self, color='k', alpha=None):
(8)                 """Set the color for low out-of-range values."""
(8)                 self._rgba_under = to_rgba(color, alpha)
(8)                 if self._isinit:
(12)                    self._set_extremes()
(4)             def get_over(self):
(8)                 """Get the color for high out-of-range values."""
(8)                 if not self._isinit:
(12)                    self._init()
(8)                 return np.array(self._lut[self._i_over])
(4)             def set_over(self, color='k', alpha=None):
(8)                 """Set the color for high out-of-range values."""
(8)                 self._rgba_over = to_rgba(color, alpha)
(8)                 if self._isinit:
(12)                    self._set_extremes()
(4)             def set_extremes(self, *, bad=None, under=None, over=None):
(8)                 """
(8)                 Set the colors for masked (*bad*) values and, when ``norm.clip =
(8)                 False``, low (*under*) and high (*over*) out-of-range values.
(8)                 """
(8)                 if bad is not None:
(12)                    self.set_bad(bad)
(8)                 if under is not None:
(12)                    self.set_under(under)
(8)                 if over is not None:
(12)                    self.set_over(over)
(4)             def with_extremes(self, *, bad=None, under=None, over=None):
(8)                 """
(8)                 Return a copy of the colormap, for which the colors for masked (*bad*)
(8)                 values and, when ``norm.clip = False``, low (*under*) and high
(*over*)
(8)                 out-of-range values, have been set accordingly.
(8)                 """
(8)                 new_cm = self.copy()
(8)                 new_cm.set_extremes(bad=bad, under=under, over=over)
(8)                 return new_cm
(4)             def _set_extremes(self):
(8)                 if self._rgba_under:
(12)                    self._lut[self._i_under] = self._rgba_under
(8)                 else:
(12)                    self._lut[self._i_under] = self._lut[0]
(8)                 if self._rgba_over:
(12)                    self._lut[self._i_over] = self._rgba_over
(8)                 else:
(12)                    self._lut[self._i_over] = self._lut[self.N - 1]
(8)                 self._lut[self._i_bad] = self._rgba_bad
(4)             def _init(self):
(8)                 """Generate the lookup table, ``self._lut``."""
(8)                 raise NotImplementedError("Abstract class only")
(4)             def is_gray(self):
(8)                 """Return whether the colormap is grayscale."""
(8)                 if not self._isinit:
(12)                    self._init()
(8)                 return (np.all(self._lut[:, 0] == self._lut[:, 1]) and
(16)                        np.all(self._lut[:, 0] == self._lut[:, 2]))
(4)             def resampled(self, lutsize):
(8)                 """Return a new colormap with *lutsize* entries."""
(8)                 if hasattr(self, '_resample'):
(12)                    _api.warn_external(
(16)                        "The ability to resample a color map is now public API "
(16)                        f"However the class {type(self)} still only implements "
(16)                        "the previous private _resample method.  Please update "
(16)                        "your class."
(12)                    )
(12)                    return self._resample(lutsize)
(8)                 raise NotImplementedError()
(4)             def reversed(self, name=None):
(8)                 """
```

```
694: (8)                    Return a reversed instance of the Colormap.
695: (8)                    .. note:: This function is not implemented for the base class.
696: (8)                    Parameters
697: (8)                    ----------
698: (8)                    name : str, optional
699: (12)                       The name for the reversed colormap. If None, the
700: (12)                       name is set to ``self.name + "_r"``.
701: (8)                    See Also
702: (8)                    --------
703: (8)                    LinearSegmentedColormap.reversed
704: (8)                    ListedColormap.reversed
705: (8)                    """
706: (8)                    raise NotImplementedError()
707: (4)                def _repr_png_(self):
708: (8)                    """Generate a PNG representation of the Colormap."""
709: (8)                    X = np.tile(np.linspace(0, 1, _REPR_PNG_SIZE[0]),
710: (20)                               (_REPR_PNG_SIZE[1], 1))
711: (8)                    pixels = self(X, bytes=True)
712: (8)                    png_bytes = io.BytesIO()
713: (8)                    title = self.name + ' colormap'
714: (8)                    author = f'Matplotlib v{mpl.__version__}, https://matplotlib.org'
715: (8)                    pnginfo = PngInfo()
716: (8)                    pnginfo.add_text('Title', title)
717: (8)                    pnginfo.add_text('Description', title)
718: (8)                    pnginfo.add_text('Author', author)
719: (8)                    pnginfo.add_text('Software', author)
720: (8)                    Image.fromarray(pixels).save(png_bytes, format='png', pnginfo=pnginfo)
721: (8)                    return png_bytes.getvalue()
722: (4)                def _repr_html_(self):
723: (8)                    """Generate an HTML representation of the Colormap."""
724: (8)                    png_bytes = self._repr_png_()
725: (8)                    png_base64 = base64.b64encode(png_bytes).decode('ascii')
726: (8)                    def color_block(color):
727: (12)                       hex_color = to_hex(color, keep_alpha=True)
728: (12)                       return (f'<div title="{hex_color}" '
729: (20)                               'style="display: inline-block; '
730: (20)                               'width: 1em; height: 1em; '
731: (20)                               'margin: 0; '
732: (20)                               'vertical-align: middle; '
733: (20)                               'border: 1px solid #555; '
734: (20)                               f'background-color: {hex_color};"></div>')
735: (8)                    return ('<div style="vertical-align: middle;">'
736: (16)                           f'<strong>{self.name}</strong> '
737: (16)                           '</div>'
738: (16)                           '<div class="cmap"><img '
739: (16)                           f'alt="{self.name} colormap" '
740: (16)                           f'title="{self.name}" '
741: (16)                           'style="border: 1px solid #555;" '
742: (16)                           f'src="{png_base64}"></div>'
743: (16)                           '<div style="vertical-align: middle; '
744: (16)                           f'max-width: {_REPR_PNG_SIZE[0]+2}px; '
745: (16)                           'display: flex; justify-content: space-between;">'
746: (16)                           '<div style="float: left;">'
747: (16)                           f'{color_block(self.get_under())} under'
748: (16)                           '</div>'
749: (16)                           '<div style="margin: 0 auto; display: inline-block;">'
750: (16)                           f'bad {color_block(self.get_bad())}'
751: (16)                           '</div>'
752: (16)                           '<div style="float: right;">'
753: (16)                           f'over {color_block(self.get_over())}'
754: (16)                           '</div>'
755: (16)                           '</div>')
756: (4)                def copy(self):
757: (8)                    """Return a copy of the colormap."""
758: (8)                    return self.__copy__()
759: (0)            class LinearSegmentedColormap(Colormap):
760: (4)                """
761: (4)                Colormap objects based on lookup tables using linear segments.
762: (4)                The lookup table is generated using linear interpolation for each
```

```
(4)                 primary color, with the 0-1 domain divided into any number of
(4)                 segments.
(4)                 """
(4)                 def __init__(self, name, segmentdata, N=256, gamma=1.0):
(8)                     """
(8)                     Create colormap from linear mapping segments
(8)                     segmentdata argument is a dictionary with a red, green and blue
(8)                     entries. Each entry should be a list of *x*, *y0*, *y1* tuples,
(8)                     forming rows in a table. Entries for alpha are optional.
(8)                     Example: suppose you want red to increase from 0 to 1 over
(8)                     the bottom half, green to do the same over the middle half,
(8)                     and blue over the top half.  Then you would use::
(12)                        cdict = {'red':   [(0.0,  0.0, 0.0),
(31)                                           (0.5,  1.0, 1.0),
(31)                                           (1.0,  1.0, 1.0)],
(21)                                 'green': [(0.0,  0.0, 0.0),
(31)                                           (0.25, 0.0, 0.0),
(31)                                           (0.75, 1.0, 1.0),
(31)                                           (1.0,  1.0, 1.0)],
(21)                                 'blue':  [(0.0,  0.0, 0.0),
(31)                                           (0.5,  0.0, 0.0),
(31)                                           (1.0,  1.0, 1.0)]}
(8)                     Each row in the table for a given color is a sequence of
(8)                     *x*, *y0*, *y1* tuples.  In each sequence, *x* must increase
(8)                     monotonically from 0 to 1.  For any input value *z* falling
(8)                     between *x[i]* and *x[i+1]*, the output value of a given color
(8)                     will be linearly interpolated between *y1[i]* and *y0[i+1]*::
(12)                        row i:   x  y0  y1
(27)                                       /
(26)                                      /
(12)                        row i+1: x  y0  y1
(8)                     Hence y0 in the first row and y1 in the last row are never used.
(8)                     See Also
(8)                     --------
(8)                     LinearSegmentedColormap.from_list
(12)                        Static method; factory function for generating a smoothly-varying
(12)                        LinearSegmentedColormap.
(8)                     """
(8)                     self.monochrome = False
(8)                     super().__init__(name, N)
(8)                     self._segmentdata = segmentdata
(8)                     self._gamma = gamma
(4)                 def _init(self):
(8)                     self._lut = np.ones((self.N + 3, 4), float)
(8)                     self._lut[:-3, 0] = _create_lookup_table(
(12)                        self.N, self._segmentdata['red'], self._gamma)
(8)                     self._lut[:-3, 1] = _create_lookup_table(
(12)                        self.N, self._segmentdata['green'], self._gamma)
(8)                     self._lut[:-3, 2] = _create_lookup_table(
(12)                        self.N, self._segmentdata['blue'], self._gamma)
(8)                     if 'alpha' in self._segmentdata:
(12)                        self._lut[:-3, 3] = _create_lookup_table(
(16)                            self.N, self._segmentdata['alpha'], 1)
(8)                     self._isinit = True
(8)                     self._set_extremes()
(4)                 def set_gamma(self, gamma):
(8)                     """Set a new gamma value and regenerate colormap."""
(8)                     self._gamma = gamma
(8)                     self._init()
(4)                 @staticmethod
(4)                 def from_list(name, colors, N=256, gamma=1.0):
(8)                     """
(8)                     Create a `LinearSegmentedColormap` from a list of colors.
(8)                     Parameters
(8)                     ----------
(8)                     name : str
(12)                        The name of the colormap.
(8)                     colors : list of :mpltype:`color` or list of (value, color)
(12)                        If only colors are given, they are equidistantly mapped from the
```

```
(12)                        range :math:`[0, 1]`; i.e. 0 maps to ``colors[0]`` and 1 maps to
(12)                        ``colors[-1]``.
(12)                        If (value, color) pairs are given, the mapping is from *value*
(12)                        to *color*. This can be used to divide the range unevenly.
(8)                     N : int
(12)                        The number of RGB quantization levels.
(8)                     gamma : float
(8)                     """
(8)                     if not np.iterable(colors):
(12)                        raise ValueError('colors must be iterable')
(8)                     if (isinstance(colors[0], Sized) and len(colors[0]) == 2
(16)                            and not isinstance(colors[0], str)):
(12)                        vals, colors = zip(*colors)
(8)                     else:
(12)                        vals = np.linspace(0, 1, len(colors))
(8)                     r, g, b, a = to_rgba_array(colors).T
(8)                     cdict = {
(12)                        "red": np.column_stack([vals, r, r]),
(12)                        "green": np.column_stack([vals, g, g]),
(12)                        "blue": np.column_stack([vals, b, b]),
(12)                        "alpha": np.column_stack([vals, a, a]),
(8)                     }
(8)                     return LinearSegmentedColormap(name, cdict, N, gamma)
(4)                 def resampled(self, lutsize):
(8)                     """Return a new colormap with *lutsize* entries."""
(8)                     new_cmap = LinearSegmentedColormap(self.name, self._segmentdata,
(43)                                                       lutsize)
(8)                     new_cmap._rgba_over = self._rgba_over
(8)                     new_cmap._rgba_under = self._rgba_under
(8)                     new_cmap._rgba_bad = self._rgba_bad
(8)                     return new_cmap
(4)                 @staticmethod
(4)                 def _reverser(func, x):
(8)                     return func(1 - x)
(4)                 def reversed(self, name=None):
(8)                     """
(8)                     Return a reversed instance of the Colormap.
(8)                     Parameters
(8)                     ----------
(8)                     name : str, optional
(12)                        The name for the reversed colormap. If None, the
(12)                        name is set to ``self.name + "_r"``.
(8)                     Returns
(8)                     -------
(8)                     LinearSegmentedColormap
(12)                        The reversed colormap.
(8)                     """
(8)                     if name is None:
(12)                        name = self.name + "_r"
(8)                     data_r = {key: (functools.partial(self._reverser, data)
(24)                                    if callable(data) else
(24)                                    [(1.0 - x, y1, y0) for x, y0, y1 in reversed(data)])
(18)                              for key, data in self._segmentdata.items()}
(8)                     new_cmap = LinearSegmentedColormap(name, data_r, self.N, self._gamma)
(8)                     new_cmap._rgba_over = self._rgba_under
(8)                     new_cmap._rgba_under = self._rgba_over
(8)                     new_cmap._rgba_bad = self._rgba_bad
(8)                     return new_cmap
(0)             class ListedColormap(Colormap):
(4)                 """
(4)                 Colormap object generated from a list of colors.
(4)                 This may be most useful when indexing directly into a colormap,
(4)                 but it can also be used to generate special colormaps for ordinary
(4)                 mapping.
(4)                 Parameters
(4)                 ----------
(4)                 colors : list, array
(8)                     Sequence of Matplotlib color specifications (color names or RGB(A)
(8)                     values).
```

```
(4)                 name : str, optional
(8)                     String to identify the colormap.
(4)                 N : int, optional
(8)                     Number of entries in the map. The default is *None*, in which case
(8)                     there is one colormap entry for each element in the list of colors.
(8)                     If ::
(12)                        N < len(colors)
(8)                     the list will be truncated at *N*. If ::
(12)                        N > len(colors)
(8)                     the list will be extended by repetition.
(4)                 """
(4)                 def __init__(self, colors, name='from_list', N=None):
(8)                     self.monochrome = False  # Are all colors identical? (for contour.py)
(8)                     if N is None:
(12)                        self.colors = colors
(12)                        N = len(colors)
(8)                     else:
(12)                        if isinstance(colors, str):
(16)                            self.colors = [colors] * N
(16)                            self.monochrome = True
(12)                        elif np.iterable(colors):
(16)                            if len(colors) == 1:
(20)                                self.monochrome = True
(16)                            self.colors = list(
(20)                                itertools.islice(itertools.cycle(colors), N))
(12)                        else:
(16)                            try:
(20)                                gray = float(colors)
(16)                            except TypeError:
(20)                                pass
(16)                            else:
(20)                                self.colors = [gray] * N
(16)                            self.monochrome = True
(8)                     super().__init__(name, N)
(4)                 def _init(self):
(8)                     self._lut = np.zeros((self.N + 3, 4), float)
(8)                     self._lut[:-3] = to_rgba_array(self.colors)
(8)                     self._isinit = True
(8)                     self._set_extremes()
(4)                 def resampled(self, lutsize):
(8)                     """Return a new colormap with *lutsize* entries."""
(8)                     colors = self(np.linspace(0, 1, lutsize))
(8)                     new_cmap = ListedColormap(colors, name=self.name)
(8)                     new_cmap._rgba_over = self._rgba_over
(8)                     new_cmap._rgba_under = self._rgba_under
(8)                     new_cmap._rgba_bad = self._rgba_bad
(8)                     return new_cmap
(4)                 def reversed(self, name=None):
(8)                     """
(8)                     Return a reversed instance of the Colormap.
(8)                     Parameters
(8)                     ----------
(8)                     name : str, optional
(12)                        The name for the reversed colormap. If None, the
(12)                        name is set to ``self.name + "_r"``.
(8)                     Returns
(8)                     -------
(8)                     ListedColormap
(12)                        A reversed instance of the colormap.
(8)                     """
(8)                     if name is None:
(12)                        name = self.name + "_r"
(8)                     colors_r = list(reversed(self.colors))
(8)                     new_cmap = ListedColormap(colors_r, name=name, N=self.N)
(8)                     new_cmap._rgba_over = self._rgba_under
(8)                     new_cmap._rgba_under = self._rgba_over
(8)                     new_cmap._rgba_bad = self._rgba_bad
(8)                     return new_cmap
(0)             class Normalize:
```

```
    """
    A class which, when called, maps values within the interval
    ``[vmin, vmax]`` linearly to the interval ``[0.0, 1.0]``. The mapping of
    values outside ``[vmin, vmax]`` depends on *clip*.
    Examples
    --------
    ::
        x = [-2, -1, 0, 1, 2]
        norm = mpl.colors.Normalize(vmin=-1, vmax=1, clip=False)
        norm(x)  # [-0.5, 0., 0.5, 1., 1.5]
        norm = mpl.colors.Normalize(vmin=-1, vmax=1, clip=True)
        norm(x)  # [0., 0., 0.5, 1., 1.]
    See Also
    --------
    :ref:`colormapnorms`
    """
    def __init__(self, vmin=None, vmax=None, clip=False):
        """
        Parameters
        ----------
        vmin, vmax : float or None
            Values within the range ``[vmin, vmax]`` from the input data will be

            linearly mapped to ``[0, 1]``. If either *vmin* or *vmax* is not
            provided, they default to the minimum and maximum values of the input,

            respectively.
        clip : bool, default: False
            Determines the behavior for mapping values outside the range
            ``[vmin, vmax]``.
            If clipping is off, values outside the range ``[vmin, vmax]`` are
            also transformed, resulting in values outside ``[0, 1]``.  This
            behavior is usually desirable, as colormaps can mark these *under*
            and *over* values with specific colors.
            If clipping is on, values below *vmin* are mapped to 0 and values
            above *vmax* are mapped to 1. Such values become indistinguishable
            from regular boundary values, which may cause misinterpretation of
            the data.
        Notes
        -----
        If ``vmin == vmax``, input data will be mapped to 0.
        """
        self._vmin = _sanitize_extrema(vmin)
        self._vmax = _sanitize_extrema(vmax)
        self._clip = clip
        self._scale = None
        self.callbacks = cbook.CallbackRegistry(signals=["changed"])
    @property
    def vmin(self):
        return self._vmin
    @vmin.setter
    def vmin(self, value):
        value = _sanitize_extrema(value)
        if value != self._vmin:
            self._vmin = value
            self._changed()
    @property
    def vmax(self):
        return self._vmax
    @vmax.setter
    def vmax(self, value):
        value = _sanitize_extrema(value)
        if value != self._vmax:
            self._vmax = value
            self._changed()
    @property
    def clip(self):
        return self._clip
    @clip.setter
```

```
(4)     def clip(self, value):
(8)         if value != self._clip:
(12)            self._clip = value
(12)            self._changed()
(4)     def _changed(self):
(8)         """
(8)         Call this whenever the norm is changed to notify all the
(8)         callback listeners to the 'changed' signal.
(8)         """
(8)         self.callbacks.process('changed')
(4)     @staticmethod
(4)     def process_value(value):
(8)         """
(8)         Homogenize the input *value* for easy and efficient normalization.
(8)         *value* can be a scalar or sequence.
(8)         Parameters
(8)         ----------
(8)         value
(12)            Data to normalize.
(8)         Returns
(8)         -------
(8)         result : masked array
(12)            Masked array with the same shape as *value*.
(8)         is_scalar : bool
(12)            Whether *value* is a scalar.
(8)         Notes
(8)         -----
(8)         Float dtypes are preserved; integer types with two bytes or smaller
are
(8)         converted to np.float32, and larger types are converted to np.float64.
(8)         Preserving float32 when possible, and using in-place operations,
(8)         greatly improves speed for large arrays.
(8)         """
(8)         is_scalar = not np.iterable(value)
(8)         if is_scalar:
(12)            value = [value]
(8)         dtype = np.min_scalar_type(value)
(8)         if np.issubdtype(dtype, np.integer) or dtype.type is np.bool_:
(12)            dtype = np.promote_types(dtype, np.float32)
(8)         mask = np.ma.getmask(value)
(8)         data = np.asarray(value)
(8)         result = np.ma.array(data, mask=mask, dtype=dtype, copy=True)
(8)         return result, is_scalar
(4)     def __call__(self, value, clip=None):
(8)         """
(8)         Normalize the data and return the normalized data.
(8)         Parameters
(8)         ----------
(8)         value
(12)            Data to normalize.
(8)         clip : bool, optional
(12)            See the description of the parameter *clip* in `.Normalize`.
(12)            If ``None``, defaults to ``self.clip`` (which defaults to
(12)            ``False``).
(8)         Notes
(8)         -----
(8)         If not already initialized, ``self.vmin`` and ``self.vmax`` are
(8)         initialized using ``self.autoscale_None(value)``.
(8)         """
(8)         if clip is None:
(12)            clip = self.clip
(8)         result, is_scalar = self.process_value(value)
(8)         if self.vmin is None or self.vmax is None:
(12)            self.autoscale_None(result)
(8)         (vmin,), _ = self.process_value(self.vmin)
(8)         (vmax,), _ = self.process_value(self.vmax)
(8)         if vmin == vmax:
(12)            result.fill(0)  # Or should it be all masked?  Or 0.5?
(8)         elif vmin > vmax:
```

```
1105: (12)                      raise ValueError("minvalue must be less than or equal to
maxvalue")
1106: (8)                   else:
1107: (12)                      if clip:
1108: (16)                          mask = np.ma.getmask(result)
1109: (16)                          result = np.ma.array(np.clip(result.filled(vmax), vmin, vmax),
1110: (37)                                               mask=mask)
1111: (12)                      resdat = result.data
1112: (12)                      resdat -= vmin
1113: (12)                      resdat /= (vmax - vmin)
1114: (12)                      result = np.ma.array(resdat, mask=result.mask, copy=False)
1115: (8)                   if is_scalar:
1116: (12)                      result = result[0]
1117: (8)                   return result
1118: (4)               def inverse(self, value):
1119: (8)                   """
1120: (8)                   Maps the normalized value (i.e., index in the colormap) back to image
1121: (8)                   data value.
1122: (8)                   Parameters
1123: (8)                   ----------
1124: (8)                   value
1125: (12)                      Normalized value.
1126: (8)                   """
1127: (8)                   if not self.scaled():
1128: (12)                      raise ValueError("Not invertible until both vmin and vmax are
set")
1129: (8)                   (vmin,), _ = self.process_value(self.vmin)
1130: (8)                   (vmax,), _ = self.process_value(self.vmax)
1131: (8)                   if np.iterable(value):
1132: (12)                      val = np.ma.asarray(value)
1133: (12)                      return vmin + val * (vmax - vmin)
1134: (8)                   else:
1135: (12)                      return vmin + value * (vmax - vmin)
1136: (4)               def autoscale(self, A):
1137: (8)                   """Set *vmin*, *vmax* to min, max of *A*."""
1138: (8)                   with self.callbacks.blocked():
1139: (12)                      self.vmin = self.vmax = None
1140: (12)                      self.autoscale_None(A)
1141: (8)                   self._changed()
1142: (4)               def autoscale_None(self, A):
1143: (8)                   """If *vmin* or *vmax* are not set, use the min/max of *A* to set
them."""
1144: (8)                   A = np.asanyarray(A)
1145: (8)                   if isinstance(A, np.ma.MaskedArray):
1146: (12)                      if A.mask is False or not A.mask.shape:
1147: (16)                          A = A.data
1148: (8)                   if self.vmin is None and A.size:
1149: (12)                      self.vmin = A.min()
1150: (8)                   if self.vmax is None and A.size:
1151: (12)                      self.vmax = A.max()
1152: (4)               def scaled(self):
1153: (8)                   """Return whether *vmin* and *vmax* are both set."""
1154: (8)                   return self.vmin is not None and self.vmax is not None
1155: (0)           class TwoSlopeNorm(Normalize):
1156: (4)               def __init__(self, vcenter, vmin=None, vmax=None):
1157: (8)                   """
1158: (8)                   Normalize data with a set center.
1159: (8)                   Useful when mapping data with an unequal rates of change around a
1160: (8)                   conceptual center, e.g., data that range from -2 to 4, with 0 as
1161: (8)                   the midpoint.
1162: (8)                   Parameters
1163: (8)                   ----------
1164: (8)                   vcenter : float
1165: (12)                      The data value that defines ``0.5`` in the normalization.
1166: (8)                   vmin : float, optional
1167: (12)                      The data value that defines ``0.0`` in the normalization.
1168: (12)                      Defaults to the min value of the dataset.
1169: (8)                   vmax : float, optional
1170: (12)                      The data value that defines ``1.0`` in the normalization.
```

```
1171: (12)                      Defaults to the max value of the dataset.
1172: (8)                   Examples
1173: (8)                   --------
1174: (8)                   This maps data value -4000 to 0., 0 to 0.5, and +10000 to 1.0; data
1175: (8)                   between is linearly interpolated::
1176: (12)                      >>> import matplotlib.colors as mcolors
1177: (12)                      >>> offset = mcolors.TwoSlopeNorm(vmin=-4000.,
1178: (12)                      ...                               vcenter=0., vmax=10000)
1179: (12)                      >>> data = [-4000., -2000., 0., 2500., 5000., 7500., 10000.]
1180: (12)                      >>> offset(data)
1181: (12)                      array([0., 0.25, 0.5, 0.625, 0.75, 0.875, 1.0])
1182: (8)                   """
1183: (8)                   super().__init__(vmin=vmin, vmax=vmax)
1184: (8)                   self._vcenter = vcenter
1185: (8)                   if vcenter is not None and vmax is not None and vcenter >= vmax:
1186: (12)                      raise ValueError('vmin, vcenter, and vmax must be in '
1187: (29)                                       'ascending order')
1188: (8)                   if vcenter is not None and vmin is not None and vcenter <= vmin:
1189: (12)                      raise ValueError('vmin, vcenter, and vmax must be in '
1190: (29)                                       'ascending order')
1191: (4)               @property
1192: (4)               def vcenter(self):
1193: (8)                   return self._vcenter
1194: (4)               @vcenter.setter
1195: (4)               def vcenter(self, value):
1196: (8)                   if value != self._vcenter:
1197: (12)                      self._vcenter = value
1198: (12)                      self._changed()
1199: (4)               def autoscale_None(self, A):
1200: (8)                   """
1201: (8)                   Get vmin and vmax.
1202: (8)                   If vcenter isn't in the range [vmin, vmax], either vmin or vmax
1203: (8)                   is expanded so that vcenter lies in the middle of the modified range
1204: (8)                   [vmin, vmax].
1205: (8)                   """
1206: (8)                   super().autoscale_None(A)
1207: (8)                   if self.vmin >= self.vcenter:
1208: (12)                      self.vmin = self.vcenter - (self.vmax - self.vcenter)
1209: (8)                   if self.vmax <= self.vcenter:
1210: (12)                      self.vmax = self.vcenter + (self.vcenter - self.vmin)
1211: (4)               def __call__(self, value, clip=None):
1212: (8)                   """
1213: (8)                   Map value to the interval [0, 1]. The *clip* argument is unused.
1214: (8)                   """
1215: (8)                   result, is_scalar = self.process_value(value)
1216: (8)                   self.autoscale_None(result)  # sets self.vmin, self.vmax if None
1217: (8)                   if not self.vmin <= self.vcenter <= self.vmax:
1218: (12)                      raise ValueError("vmin, vcenter, vmax must increase
monotonically")
1219: (8)                   result = np.ma.masked_array(
1220: (12)                      np.interp(result, [self.vmin, self.vcenter, self.vmax],
1221: (22)                                [0, 0.5, 1], left=-np.inf, right=np.inf),
1222: (12)                      mask=np.ma.getmask(result))
1223: (8)                   if is_scalar:
1224: (12)                      result = np.atleast_1d(result)[0]
1225: (8)                   return result
1226: (4)               def inverse(self, value):
1227: (8)                   if not self.scaled():
1228: (12)                      raise ValueError("Not invertible until both vmin and vmax are
set")
1229: (8)                   (vmin,), _ = self.process_value(self.vmin)
1230: (8)                   (vmax,), _ = self.process_value(self.vmax)
1231: (8)                   (vcenter,), _ = self.process_value(self.vcenter)
1232: (8)                   result = np.interp(value, [0, 0.5, 1], [vmin, vcenter, vmax],
1233: (27)                                     left=-np.inf, right=np.inf)
1234: (8)                   return result
1235: (0)           class CenteredNorm(Normalize):
1236: (4)               def __init__(self, vcenter=0, halfrange=None, clip=False):
1237: (8)                   """
```

```
(8)                 Normalize symmetrical data around a center (0 by default).
(8)                 Unlike `TwoSlopeNorm`, `CenteredNorm` applies an equal rate of change
(8)                 around the center.
(8)                 Useful when mapping symmetrical data around a conceptual center
(8)                 e.g., data that range from -2 to 4, with 0 as the midpoint, and
(8)                 with equal rates of change around that midpoint.
(8)                 Parameters
(8)                 ----------
(8)                 vcenter : float, default: 0
(12)                    The data value that defines ``0.5`` in the normalization.
(8)                 halfrange : float, optional
(12)                    The range of data values that defines a range of ``0.5`` in the
(12)                    normalization, so that *vcenter* - *halfrange* is ``0.0`` and
(12)                    *vcenter* + *halfrange* is ``1.0`` in the normalization.
(12)                    Defaults to the largest absolute difference to *vcenter* for
(12)                    the values in the dataset.
(8)                 clip : bool, default: False
(12)                    Determines the behavior for mapping values outside the range
(12)                    ``[vmin, vmax]``.
(12)                    If clipping is off, values outside the range ``[vmin, vmax]`` are
(12)                    also transformed, resulting in values outside ``[0, 1]``.  This
(12)                    behavior is usually desirable, as colormaps can mark these *under*
(12)                    and *over* values with specific colors.
(12)                    If clipping is on, values below *vmin* are mapped to 0 and values
(12)                    above *vmax* are mapped to 1. Such values become indistinguishable
(12)                    from regular boundary values, which may cause misinterpretation of
(12)                    the data.
(8)                 Examples
(8)                 --------
(8)                 This maps data values -2 to 0.25, 0 to 0.5, and 4 to 1.0
(8)                 (assuming equal rates of change above and below 0.0):
(12)                    >>> import matplotlib.colors as mcolors
(12)                    >>> norm = mcolors.CenteredNorm(halfrange=4.0)
(12)                    >>> data = [-2., 0., 4.]
(12)                    >>> norm(data)
(12)                    array([0.25, 0.5 , 1.  ])
(8)                 """
(8)                 super().__init__(vmin=None, vmax=None, clip=clip)
(8)                 self._vcenter = vcenter
(8)                 self.halfrange = halfrange
(4)             def autoscale(self, A):
(8)                 """
(8)                 Set *halfrange* to ``max(abs(A-vcenter))``, then set *vmin* and
*vmax*.
(8)                 """
(8)                 A = np.asanyarray(A)
(8)                 self.halfrange = max(self._vcenter-A.min(),
(29)                                     A.max()-self._vcenter)
(4)             def autoscale_None(self, A):
(8)                 """Set *vmin* and *vmax*."""
(8)                 A = np.asanyarray(A)
(8)                 if self.halfrange is None and A.size:
(12)                    self.autoscale(A)
(4)             @property
(4)             def vmin(self):
(8)                 return self._vmin
(4)             @vmin.setter
(4)             def vmin(self, value):
(8)                 value = _sanitize_extrema(value)
(8)                 if value != self._vmin:
(12)                    self._vmin = value
(12)                    self._vmax = 2*self.vcenter - value
(12)                    self._changed()
(4)             @property
(4)             def vmax(self):
(8)                 return self._vmax
(4)             @vmax.setter
(4)             def vmax(self, value):
(8)                 value = _sanitize_extrema(value)
```

```
(8)                 if value != self._vmax:
(12)                    self._vmax = value
(12)                    self._vmin = 2*self.vcenter - value
(12)                    self._changed()
(4)             @property
(4)             def vcenter(self):
(8)                 return self._vcenter
(4)             @vcenter.setter
(4)             def vcenter(self, vcenter):
(8)                 if vcenter != self._vcenter:
(12)                    self._vcenter = vcenter
(12)                    self.halfrange = self.halfrange
(12)                    self._changed()
(4)             @property
(4)             def halfrange(self):
(8)                 if self.vmin is None or self.vmax is None:
(12)                    return None
(8)                 return (self.vmax - self.vmin) / 2
(4)             @halfrange.setter
(4)             def halfrange(self, halfrange):
(8)                 if halfrange is None:
(12)                    self.vmin = None
(12)                    self.vmax = None
(8)                 else:
(12)                    self.vmin = self.vcenter - abs(halfrange)
(12)                    self.vmax = self.vcenter + abs(halfrange)
(0)         def make_norm_from_scale(scale_cls, base_norm_cls=None, *, init=None):
(4)             """
(4)             Decorator for building a `.Normalize` subclass from a `~.scale.ScaleBase`
(4)             subclass.
(4)             After ::
(8)                 @make_norm_from_scale(scale_cls)
(8)                 class norm_cls(Normalize):
(12)                    ...
(4)             *norm_cls* is filled with methods so that normalization computations are
(4)             forwarded to *scale_cls* (i.e., *scale_cls* is the scale that would be
used
(4)             for the colorbar of a mappable normalized with *norm_cls*).
(4)             If *init* is not passed, then the constructor signature of *norm_cls*
(4)             will be ``norm_cls(vmin=None, vmax=None, clip=False)``; these three
(4)             parameters will be forwarded to the base class (``Normalize.__init__``),
(4)             and a *scale_cls* object will be initialized with no arguments (other than
(4)             a dummy axis).
(4)             If the *scale_cls* constructor takes additional parameters, then *init*
(4)             should be passed to `make_norm_from_scale`.  It is a callable which is
(4)             *only* used for its signature.  First, this signature will become the
(4)             signature of *norm_cls*.  Second, the *norm_cls* constructor will bind the
(4)             parameters passed to it using this signature, extract the bound *vmin*,
(4)             *vmax*, and *clip* values, pass those to ``Normalize.__init__``, and
(4)             forward the remaining bound values (including any defaults defined by the
(4)             signature) to the *scale_cls* constructor.
(4)             """
(4)             if base_norm_cls is None:
(8)                 return functools.partial(make_norm_from_scale, scale_cls, init=init)
(4)             if isinstance(scale_cls, functools.partial):
(8)                 scale_args = scale_cls.args
(8)                 scale_kwargs_items = tuple(scale_cls.keywords.items())
(8)                 scale_cls = scale_cls.func
(4)             else:
(8)                 scale_args = scale_kwargs_items = ()
(4)             if init is None:
(8)                 def init(vmin=None, vmax=None, clip=False): pass
(4)             return _make_norm_from_scale(
(8)                 scale_cls, scale_args, scale_kwargs_items,
(8)                 base_norm_cls, inspect.signature(init))
(0)         @functools.cache
(0)         def _make_norm_from_scale(
(4)             scale_cls, scale_args, scale_kwargs_items,
(4)             base_norm_cls, bound_init_signature,
```

```
1374: (0)            ):
1375: (4)                """
1376: (4)                Helper for `make_norm_from_scale`.
1377: (4)                This function is split out to enable caching (in particular so that
1378: (4)                different unpickles reuse the same class).  In order to do so,
1379: (4)                - ``functools.partial`` *scale_cls* is expanded into ``func, args,
kwargs``
1380: (6)                  to allow memoizing returned norms (partial instances always compare
1381: (6)                  unequal, but we can check identity based on ``func, args, kwargs``;
1382: (4)                - *init* is replaced by *init_signature*, as signatures are picklable,
1383: (6)                  unlike to arbitrary lambdas.
1384: (4)                """
1385: (4)                class Norm(base_norm_cls):
1386: (8)                    def __reduce__(self):
1387: (12)                       cls = type(self)
1388: (12)                       try:
1389: (16)                           if cls is getattr(importlib.import_module(cls.__module__),
1390: (34)                                             cls.__qualname__):
1391: (20)                               return (_create_empty_object_of_class, (cls,), vars(self))
1392: (12)                       except (ImportError, AttributeError):
1393: (16)                           pass
1394: (12)                       return (_picklable_norm_constructor,
1395: (20)                               (scale_cls, scale_args, scale_kwargs_items,
1396: (21)                                base_norm_cls, bound_init_signature),
1397: (20)                               vars(self))
1398: (8)                    def __init__(self, *args, **kwargs):
1399: (12)                       ba = bound_init_signature.bind(*args, **kwargs)
1400: (12)                       ba.apply_defaults()
1401: (12)                       super().__init__(
1402: (16)                           **{k: ba.arguments.pop(k) for k in ["vmin", "vmax", "clip"]})
1403: (12)                       self._scale = functools.partial(
1404: (16)                           scale_cls, *scale_args, **dict(scale_kwargs_items))(
1405: (20)                               axis=None, **ba.arguments)
1406: (12)                       self._trf = self._scale.get_transform()
1407: (8)                    __init__.__signature__ = bound_init_signature.replace(parameters=[
1408: (12)                       inspect.Parameter("self",
inspect.Parameter.POSITIONAL_OR_KEYWORD),
1409: (12)                       *bound_init_signature.parameters.values()])
1410: (8)                    def __call__(self, value, clip=None):
1411: (12)                       value, is_scalar = self.process_value(value)
1412: (12)                       if self.vmin is None or self.vmax is None:
1413: (16)                           self.autoscale_None(value)
1414: (12)                       if self.vmin > self.vmax:
1415: (16)                           raise ValueError("vmin must be less or equal to vmax")
1416: (12)                       if self.vmin == self.vmax:
1417: (16)                           return np.full_like(value, 0)
1418: (12)                       if clip is None:
1419: (16)                           clip = self.clip
1420: (12)                       if clip:
1421: (16)                           value = np.clip(value, self.vmin, self.vmax)
1422: (12)                       t_value = self._trf.transform(value).reshape(np.shape(value))
1423: (12)                       t_vmin, t_vmax = self._trf.transform([self.vmin, self.vmax])
1424: (12)                       if not np.isfinite([t_vmin, t_vmax]).all():
1425: (16)                           raise ValueError("Invalid vmin or vmax")
1426: (12)                       t_value -= t_vmin
1427: (12)                       t_value /= (t_vmax - t_vmin)
1428: (12)                       t_value = np.ma.masked_invalid(t_value, copy=False)
1429: (12)                       return t_value[0] if is_scalar else t_value
1430: (8)                    def inverse(self, value):
1431: (12)                       if not self.scaled():
1432: (16)                           raise ValueError("Not invertible until scaled")
1433: (12)                       if self.vmin > self.vmax:
1434: (16)                           raise ValueError("vmin must be less or equal to vmax")
1435: (12)                       t_vmin, t_vmax = self._trf.transform([self.vmin, self.vmax])
1436: (12)                       if not np.isfinite([t_vmin, t_vmax]).all():
1437: (16)                           raise ValueError("Invalid vmin or vmax")
1438: (12)                       value, is_scalar = self.process_value(value)
1439: (12)                       rescaled = value * (t_vmax - t_vmin)
1440: (12)                       rescaled += t_vmin
```

```
1441: (12)                      value = (self._trf
1442: (21)                               .inverted()
1443: (21)                               .transform(rescaled)
1444: (21)                               .reshape(np.shape(value)))
1445: (12)                      return value[0] if is_scalar else value
1446: (8)                   def autoscale_None(self, A):
1447: (12)                      in_trf_domain = np.extract(np.isfinite(self._trf.transform(A)), A)
1448: (12)                      if in_trf_domain.size == 0:
1449: (16)                          in_trf_domain = np.ma.masked
1450: (12)                      return super().autoscale_None(in_trf_domain)
1451: (4)               if base_norm_cls is Normalize:
1452: (8)                   Norm.__name__ = f"{scale_cls.__name__}Norm"
1453: (8)                   Norm.__qualname__ = f"{scale_cls.__qualname__}Norm"
1454: (4)               else:
1455: (8)                   Norm.__name__ = base_norm_cls.__name__
1456: (8)                   Norm.__qualname__ = base_norm_cls.__qualname__
1457: (4)               Norm.__module__ = base_norm_cls.__module__
1458: (4)               Norm.__doc__ = base_norm_cls.__doc__
1459: (4)               return Norm
1460: (0)           def _create_empty_object_of_class(cls):
1461: (4)               return cls.__new__(cls)
1462: (0)           def _picklable_norm_constructor(*args):
1463: (4)               return _create_empty_object_of_class(_make_norm_from_scale(*args))
1464: (0)           @make_norm_from_scale(
1465: (4)               scale.FuncScale,
1466: (4)               init=lambda functions, vmin=None, vmax=None, clip=False: None)
1467: (0)           class FuncNorm(Normalize):
1468: (4)               """
1469: (4)               Arbitrary normalization using functions for the forward and inverse.
1470: (4)               Parameters
1471: (4)               ----------
1472: (4)               functions : (callable, callable)
1473: (8)                   two-tuple of the forward and inverse functions for the normalization.
1474: (8)                   The forward function must be monotonic.
1475: (8)                   Both functions must have the signature ::
1476: (11)                     def forward(values: array-like) -> array-like
1477: (4)               vmin, vmax : float or None
1478: (8)                   If *vmin* and/or *vmax* is not given, they are initialized from the
1479: (8)                   minimum and maximum value, respectively, of the first input
1480: (8)                   processed; i.e., ``__call__(A)`` calls ``autoscale_None(A)``.
1481: (4)               clip : bool, default: False
1482: (8)                   Determines the behavior for mapping values outside the range
1483: (8)                   ``[vmin, vmax]``.
1484: (8)                   If clipping is off, values outside the range ``[vmin, vmax]`` are also
1485: (8)                   transformed by the function, resulting in values outside ``[0, 1]``.
1486: (8)                   This behavior is usually desirable, as colormaps can mark these
*under*
1487: (8)                   and *over* values with specific colors.
1488: (8)                   If clipping is on, values below *vmin* are mapped to 0 and values
above
1489: (8)                   *vmax* are mapped to 1. Such values become indistinguishable from
1490: (8)                   regular boundary values, which may cause misinterpretation of the
data.
1491: (4)               """
1492: (0)           LogNorm = make_norm_from_scale(
1493: (4)               functools.partial(scale.LogScale, nonpositive="mask"))(Normalize)
1494: (0)           LogNorm.__name__ = LogNorm.__qualname__ = "LogNorm"
1495: (0)           LogNorm.__doc__ = "Normalize a given value to the 0-1 range on a log scale."
1496: (0)           @make_norm_from_scale(
1497: (4)               scale.SymmetricalLogScale,
1498: (4)               init=lambda linthresh, linscale=1., vmin=None, vmax=None, clip=False, *,
1499: (16)                          base=10: None)
1500: (0)           class SymLogNorm(Normalize):
1501: (4)               """
1502: (4)               The symmetrical logarithmic scale is logarithmic in both the
1503: (4)               positive and negative directions from the origin.
1504: (4)               Since the values close to zero tend toward infinity, there is a
1505: (4)               need to have a range around zero that is linear.  The parameter
1506: (4)               *linthresh* allows the user to specify the size of this range
```

```
(4)                 (-*linthresh*, *linthresh*).
(4)                 Parameters
(4)                 ----------
(4)                 linthresh : float
(8)                     The range within which the plot is linear (to avoid having the plot
(8)                     go to infinity around zero).
(4)                 linscale : float, default: 1
(8)                     This allows the linear range (-*linthresh* to *linthresh*) to be
(8)                     stretched relative to the logarithmic range. Its value is the
(8)                     number of decades to use for each half of the linear range. For
(8)                     example, when *linscale* == 1.0 (the default), the space used for
(8)                     the positive and negative halves of the linear range will be equal
(8)                     to one decade in the logarithmic range.
(4)                 base : float, default: 10
(4)                 """
(4)                 @property
(4)                 def linthresh(self):
(8)                     return self._scale.linthresh
(4)                 @linthresh.setter
(4)                 def linthresh(self, value):
(8)                     self._scale.linthresh = value
(0)             @make_norm_from_scale(
(4)                 scale.AsinhScale,
(4)                 init=lambda linear_width=1, vmin=None, vmax=None, clip=False: None)
(0)             class AsinhNorm(Normalize):
(4)                 """
(4)                 The inverse hyperbolic sine scale is approximately linear near
(4)                 the origin, but becomes logarithmic for larger positive
(4)                 or negative values. Unlike the `SymLogNorm`, the transition between
(4)                 these linear and logarithmic regions is smooth, which may reduce
(4)                 the risk of visual artifacts.
(4)                 .. note::
(7)                    This API is provisional and may be revised in the future
(7)                    based on early user feedback.
(4)                 Parameters
(4)                 ----------
(4)                 linear_width : float, default: 1
(8)                     The effective width of the linear region, beyond which
(8)                     the transformation becomes asymptotically logarithmic
(4)                 """
(4)                 @property
(4)                 def linear_width(self):
(8)                     return self._scale.linear_width
(4)                 @linear_width.setter
(4)                 def linear_width(self, value):
(8)                     self._scale.linear_width = value
(0)             class PowerNorm(Normalize):
(4)                 r"""
(4)                 Linearly map a given value to the 0-1 range and then apply
(4)                 a power-law normalization over that range.
(4)                 Parameters
(4)                 ----------
(4)                 gamma : float
(8)                     Power law exponent.
(4)                 vmin, vmax : float or None
(8)                     If *vmin* and/or *vmax* is not given, they are initialized from the
(8)                     minimum and maximum value, respectively, of the first input
(8)                     processed; i.e., ``__call__(A)`` calls ``autoscale_None(A)``.
(4)                 clip : bool, default: False
(8)                     Determines the behavior for mapping values outside the range
(8)                     ``[vmin, vmax]``.
(8)                     If clipping is off, values above *vmax* are transformed by the power
(8)                     function, resulting in values above 1, and values below *vmin* are
linearly
(8)                     transformed resulting in values below 0. This behavior is usually
desirable, as
(8)                     colormaps can mark these *under* and *over* values with specific
colors.
(8)                     If clipping is on, values below *vmin* are mapped to 0 and values
```

```
above
1573: (8)                   *vmax* are mapped to 1. Such values become indistinguishable from
1574: (8)                   regular boundary values, which may cause misinterpretation of the
data.
1575: (4)               Notes
1576: (4)               -----
1577: (4)               The normalization formula is
1578: (4)               .. math::
1579: (8)                   \left ( \frac{x - v_{min}}{v_{max}  - v_{min}} \right )^{\gamma}
1580: (4)               For input values below *vmin*, gamma is set to one.
1581: (4)               """
1582: (4)               def __init__(self, gamma, vmin=None, vmax=None, clip=False):
1583: (8)                   super().__init__(vmin, vmax, clip)
1584: (8)                   self.gamma = gamma
1585: (4)               def __call__(self, value, clip=None):
1586: (8)                   if clip is None:
1587: (12)                      clip = self.clip
1588: (8)                   result, is_scalar = self.process_value(value)
1589: (8)                   self.autoscale_None(result)
1590: (8)                   gamma = self.gamma
1591: (8)                   vmin, vmax = self.vmin, self.vmax
1592: (8)                   if vmin > vmax:
1593: (12)                      raise ValueError("minvalue must be less than or equal to
maxvalue")
1594: (8)                   elif vmin == vmax:
1595: (12)                      result.fill(0)
1596: (8)                   else:
1597: (12)                      if clip:
1598: (16)                          mask = np.ma.getmask(result)
1599: (16)                          result = np.ma.array(np.clip(result.filled(vmax), vmin, vmax),
1600: (37)                                               mask=mask)
1601: (12)                      resdat = result.data
1602: (12)                      resdat -= vmin
1603: (12)                      resdat /= (vmax - vmin)
1604: (12)                      resdat[resdat > 0] = np.power(resdat[resdat > 0], gamma)
1605: (12)                      result = np.ma.array(resdat, mask=result.mask, copy=False)
1606: (8)                   if is_scalar:
1607: (12)                      result = result[0]
1608: (8)                   return result
1609: (4)               def inverse(self, value):
1610: (8)                   if not self.scaled():
1611: (12)                      raise ValueError("Not invertible until scaled")
1612: (8)                   result, is_scalar = self.process_value(value)
1613: (8)                   gamma = self.gamma
1614: (8)                   vmin, vmax = self.vmin, self.vmax
1615: (8)                   resdat = result.data
1616: (8)                   resdat[resdat > 0] = np.power(resdat[resdat > 0], 1 / gamma)
1617: (8)                   resdat *= (vmax - vmin)
1618: (8)                   resdat += vmin
1619: (8)                   result = np.ma.array(resdat, mask=result.mask, copy=False)
1620: (8)                   if is_scalar:
1621: (12)                      result = result[0]
1622: (8)                   return result
1623: (0)           class BoundaryNorm(Normalize):
1624: (4)               """
1625: (4)               Generate a colormap index based on discrete intervals.
1626: (4)               Unlike `Normalize` or `LogNorm`, `BoundaryNorm` maps values to integers
1627: (4)               instead of to the interval 0-1.
1628: (4)               """
1629: (4)               def __init__(self, boundaries, ncolors, clip=False, *, extend='neither'):
1630: (8)                   """
1631: (8)                   Parameters
1632: (8)                   ----------
1633: (8)                   boundaries : array-like
1634: (12)                      Monotonically increasing sequence of at least 2 bin edges:  data
1635: (12)                      falling in the n-th bin will be mapped to the n-th color.
1636: (8)                   ncolors : int
1637: (12)                      Number of colors in the colormap to be used.
1638: (8)                   clip : bool, optional
```

```
(12)                    If clip is ``True``, out of range values are mapped to 0 if they
(12)                    are below ``boundaries[0]`` or mapped to ``ncolors - 1`` if they
(12)                    are above ``boundaries[-1]``.
(12)                    If clip is ``False``, out of range values are mapped to -1 if
(12)                    they are below ``boundaries[0]`` or mapped to *ncolors* if they
are
(12)                    above ``boundaries[-1]``. These are then converted to valid
indices
(12)                    by `Colormap.__call__`.
(8)                 extend : {'neither', 'both', 'min', 'max'}, default: 'neither'
(12)                    Extend the number of bins to include one or both of the
(12)                    regions beyond the boundaries.  For example, if ``extend``
(12)                    is 'min', then the color to which the region between the first
(12)                    pair of boundaries is mapped will be distinct from the first
(12)                    color in the colormap, and by default a
(12)                    `~matplotlib.colorbar.Colorbar` will be drawn with
(12)                    the triangle extension on the left or lower end.
(8)                 Notes
(8)                 -----
(8)                 If there are fewer bins (including extensions) than colors, then the
(8)                 color index is chosen by linearly interpolating the ``[0, nbins - 1]``
(8)                 range onto the ``[0, ncolors - 1]`` range, effectively skipping some
(8)                 colors in the middle of the colormap.
(8)                 """
(8)                 if clip and extend != 'neither':
(12)                    raise ValueError("'clip=True' is not compatible with 'extend'")
(8)                 super().__init__(vmin=boundaries[0], vmax=boundaries[-1], clip=clip)
(8)                 self.boundaries = np.asarray(boundaries)
(8)                 self.N = len(self.boundaries)
(8)                 if self.N < 2:
(12)                    raise ValueError("You must provide at least 2 boundaries "
(29)                                     f"(1 region) but you passed in {boundaries!r}")
(8)                 self.Ncmap = ncolors
(8)                 self.extend = extend
(8)                 self._scale = None  # don't use the default scale.
(8)                 self._n_regions = self.N - 1  # number of colors needed
(8)                 self._offset = 0
(8)                 if extend in ('min', 'both'):
(12)                    self._n_regions += 1
(12)                    self._offset = 1
(8)                 if extend in ('max', 'both'):
(12)                    self._n_regions += 1
(8)                 if self._n_regions > self.Ncmap:
(12)                    raise ValueError(f"There are {self._n_regions} color bins "
(29)                                     "including extensions, but ncolors = "
(29)                                     f"{ncolors}; ncolors must equal or exceed the "
(29)                                     "number of bins")
(4)             def __call__(self, value, clip=None):
(8)                 """
(8)                 This method behaves similarly to `.Normalize.__call__`, except that it
(8)                 returns integers or arrays of int16.
(8)                 """
(8)                 if clip is None:
(12)                    clip = self.clip
(8)                 xx, is_scalar = self.process_value(value)
(8)                 mask = np.ma.getmaskarray(xx)
(8)                 xx = np.atleast_1d(xx.filled(self.vmax + 1))
(8)                 if clip:
(12)                    np.clip(xx, self.vmin, self.vmax, out=xx)
(12)                    max_col = self.Ncmap - 1
(8)                 else:
(12)                    max_col = self.Ncmap
(8)                 iret = np.digitize(xx, self.boundaries) - 1 + self._offset
(8)                 if self.Ncmap > self._n_regions:
(12)                    if self._n_regions == 1:
(16)                        iret[iret == 0] = (self.Ncmap - 1) // 2
(12)                    else:
(16)                        iret = (self.Ncmap - 1) / (self._n_regions - 1) * iret
(8)                 iret = iret.astype(np.int16)
```

```
(8)                 iret[xx < self.vmin] = -1
(8)                 iret[xx >= self.vmax] = max_col
(8)                 ret = np.ma.array(iret, mask=mask)
(8)                 if is_scalar:
(12)                    ret = int(ret[0])  # assume python scalar
(8)                 return ret
(4)             def inverse(self, value):
(8)                 """
(8)                 Raises
(8)                 ------
(8)                 ValueError
(12)                    BoundaryNorm is not invertible, so calling this method will always
(12)                    raise an error
(8)                 """
(8)                 raise ValueError("BoundaryNorm is not invertible")
(0)         class NoNorm(Normalize):
(4)             """
(4)             Dummy replacement for `Normalize`, for the case where we want to use
(4)             indices directly in a `~matplotlib.cm.ScalarMappable`.
(4)             """
(4)             def __call__(self, value, clip=None):
(8)                 if np.iterable(value):
(12)                    return np.ma.array(value)
(8)                 return value
(4)             def inverse(self, value):
(8)                 if np.iterable(value):
(12)                    return np.ma.array(value)
(8)                 return value
(0)         def rgb_to_hsv(arr):
(4)             """
(4)             Convert an array of float RGB values (in the range [0, 1]) to HSV values.
(4)             Parameters
(4)             ----------
(4)             arr : (..., 3) array-like
(7)                All values must be in the range [0, 1]
(4)             Returns
(4)             -------
(4)             (..., 3) `~numpy.ndarray`
(7)                Colors converted to HSV values in range [0, 1]
(4)             """
(4)             arr = np.asarray(arr)
(4)             if arr.shape[-1] != 3:
(8)                 raise ValueError("Last dimension of input array must be 3; "
(25)                                 f"shape {arr.shape} was found.")
(4)             in_shape = arr.shape
(4)             arr = np.array(
(8)                 arr, copy=False,
(8)                 dtype=np.promote_types(arr.dtype, np.float32),  # Don't work on ints.
(8)                 ndmin=2,  # In case input was 1D.
(4)             )
(4)             out = np.zeros_like(arr)
(4)             arr_max = arr.max(-1)
(4)             ipos = arr_max > 0
(4)             delta = np.ptp(arr, -1)
(4)             s = np.zeros_like(delta)
(4)             s[ipos] = delta[ipos] / arr_max[ipos]
(4)             ipos = delta > 0
(4)             idx = (arr[..., 0] == arr_max) & ipos
(4)             out[idx, 0] = (arr[idx, 1] - arr[idx, 2]) / delta[idx]
(4)             idx = (arr[..., 1] == arr_max) & ipos
(4)             out[idx, 0] = 2. + (arr[idx, 2] - arr[idx, 0]) / delta[idx]
(4)             idx = (arr[..., 2] == arr_max) & ipos
(4)             out[idx, 0] = 4. + (arr[idx, 0] - arr[idx, 1]) / delta[idx]
(4)             out[..., 0] = (out[..., 0] / 6.0) % 1.0
(4)             out[..., 1] = s
(4)             out[..., 2] = arr_max
(4)             return out.reshape(in_shape)
(0)         def hsv_to_rgb(hsv):
(4)             """
```

```
(4)             Convert HSV values to RGB.
(4)             Parameters
(4)             ----------
(4)             hsv : (..., 3) array-like
(7)                All values assumed to be in range [0, 1]
(4)             Returns
(4)             -------
(4)             (..., 3) `~numpy.ndarray`
(7)                Colors converted to RGB values in range [0, 1]
(4)             """
(4)             hsv = np.asarray(hsv)
(4)             if hsv.shape[-1] != 3:
(8)                 raise ValueError("Last dimension of input array must be 3; "
(25)                                 f"shape {hsv.shape} was found.")
(4)             in_shape = hsv.shape
(4)             hsv = np.array(
(8)                 hsv, copy=False,
(8)                 dtype=np.promote_types(hsv.dtype, np.float32),  # Don't work on ints.
(8)                 ndmin=2,  # In case input was 1D.
(4)             )
(4)             h = hsv[..., 0]
(4)             s = hsv[..., 1]
(4)             v = hsv[..., 2]
(4)             r = np.empty_like(h)
(4)             g = np.empty_like(h)
(4)             b = np.empty_like(h)
(4)             i = (h * 6.0).astype(int)
(4)             f = (h * 6.0) - i
(4)             p = v * (1.0 - s)
(4)             q = v * (1.0 - s * f)
(4)             t = v * (1.0 - s * (1.0 - f))
(4)             idx = i % 6 == 0
(4)             r[idx] = v[idx]
(4)             g[idx] = t[idx]
(4)             b[idx] = p[idx]
(4)             idx = i == 1
(4)             r[idx] = q[idx]
(4)             g[idx] = v[idx]
(4)             b[idx] = p[idx]
(4)             idx = i == 2
(4)             r[idx] = p[idx]
(4)             g[idx] = v[idx]
(4)             b[idx] = t[idx]
(4)             idx = i == 3
(4)             r[idx] = p[idx]
(4)             g[idx] = q[idx]
(4)             b[idx] = v[idx]
(4)             idx = i == 4
(4)             r[idx] = t[idx]
(4)             g[idx] = p[idx]
(4)             b[idx] = v[idx]
(4)             idx = i == 5
(4)             r[idx] = v[idx]
(4)             g[idx] = p[idx]
(4)             b[idx] = q[idx]
(4)             idx = s == 0
(4)             r[idx] = v[idx]
(4)             g[idx] = v[idx]
(4)             b[idx] = v[idx]
(4)             rgb = np.stack([r, g, b], axis=-1)
(4)             return rgb.reshape(in_shape)
(0)         def _vector_magnitude(arr):
(4)             sum_sq = 0
(4)             for i in range(arr.shape[-1]):
(8)                 sum_sq += arr[..., i, np.newaxis] ** 2
(4)             return np.sqrt(sum_sq)
(0)         class LightSource:
(4)             """
(4)             Create a light source coming from the specified azimuth and elevation.
```

```
(4)                 Angles are in degrees, with the azimuth measured
(4)                 clockwise from north and elevation up from the zero plane of the surface.
(4)                 `shade` is used to produce "shaded" RGB values for a data array.
(4)                 `shade_rgb` can be used to combine an RGB image with an elevation map.
(4)                 `hillshade` produces an illumination map of a surface.
(4)                 """
(4)                 def __init__(self, azdeg=315, altdeg=45, hsv_min_val=0, hsv_max_val=1,
(17)                         hsv_min_sat=1, hsv_max_sat=0):
(8)                     """
(8)                     Specify the azimuth (measured clockwise from south) and altitude
(8)                     (measured up from the plane of the surface) of the light source
(8)                     in degrees.
(8)                     Parameters
(8)                     ----------
(8)                     azdeg : float, default: 315 degrees (from the northwest)
(12)                        The azimuth (0-360, degrees clockwise from North) of the light
(12)                        source.
(8)                     altdeg : float, default: 45 degrees
(12)                        The altitude (0-90, degrees up from horizontal) of the light
(12)                        source.
(8)                     hsv_min_val : number, default: 0
(12)                        The minimum value ("v" in "hsv") that the *intensity* map can
shift the
(12)                        output image to.
(8)                     hsv_max_val : number, default: 1
(12)                        The maximum value ("v" in "hsv") that the *intensity* map can
shift the
(12)                        output image to.
(8)                     hsv_min_sat : number, default: 1
(12)                        The minimum saturation value that the *intensity* map can shift
the output
(12)                        image to.
(8)                     hsv_max_sat : number, default: 0
(12)                        The maximum saturation value that the *intensity* map can shift
the output
(12)                        image to.
(8)                     Notes
(8)                     -----
(8)                     For backwards compatibility, the parameters *hsv_min_val*,
(8)                     *hsv_max_val*, *hsv_min_sat*, and *hsv_max_sat* may be supplied at
(8)                     initialization as well.  However, these parameters will only be used
if
(8)                     "blend_mode='hsv'" is passed into `shade` or `shade_rgb`.
(8)                     See the documentation for `blend_hsv` for more details.
(8)                     """
(8)                     self.azdeg = azdeg
(8)                     self.altdeg = altdeg
(8)                     self.hsv_min_val = hsv_min_val
(8)                     self.hsv_max_val = hsv_max_val
(8)                     self.hsv_min_sat = hsv_min_sat
(8)                     self.hsv_max_sat = hsv_max_sat
(4)                 @property
(4)                 def direction(self):
(8)                     """The unit vector direction towards the light source."""
(8)                     az = np.radians(90 - self.azdeg)
(8)                     alt = np.radians(self.altdeg)
(8)                     return np.array([
(12)                        np.cos(az) * np.cos(alt),
(12)                        np.sin(az) * np.cos(alt),
(12)                        np.sin(alt)
(8)                     ])
(4)                 def hillshade(self, elevation, vert_exag=1, dx=1, dy=1, fraction=1.):
(8)                     """
(8)                     Calculate the illumination intensity for a surface using the defined
(8)                     azimuth and elevation for the light source.
(8)                     This computes the normal vectors for the surface, and then passes them
(8)                     on to `shade_normals`
(8)                     Parameters
(8)                     ----------
```

```
(8)                 elevation : 2D array-like
(12)                    The height values used to generate an illumination map
(8)                 vert_exag : number, optional
(12)                    The amount to exaggerate the elevation values by when calculating
(12)                    illumination. This can be used either to correct for differences
in
(12)                    units between the x-y coordinate system and the elevation
(12)                    coordinate system (e.g. decimal degrees vs. meters) or to
(12)                    exaggerate or de-emphasize topographic effects.
(8)                 dx : number, optional
(12)                    The x-spacing (columns) of the input *elevation* grid.
(8)                 dy : number, optional
(12)                    The y-spacing (rows) of the input *elevation* grid.
(8)                 fraction : number, optional
(12)                    Increases or decreases the contrast of the hillshade.  Values
(12)                    greater than one will cause intermediate values to move closer to
(12)                    full illumination or shadow (and clipping any values that move
(12)                    beyond 0 or 1). Note that this is not visually or mathematically
(12)                    the same as vertical exaggeration.
(8)                 Returns
(8)                 -------
(8)                 `~numpy.ndarray`
(12)                    A 2D array of illumination values between 0-1, where 0 is
(12)                    completely in shadow and 1 is completely illuminated.
(8)                 """
(8)                 dy = -dy
(8)                 e_dy, e_dx = np.gradient(vert_exag * elevation, dy, dx)
(8)                 normal = np.empty(elevation.shape + (3,)).view(type(elevation))
(8)                 normal[..., 0] = -e_dx
(8)                 normal[..., 1] = -e_dy
(8)                 normal[..., 2] = 1
(8)                 normal /= _vector_magnitude(normal)
(8)                 return self.shade_normals(normal, fraction)
(4)             def shade_normals(self, normals, fraction=1.):
(8)                 """
(8)                 Calculate the illumination intensity for the normal vectors of a
(8)                 surface using the defined azimuth and elevation for the light source.
(8)                 Imagine an artificial sun placed at infinity in some azimuth and
(8)                 elevation position illuminating our surface. The parts of the surface
(8)                 that slope toward the sun should brighten while those sides facing
away
(8)                 should become darker.
(8)                 Parameters
(8)                 ----------
(8)                 fraction : number, optional
(12)                    Increases or decreases the contrast of the hillshade.  Values
(12)                    greater than one will cause intermediate values to move closer to
(12)                    full illumination or shadow (and clipping any values that move
(12)                    beyond 0 or 1). Note that this is not visually or mathematically
(12)                    the same as vertical exaggeration.
(8)                 Returns
(8)                 -------
(8)                 `~numpy.ndarray`
(12)                    A 2D array of illumination values between 0-1, where 0 is
(12)                    completely in shadow and 1 is completely illuminated.
(8)                 """
(8)                 intensity = normals.dot(self.direction)
(8)                 imin, imax = intensity.min(), intensity.max()
(8)                 intensity *= fraction
(8)                 if (imax - imin) > 1e-6:
(12)                    intensity -= imin
(12)                    intensity /= (imax - imin)
(8)                 intensity = np.clip(intensity, 0, 1)
(8)                 return intensity
(4)             def shade(self, data, cmap, norm=None, blend_mode='overlay', vmin=None,
(14)                  vmax=None, vert_exag=1, dx=1, dy=1, fraction=1, **kwargs):
(8)                 """
(8)                 Combine colormapped data values with an illumination intensity map
(8)                 (a.k.a.  "hillshade") of the values.
```

```
        Parameters
        ----------
        data : 2D array-like
            The height values used to generate a shaded map.
        cmap : `~matplotlib.colors.Colormap`
            The colormap used to color the *data* array. Note that this must be

            a `~matplotlib.colors.Colormap` instance.  For example, rather than

            passing in ``cmap='gist_earth'``, use
            ``cmap=plt.get_cmap('gist_earth')`` instead.
        norm : `~matplotlib.colors.Normalize` instance, optional
            The normalization used to scale values before colormapping. If
            None, the input will be linearly scaled between its min and max.
        blend_mode : {'hsv', 'overlay', 'soft'} or callable, optional
            The type of blending used to combine the colormapped data
            values with the illumination intensity.  Default is
            "overlay".  Note that for most topographic surfaces,
            "overlay" or "soft" appear more visually realistic. If a
            user-defined function is supplied, it is expected to
            combine an (M, N, 3) RGB array of floats (ranging 0 to 1) with
            an (M, N, 1) hillshade array (also 0 to 1).  (Call signature
            ``func(rgb, illum, **kwargs)``) Additional kwargs supplied
            to this function will be passed on to the *blend_mode*
            function.
        vmin : float or None, optional
            The minimum value used in colormapping *data*. If *None* the
            minimum value in *data* is used. If *norm* is specified, then this
            argument will be ignored.
        vmax : float or None, optional
            The maximum value used in colormapping *data*. If *None* the
            maximum value in *data* is used. If *norm* is specified, then this
            argument will be ignored.
        vert_exag : number, optional
            The amount to exaggerate the elevation values by when calculating
            illumination. This can be used either to correct for differences in

            units between the x-y coordinate system and the elevation
            coordinate system (e.g. decimal degrees vs. meters) or to
            exaggerate or de-emphasize topography.
        dx : number, optional
            The x-spacing (columns) of the input *elevation* grid.
        dy : number, optional
            The y-spacing (rows) of the input *elevation* grid.
        fraction : number, optional
            Increases or decreases the contrast of the hillshade.  Values
            greater than one will cause intermediate values to move closer to
            full illumination or shadow (and clipping any values that move
            beyond 0 or 1). Note that this is not visually or mathematically
            the same as vertical exaggeration.
        **kwargs
            Additional kwargs are passed on to the *blend_mode* function.
        Returns
        -------
        `~numpy.ndarray`
            An (M, N, 4) array of floats ranging between 0-1.
        """
        if vmin is None:
            vmin = data.min()
        if vmax is None:
            vmax = data.max()
        if norm is None:
            norm = Normalize(vmin=vmin, vmax=vmax)
        rgb0 = cmap(norm(data))
        rgb1 = self.shade_rgb(rgb0, elevation=data, blend_mode=blend_mode,
                              vert_exag=vert_exag, dx=dx, dy=dy,
                              fraction=fraction, **kwargs)
        rgb0[..., :3] = rgb1[..., :3]
        return rgb0
```

```
(4)     def shade_rgb(self, rgb, elevation, fraction=1., blend_mode='hsv',
(18)                   vert_exag=1, dx=1, dy=1, **kwargs):
(8)         """
(8)         Use this light source to adjust the colors of the *rgb* input array to
(8)         give the impression of a shaded relief map with the given *elevation*.
(8)         Parameters
(8)         ----------
(8)         rgb : array-like
(12)            An (M, N, 3) RGB array, assumed to be in the range of 0 to 1.
(8)         elevation : array-like
(12)            An (M, N) array of the height values used to generate a shaded map.
(8)         fraction : number
(12)            Increases or decreases the contrast of the hillshade.  Values
(12)            greater than one will cause intermediate values to move closer to
(12)            full illumination or shadow (and clipping any values that move
(12)            beyond 0 or 1). Note that this is not visually or mathematically
(12)            the same as vertical exaggeration.
(8)         blend_mode : {'hsv', 'overlay', 'soft'} or callable, optional
(12)            The type of blending used to combine the colormapped data values
(12)            with the illumination intensity.  For backwards compatibility, this
(12)            defaults to "hsv". Note that for most topographic surfaces,
(12)            "overlay" or "soft" appear more visually realistic. If a
(12)            user-defined function is supplied, it is expected to combine an
(12)            (M, N, 3) RGB array of floats (ranging 0 to 1) with an (M, N, 1)
(12)            hillshade array (also 0 to 1).  (Call signature
(12)            ``func(rgb, illum, **kwargs)``)
(12)            Additional kwargs supplied to this function will be passed on to
(12)            the *blend_mode* function.
(8)         vert_exag : number, optional
(12)            The amount to exaggerate the elevation values by when calculating
(12)            illumination. This can be used either to correct for differences in
(12)            units between the x-y coordinate system and the elevation
(12)            coordinate system (e.g. decimal degrees vs. meters) or to
(12)            exaggerate or de-emphasize topography.
(8)         dx : number, optional
(12)            The x-spacing (columns) of the input *elevation* grid.
(8)         dy : number, optional
(12)            The y-spacing (rows) of the input *elevation* grid.
(8)         **kwargs
(12)            Additional kwargs are passed on to the *blend_mode* function.
(8)         Returns
(8)         -------
(8)         `~numpy.ndarray`
(12)            An (m, n, 3) array of floats ranging between 0-1.
(8)         """
(8)         intensity = self.hillshade(elevation, vert_exag, dx, dy, fraction)
(8)         intensity = intensity[..., np.newaxis]
(8)         lookup = {
(16)                'hsv': self.blend_hsv,
(16)                'soft': self.blend_soft_light,
(16)                'overlay': self.blend_overlay,
(16)                }
(8)         if blend_mode in lookup:
(12)            blend = lookup[blend_mode](rgb, intensity, **kwargs)
(8)         else:
(12)            try:
(16)                blend = blend_mode(rgb, intensity, **kwargs)
(12)            except TypeError as err:
(16)                raise ValueError('"blend_mode" must be callable or one of '
(33)                                 f'{lookup.keys}') from err
(8)         if np.ma.is_masked(intensity):
(12)            mask = intensity.mask[..., 0]
(12)            for i in range(3):
(16)                blend[..., i][mask] = rgb[..., i][mask]
(8)         return blend
(4)     def blend_hsv(self, rgb, intensity, hsv_max_sat=None, hsv_max_val=None,
```

```
                                hsv_min_val=None, hsv_min_sat=None):
                    """
                    Take the input data array, convert to HSV values in the given colormap,
                    then adjust those color values to give the impression of a shaded
                    relief map with a specified light source.  RGBA values are returned,
                    which can then be used to plot the shaded image with imshow.
                    The color of the resulting image will be darkened by moving the (s, v)
                    values (in HSV colorspace) toward (hsv_min_sat, hsv_min_val) in the
                    shaded regions, or lightened by sliding (s, v) toward (hsv_max_sat,
                    hsv_max_val) in regions that are illuminated.  The default extremes are
                    chose so that completely shaded points are nearly black (s = 1, v = 0)
                    and completely illuminated points are nearly white (s = 0, v = 1).
                    Parameters
                    ----------
                    rgb : `~numpy.ndarray`
                        An (M, N, 3) RGB array of floats ranging from 0 to 1 (color image).
                    intensity : `~numpy.ndarray`
                        An (M, N, 1) array of floats ranging from 0 to 1 (grayscale image).
                    hsv_max_sat : number, optional
                        The maximum saturation value that the *intensity* map can shift the output
                        image to. If not provided, use the value provided upon initialization.
                    hsv_min_sat : number, optional
                        The minimum saturation value that the *intensity* map can shift the output
                        image to. If not provided, use the value provided upon initialization.
                    hsv_max_val : number, optional
                        The maximum value ("v" in "hsv") that the *intensity* map can shift the
                        output image to. If not provided, use the value provided upon
                        initialization.
                    hsv_min_val : number, optional
                        The minimum value ("v" in "hsv") that the *intensity* map can shift the
                        output image to. If not provided, use the value provided upon
                        initialization.
                    Returns
                    -------
                    `~numpy.ndarray`
                        An (M, N, 3) RGB array representing the combined images.
                    """
                    if hsv_max_sat is None:
                        hsv_max_sat = self.hsv_max_sat
                    if hsv_max_val is None:
                        hsv_max_val = self.hsv_max_val
                    if hsv_min_sat is None:
                        hsv_min_sat = self.hsv_min_sat
                    if hsv_min_val is None:
                        hsv_min_val = self.hsv_min_val
                    intensity = intensity[..., 0]
                    intensity = 2 * intensity - 1
                    hsv = rgb_to_hsv(rgb[:, :, 0:3])
                    hue, sat, val = np.moveaxis(hsv, -1, 0)
                    np.putmask(sat, (np.abs(sat) > 1.e-10) & (intensity > 0),
                               (1 - intensity) * sat + intensity * hsv_max_sat)
                    np.putmask(sat, (np.abs(sat) > 1.e-10) & (intensity < 0),
                               (1 + intensity) * sat - intensity * hsv_min_sat)
                    np.putmask(val, intensity > 0,
                               (1 - intensity) * val + intensity * hsv_max_val)
                    np.putmask(val, intensity < 0,
                               (1 + intensity) * val - intensity * hsv_min_val)
                    np.clip(hsv[:, :, 1:], 0, 1, out=hsv[:, :, 1:])
                    return hsv_to_rgb(hsv)
```

```
    def blend_soft_light(self, rgb, intensity):
        """
        Combine an RGB image with an intensity map using "soft light" blending,
        using the "pegtop" formula.
        Parameters
        ----------
        rgb : `~numpy.ndarray`
            An (M, N, 3) RGB array of floats ranging from 0 to 1 (color image).
        intensity : `~numpy.ndarray`
            An (M, N, 1) array of floats ranging from 0 to 1 (grayscale image).
        Returns
        -------
        `~numpy.ndarray`
            An (M, N, 3) RGB array representing the combined images.
        """
        return 2 * intensity * rgb + (1 - 2 * intensity) * rgb**2
    def blend_overlay(self, rgb, intensity):
        """
        Combine an RGB image with an intensity map using "overlay" blending.
        Parameters
        ----------
        rgb : `~numpy.ndarray`
            An (M, N, 3) RGB array of floats ranging from 0 to 1 (color image).
        intensity : `~numpy.ndarray`
            An (M, N, 1) array of floats ranging from 0 to 1 (grayscale image).
        Returns
        -------
        ndarray
            An (M, N, 3) RGB array representing the combined images.
        """
        low = 2 * intensity * rgb
        high = 1 - 2 * (1 - intensity) * (1 - rgb)
        return np.where(rgb <= 0.5, low, high)
def from_levels_and_colors(levels, colors, extend='neither'):
    """
    A helper routine to generate a cmap and a norm instance which
    behave similar to contourf's levels and colors arguments.
    Parameters
    ----------
    levels : sequence of numbers
        The quantization levels used to construct the `BoundaryNorm`.
        Value ``v`` is quantized to level ``i`` if ``lev[i] <= v < lev[i+1]``.
    colors : sequence of colors
        The fill color to use for each level. If *extend* is "neither" there
        must be ``n_level - 1`` colors. For an *extend* of "min" or "max" add
        one extra color, and for an *extend* of "both" add two colors.
    extend : {'neither', 'min', 'max', 'both'}, optional
        The behaviour when a value falls out of range of the given levels.
        See `~.Axes.contourf` for details.
    Returns
    -------
    cmap : `~matplotlib.colors.Colormap`
    norm : `~matplotlib.colors.Normalize`
    """
    slice_map = {
        'both': slice(1, -1),
        'min': slice(1, None),
        'max': slice(0, -1),
        'neither': slice(0, None),
    }
    _api.check_in_list(slice_map, extend=extend)
    color_slice = slice_map[extend]
    n_data_colors = len(levels) - 1
    n_expected = n_data_colors + color_slice.start - (color_slice.stop or 0)
```

```
2230: (4)                if len(colors) != n_expected:
2231: (8)                    raise ValueError(
2232: (12)                       f'With extend == {extend!r} and {len(levels)} levels, '
2233: (12)                       f'expected {n_expected} colors, but got {len(colors)}')
2234: (4)                cmap = ListedColormap(colors[color_slice], N=n_data_colors)
2235: (4)                if extend in ['min', 'both']:
2236: (8)                    cmap.set_under(colors[0])
2237: (4)                else:
2238: (8)                    cmap.set_under('none')
2239: (4)                if extend in ['max', 'both']:
2240: (8)                    cmap.set_over(colors[-1])
2241: (4)                else:
2242: (8)                    cmap.set_over('none')
2243: (4)                cmap.colorbar_extend = extend
2244: (4)                norm = BoundaryNorm(levels, ncolors=n_data_colors)
2245: (4)                return cmap, norm
```

File 34 - container.py:

```
1: (0)              from matplotlib import cbook
2: (0)              from matplotlib.artist import Artist
3: (0)              class Container(tuple):
4: (4)                  """
5: (4)                  Base class for containers.
6: (4)                  Containers are classes that collect semantically related Artists such as
7: (4)                  the bars of a bar plot.
8: (4)                  """
9: (4)                  def __repr__(self):
10: (8)                     return f"<{type(self).__name__} object of {len(self)} artists>"
11: (4)                 def __new__(cls, *args, **kwargs):
12: (8)                     return tuple.__new__(cls, args[0])
13: (4)                 def __init__(self, kl, label=None):
14: (8)                     self._callbacks = cbook.CallbackRegistry(signals=["pchanged"])
15: (8)                     self._remove_method = None
16: (8)                     self._label = str(label) if label is not None else None
17: (4)                 def remove(self):
18: (8)                     for c in cbook.flatten(
19: (16)                            self, scalarp=lambda x: isinstance(x, Artist)):
20: (12)                        if c is not None:
21: (16)                            c.remove()
22: (8)                     if self._remove_method:
23: (12)                        self._remove_method(self)
24: (4)                 def get_children(self):
25: (8)                     return [child for child in cbook.flatten(self) if child is not None]
26: (4)                 get_label = Artist.get_label
27: (4)                 set_label = Artist.set_label
28: (4)                 add_callback = Artist.add_callback
29: (4)                 remove_callback = Artist.remove_callback
30: (4)                 pchanged = Artist.pchanged
31: (0)             class BarContainer(Container):
32: (4)                 """
33: (4)                 Container for the artists of bar plots (e.g. created by `.Axes.bar`).
34: (4)                 The container can be treated as a tuple of the *patches* themselves.
35: (4)                 Additionally, you can access these and further parameters by the
36: (4)                 attributes.
37: (4)                 Attributes
38: (4)                 ----------
39: (4)                 patches : list of :class:`~matplotlib.patches.Rectangle`
40: (8)                     The artists of the bars.
41: (4)                 errorbar : None or :class:`~matplotlib.container.ErrorbarContainer`
42: (8)                     A container for the error bar artists if error bars are present.
43: (8)                     *None* otherwise.
44: (4)                 datavalues : None or array-like
45: (8)                     The underlying data values corresponding to the bars.
46: (4)                 orientation : {'vertical', 'horizontal'}, default: None
47: (8)                     If 'vertical', the bars are assumed to be vertical.
48: (8)                     If 'horizontal', the bars are assumed to be horizontal.
```

```
49: (4)                 """
50: (4)                 def __init__(self, patches, errorbar=None, *, datavalues=None,
51: (17)                            orientation=None, **kwargs):
52: (8)                     self.patches = patches
53: (8)                     self.errorbar = errorbar
54: (8)                     self.datavalues = datavalues
55: (8)                     self.orientation = orientation
56: (8)                     super().__init__(patches, **kwargs)
57: (0)             class ErrorbarContainer(Container):
58: (4)                 """
59: (4)                 Container for the artists of error bars (e.g. created by
`.Axes.errorbar`).
60: (4)                 The container can be treated as the *lines* tuple itself.
61: (4)                 Additionally, you can access these and further parameters by the
62: (4)                 attributes.
63: (4)                 Attributes
64: (4)                 ----------
65: (4)                 lines : tuple
66: (8)                     Tuple of ``(data_line, caplines, barlinecols)``.
67: (8)                     - data_line : A `~matplotlib.lines.Line2D` instance of x, y plot
markers
68: (10)                      and/or line.
69: (8)                     - caplines : A tuple of `~matplotlib.lines.Line2D` instances of the
error
70: (10)                      bar caps.
71: (8)                     - barlinecols : A tuple of `~matplotlib.collections.LineCollection`
with the
72: (10)                      horizontal and vertical error ranges.
73: (4)                 has_xerr, has_yerr : bool
74: (8)                     ``True`` if the errorbar has x/y errors.
75: (4)                 """
76: (4)                 def __init__(self, lines, has_xerr=False, has_yerr=False, **kwargs):
77: (8)                     self.lines = lines
78: (8)                     self.has_xerr = has_xerr
79: (8)                     self.has_yerr = has_yerr
80: (8)                     super().__init__(lines, **kwargs)
81: (0)             class StemContainer(Container):
82: (4)                 """
83: (4)                 Container for the artists created in a :meth:`.Axes.stem` plot.
84: (4)                 The container can be treated like a namedtuple ``(markerline, stemlines,
85: (4)                 baseline)``.
86: (4)                 Attributes
87: (4)                 ----------
88: (4)                 markerline : `~matplotlib.lines.Line2D`
89: (8)                     The artist of the markers at the stem heads.
90: (4)                 stemlines : `~matplotlib.collections.LineCollection`
91: (8)                     The artists of the vertical lines for all stems.
92: (4)                 baseline : `~matplotlib.lines.Line2D`
93: (8)                     The artist of the horizontal baseline.
94: (4)                 """
95: (4)                 def __init__(self, markerline_stemlines_baseline, **kwargs):
96: (8)                     """
97: (8)                     Parameters
98: (8)                     ----------
99: (8)                     markerline_stemlines_baseline : tuple
100: (12)                       Tuple of ``(markerline, stemlines, baseline)``.
101: (12)                       ``markerline`` contains the `.Line2D` of the markers,
102: (12)                       ``stemlines`` is a `.LineCollection` of the main lines,
103: (12)                       ``baseline`` is the `.Line2D` of the baseline.
104: (8)                     """
105: (8)                     markerline, stemlines, baseline = markerline_stemlines_baseline
106: (8)                     self.markerline = markerline
107: (8)                     self.stemlines = stemlines
108: (8)                     self.baseline = baseline
109: (8)                     super().__init__(markerline_stemlines_baseline, **kwargs)

----------------------------------------

File 35 - contour.py:
```

```
1: (0)              """
2: (0)              Classes to support contour plotting and labelling for the Axes class.
3: (0)              """
4: (0)              from contextlib import ExitStack
5: (0)              import functools
6: (0)              import math
7: (0)              from numbers import Integral
8: (0)              import numpy as np
9: (0)              from numpy import ma
10: (0)             import matplotlib as mpl
11: (0)             from matplotlib import _api, _docstring
12: (0)             from matplotlib.backend_bases import MouseButton
13: (0)             from matplotlib.lines import Line2D
14: (0)             from matplotlib.path import Path
15: (0)             from matplotlib.text import Text
16: (0)             import matplotlib.ticker as ticker
17: (0)             import matplotlib.cm as cm
18: (0)             import matplotlib.colors as mcolors
19: (0)             import matplotlib.collections as mcoll
20: (0)             import matplotlib.font_manager as font_manager
21: (0)             import matplotlib.cbook as cbook
22: (0)             import matplotlib.patches as mpatches
23: (0)             import matplotlib.transforms as mtransforms
24: (0)             def _contour_labeler_event_handler(cs, inline, inline_spacing, event):
25: (4)                 canvas = cs.axes.figure.canvas
26: (4)                 is_button = event.name == "button_press_event"
27: (4)                 is_key = event.name == "key_press_event"
28: (4)                 if (is_button and event.button == MouseButton.MIDDLE
29: (12)                        or is_key and event.key in ["escape", "enter"]):
30: (8)                     canvas.stop_event_loop()
31: (4)                 elif (is_button and event.button == MouseButton.RIGHT
32: (10)                      or is_key and event.key in ["backspace", "delete"]):
33: (8)                     if not inline:
34: (12)                        cs.pop_label()
35: (12)                        canvas.draw()
36: (4)                 elif (is_button and event.button == MouseButton.LEFT
37: (10)                      or is_key and event.key is not None):
38: (8)                     if cs.axes.contains(event)[0]:
39: (12)                        cs.add_label_near(event.x, event.y, transform=False,
40: (30)                                          inline=inline, inline_spacing=inline_spacing)
41: (12)                        canvas.draw()
42: (0)             class ContourLabeler:
43: (4)                 """Mixin to provide labelling capability to `.ContourSet`."""
44: (4)                 def clabel(self, levels=None, *,
45: (15)                           fontsize=None, inline=True, inline_spacing=5, fmt=None,
46: (15)                           colors=None, use_clabeltext=False, manual=False,
47: (15)                           rightside_up=True, zorder=None):
48: (8)                     """
49: (8)                     Label a contour plot.
50: (8)                     Adds labels to line contours in this `.ContourSet` (which inherits
from
51: (8)                     this mixin class).
52: (8)                     Parameters
53: (8)                     ----------
54: (8)                     levels : array-like, optional
55: (12)                        A list of level values, that should be labeled. The list must be
56: (12)                        a subset of ``cs.levels``. If not given, all levels are labeled.
57: (8)                     fontsize : str or float, default: :rc:`font.size`
58: (12)                        Size in points or relative size e.g., 'smaller', 'x-large'.
59: (12)                        See `.Text.set_size` for accepted string values.
60: (8)                     colors : :mpltype:`color` or colors or None, default: None
61: (12)                        The label colors:
62: (12)                        - If *None*, the color of each label matches the color of
63: (14)                          the corresponding contour.
64: (12)                        - If one string color, e.g., *colors* = 'r' or *colors* =
65: (14)                          'red', all labels will be plotted in this color.
66: (12)                        - If a tuple of colors (string, float, RGB, etc), different labels
67: (14)                          will be plotted in different colors in the order specified.
```

```
68: (8)                     inline : bool, default: True
69: (12)                        If ``True`` the underlying contour is removed where the label is
70: (12)                        placed.
71: (8)                     inline_spacing : float, default: 5
72: (12)                        Space in pixels to leave on each side of label when placing
inline.
73: (12)                        This spacing will be exact for labels at locations where the
74: (12)                        contour is straight, less so for labels on curved contours.
75: (8)                     fmt : `.Formatter` or str or callable or dict, optional
76: (12)                        How the levels are formatted:
77: (12)                        - If a `.Formatter`, it is used to format all levels at once,
using
78: (14)                          its `.Formatter.format_ticks` method.
79: (12)                        - If a str, it is interpreted as a %-style format string.
80: (12)                        - If a callable, it is called with one level at a time and should
81: (14)                          return the corresponding label.
82: (12)                        - If a dict, it should directly map levels to labels.
83: (12)                        The default is to use a standard `.ScalarFormatter`.
84: (8)                     manual : bool or iterable, default: False
85: (12)                        If ``True``, contour labels will be placed manually using
86: (12)                        mouse clicks. Click the first button near a contour to
87: (12)                        add a label, click the second button (or potentially both
88: (12)                        mouse buttons at once) to finish adding labels. The third
89: (12)                        button can be used to remove the last label added, but
90: (12)                        only if labels are not inline. Alternatively, the keyboard
91: (12)                        can be used to select label locations (enter to end label
92: (12)                        placement, delete or backspace act like the third mouse button,
93: (12)                        and any other key will select a label location).
94: (12)                        *manual* can also be an iterable object of (x, y) tuples.
95: (12)                        Contour labels will be created as if mouse is clicked at each
96: (12)                        (x, y) position.
97: (8)                     rightside_up : bool, default: True
98: (12)                        If ``True``, label rotations will always be plus
99: (12)                        or minus 90 degrees from level.
100: (8)                    use_clabeltext : bool, default: False
101: (12)                       If ``True``, use `.Text.set_transform_rotates_text` to ensure that
102: (12)                       label rotation is updated whenever the Axes aspect changes.
103: (8)                    zorder : float or None, default: ``(2 + contour.get_zorder())``
104: (12)                       zorder of the contour labels.
105: (8)                    Returns
106: (8)                    -------
107: (8)                    labels
108: (12)                       A list of `.Text` instances for the labels.
109: (8)                    """
110: (8)                    if fmt is None:
111: (12)                       fmt = ticker.ScalarFormatter(useOffset=False)
112: (12)                       fmt.create_dummy_axis()
113: (8)                    self.labelFmt = fmt
114: (8)                    self._use_clabeltext = use_clabeltext
115: (8)                    self.labelManual = manual
116: (8)                    self.rightside_up = rightside_up
117: (8)                    self._clabel_zorder = 2 + self.get_zorder() if zorder is None else
zorder
118: (8)                    if levels is None:
119: (12)                       levels = self.levels
120: (12)                       indices = list(range(len(self.cvalues)))
121: (8)                    else:
122: (12)                       levlabs = list(levels)
123: (12)                       indices, levels = [], []
124: (12)                       for i, lev in enumerate(self.levels):
125: (16)                           if lev in levlabs:
126: (20)                               indices.append(i)
127: (20)                               levels.append(lev)
128: (12)                       if len(levels) < len(levlabs):
129: (16)                           raise ValueError(f"Specified levels {levlabs} don't match "
130: (33)                                            f"available levels {self.levels}")
131: (8)                    self.labelLevelList = levels
132: (8)                    self.labelIndiceList = indices
133: (8)                    self._label_font_props = font_manager.FontProperties(size=fontsize)
```

```
134: (8)                    if colors is None:
135: (12)                       self.labelMappable = self
136: (12)                       self.labelCValueList = np.take(self.cvalues, self.labelIndiceList)
137: (8)                    else:
138: (12)                       cmap = mcolors.ListedColormap(colors, N=len(self.labelLevelList))
139: (12)                       self.labelCValueList = list(range(len(self.labelLevelList)))
140: (12)                       self.labelMappable = cm.ScalarMappable(cmap=cmap,
141: (51)                                                              norm=mcolors.NoNorm())
142: (8)                    self.labelXYs = []
143: (8)                    if np.iterable(manual):
144: (12)                       for x, y in manual:
145: (16)                           self.add_label_near(x, y, inline, inline_spacing)
146: (8)                    elif manual:
147: (12)                       print('Select label locations manually using first mouse button.')
148: (12)                       print('End manual selection with second mouse button.')
149: (12)                       if not inline:
150: (16)                           print('Remove last label by clicking third mouse button.')
151: (12)                       mpl._blocking_input.blocking_input_loop(
152: (16)                           self.axes.figure, ["button_press_event", "key_press_event"],
153: (16)                           timeout=-1, handler=functools.partial(
154: (20)                               _contour_labeler_event_handler,
155: (20)                               self, inline, inline_spacing))
156: (8)                    else:
157: (12)                       self.labels(inline, inline_spacing)
158: (8)                    return cbook.silent_list('text.Text', self.labelTexts)
159: (4)                def print_label(self, linecontour, labelwidth):
160: (8)                    """Return whether a contour is long enough to hold a label."""
161: (8)                    return (len(linecontour) > 10 * labelwidth
162: (16)                           or (len(linecontour)
163: (20)                               and (np.ptp(linecontour, axis=0) > 1.2 *
labelwidth).any()))
164: (4)                def too_close(self, x, y, lw):
165: (8)                    """Return whether a label is already near this location."""
166: (8)                    thresh = (1.2 * lw) ** 2
167: (8)                    return any((x - loc[0]) ** 2 + (y - loc[1]) ** 2 < thresh
168: (19)                              for loc in self.labelXYs)
169: (4)                def _get_nth_label_width(self, nth):
170: (8)                    """Return the width of the *nth* label, in pixels."""
171: (8)                    fig = self.axes.figure
172: (8)                    renderer = fig._get_renderer()
173: (8)                    return (Text(0, 0,
174: (21)                                self.get_text(self.labelLevelList[nth], self.labelFmt),
175: (21)                                figure=fig, fontproperties=self._label_font_props)
176: (16)                           .get_window_extent(renderer).width)
177: (4)                def get_text(self, lev, fmt):
178: (8)                    """Get the text of the label."""
179: (8)                    if isinstance(lev, str):
180: (12)                       return lev
181: (8)                    elif isinstance(fmt, dict):
182: (12)                       return fmt.get(lev, '%1.3f')
183: (8)                    elif callable(getattr(fmt, "format_ticks", None)):
184: (12)                       return fmt.format_ticks([*self.labelLevelList, lev])[-1]
185: (8)                    elif callable(fmt):
186: (12)                       return fmt(lev)
187: (8)                    else:
188: (12)                       return fmt % lev
189: (4)                def locate_label(self, linecontour, labelwidth):
190: (8)                    """
191: (8)                    Find good place to draw a label (relatively flat part of the contour).
192: (8)                    """
193: (8)                    ctr_size = len(linecontour)
194: (8)                    n_blocks = int(np.ceil(ctr_size / labelwidth)) if labelwidth > 1 else
1
195: (8)                    block_size = ctr_size if n_blocks == 1 else int(labelwidth)
196: (8)                    xx = np.resize(linecontour[:, 0], (n_blocks, block_size))
197: (8)                    yy = np.resize(linecontour[:, 1], (n_blocks, block_size))
198: (8)                    yfirst = yy[:, :1]
199: (8)                    ylast = yy[:, -1:]
200: (8)                    xfirst = xx[:, :1]
```

```
201: (8)                    xlast = xx[:, -1:]
202: (8)                    s = (yfirst - yy) * (xlast - xfirst) - (xfirst - xx) * (ylast -
yfirst)
203: (8)                    l = np.hypot(xlast - xfirst, ylast - yfirst)
204: (8)                    with np.errstate(divide='ignore', invalid='ignore'):
205: (12)                       distances = (abs(s) / l).sum(axis=-1)
206: (8)                    hbsize = block_size // 2
207: (8)                    adist = np.argsort(distances)
208: (8)                    for idx in np.append(adist, adist[0]):
209: (12)                       x, y = xx[idx, hbsize], yy[idx, hbsize]
210: (12)                       if not self.too_close(x, y, labelwidth):
211: (16)                           break
212: (8)                    return x, y, (idx * block_size + hbsize) % ctr_size
213: (4)                def _split_path_and_get_label_rotation(self, path, idx, screen_pos, lw,
spacing=5):
214: (8)                    """
215: (8)                    Prepare for insertion of a label at index *idx* of *path*.
216: (8)                    Parameters
217: (8)                    ----------
218: (8)                    path : Path
219: (12)                       The path where the label will be inserted, in data space.
220: (8)                    idx : int
221: (12)                       The vertex index after which the label will be inserted.
222: (8)                    screen_pos : (float, float)
223: (12)                       The position where the label will be inserted, in screen space.
224: (8)                    lw : float
225: (12)                       The label width, in screen space.
226: (8)                    spacing : float
227: (12)                       Extra spacing around the label, in screen space.
228: (8)                    Returns
229: (8)                    -------
230: (8)                    path : Path
231: (12)                       The path, broken so that the label can be drawn over it.
232: (8)                    angle : float
233: (12)                       The rotation of the label.
234: (8)                    Notes
235: (8)                    -----
236: (8)                    Both tasks are done together to avoid calculating path lengths
multiple times,
237: (8)                    which is relatively costly.
238: (8)                    The method used here involves computing the path length along the
contour in
239: (8)                    pixel coordinates and then looking (label width / 2) away from central
point to
240: (8)                    determine rotation and then to break contour if desired.  The extra
spacing is
241: (8)                    taken into account when breaking the path, but not when computing the
angle.
242: (8)                    """
243: (8)                    if hasattr(self, "_old_style_split_collections"):
244: (12)                       vis = False
245: (12)                       for coll in self._old_style_split_collections:
246: (16)                           vis |= coll.get_visible()
247: (16)                           coll.remove()
248: (12)                       self.set_visible(vis)
249: (12)                       del self._old_style_split_collections  # Invalidate them.
250: (8)                    xys = path.vertices
251: (8)                    codes = path.codes
252: (8)                    pos = self.get_transform().inverted().transform(screen_pos)
253: (8)                    if not np.allclose(pos, xys[idx]):
254: (12)                       xys = np.insert(xys, idx, pos, axis=0)
255: (12)                       codes = np.insert(codes, idx, Path.LINETO)
256: (8)                    movetos = (codes == Path.MOVETO).nonzero()[0]
257: (8)                    start = movetos[movetos <= idx][-1]
258: (8)                    try:
259: (12)                       stop = movetos[movetos > idx][0]
260: (8)                    except IndexError:
261: (12)                       stop = len(codes)
262: (8)                    cc_xys = xys[start:stop]
```

```
263: (8)                    idx -= start
264: (8)                    is_closed_path = codes[stop - 1] == Path.CLOSEPOLY
265: (8)                    if is_closed_path:
266: (12)                       cc_xys = np.concatenate([cc_xys[idx:-1], cc_xys[:idx+1]])
267: (12)                       idx = 0
268: (8)                    def interp_vec(x, xp, fp): return [np.interp(x, xp, col) for col in
fp.T]
269: (8)                    screen_xys = self.get_transform().transform(cc_xys)
270: (8)                    path_cpls = np.insert(
271: (12)                       np.cumsum(np.hypot(*np.diff(screen_xys, axis=0).T)), 0, 0)
272: (8)                    path_cpls -= path_cpls[idx]
273: (8)                    target_cpls = np.array([-lw/2, lw/2])
274: (8)                    if is_closed_path:  # For closed paths, target from the other end.
275: (12)                       target_cpls[0] += (path_cpls[-1] - path_cpls[0])
276: (8)                    (sx0, sx1), (sy0, sy1) = interp_vec(target_cpls, path_cpls,
screen_xys)
277: (8)                    angle = np.rad2deg(np.arctan2(sy1 - sy0, sx1 - sx0))  # Screen space.
278: (8)                    if self.rightside_up:  # Fix angle so text is never upside-down
279: (12)                       angle = (angle + 90) % 180 - 90
280: (8)                    target_cpls += [-spacing, +spacing]  # Expand range by spacing.
281: (8)                    i0, i1 = np.interp(target_cpls, path_cpls, range(len(path_cpls)),
282: (27)                                      left=-1, right=-1)
283: (8)                    i0 = math.floor(i0)
284: (8)                    i1 = math.ceil(i1)
285: (8)                    (x0, x1), (y0, y1) = interp_vec(target_cpls, path_cpls, cc_xys)
286: (8)                    new_xy_blocks = []
287: (8)                    new_code_blocks = []
288: (8)                    if is_closed_path:
289: (12)                       if i0 != -1 and i1 != -1:
290: (16)                           points = cc_xys[i1:i0+1]
291: (16)                           new_xy_blocks.extend([[(x1, y1)], points, [(x0, y0)]])
292: (16)                           nlines = len(points) + 1
293: (16)                           new_code_blocks.extend([[Path.MOVETO], [Path.LINETO] *
nlines])
294: (8)                    else:
295: (12)                       if i0 != -1:
296: (16)                           new_xy_blocks.extend([cc_xys[:i0 + 1], [(x0, y0)]])
297: (16)                           new_code_blocks.extend([[Path.MOVETO], [Path.LINETO] * (i0 +
1)])
298: (12)                       if i1 != -1:
299: (16)                           new_xy_blocks.extend([[(x1, y1)], cc_xys[i1:]])
300: (16)                           new_code_blocks.extend([
301: (20)                               [Path.MOVETO], [Path.LINETO] * (len(cc_xys) - i1)])
302: (8)                    xys = np.concatenate([xys[:start], *new_xy_blocks, xys[stop:]])
303: (8)                    codes = np.concatenate([codes[:start], *new_code_blocks,
codes[stop:]])
304: (8)                    return angle, Path(xys, codes)
305: (4)                @_api.deprecated("3.8")
306: (4)                def calc_label_rot_and_inline(self, slc, ind, lw, lc=None, spacing=5):
307: (8)                    """
308: (8)                    Calculate the appropriate label rotation given the linecontour
309: (8)                    coordinates in screen units, the index of the label location and the
310: (8)                    label width.
311: (8)                    If *lc* is not None or empty, also break contours and compute
312: (8)                    inlining.
313: (8)                    *spacing* is the empty space to leave around the label, in pixels.
314: (8)                    Both tasks are done together to avoid calculating path lengths
315: (8)                    multiple times, which is relatively costly.
316: (8)                    The method used here involves computing the path length along the
317: (8)                    contour in pixel coordinates and then looking approximately (label
318: (8)                    width / 2) away from central point to determine rotation and then to
319: (8)                    break contour if desired.
320: (8)                    """
321: (8)                    if lc is None:
322: (12)                       lc = []
323: (8)                    hlw = lw / 2.0
324: (8)                    closed = _is_closed_polygon(slc)
325: (8)                    if closed:
326: (12)                       slc = np.concatenate([slc[ind:-1], slc[:ind + 1]])
```

```
327: (12)                        if len(lc):  # Rotate lc also if not empty
328: (16)                            lc = np.concatenate([lc[ind:-1], lc[:ind + 1]])
329: (12)                        ind = 0
330: (8)                    pl = np.zeros(slc.shape[0], dtype=float)
331: (8)                    dx = np.diff(slc, axis=0)
332: (8)                    pl[1:] = np.cumsum(np.hypot(dx[:, 0], dx[:, 1]))
333: (8)                    pl = pl - pl[ind]
334: (8)                    xi = np.array([-hlw, hlw])
335: (8)                    if closed:  # Look at end also for closed contours
336: (12)                        dp = np.array([pl[-1], 0])
337: (8)                    else:
338: (12)                        dp = np.zeros_like(xi)
339: (8)                    (dx,), (dy,) = (np.diff(np.interp(dp + xi, pl, slc_col))
340: (24)                                    for slc_col in slc.T)
341: (8)                    rotation = np.rad2deg(np.arctan2(dy, dx))
342: (8)                    if self.rightside_up:
343: (12)                        rotation = (rotation + 90) % 180 - 90
344: (8)                    nlc = []
345: (8)                    if len(lc):
346: (12)                        xi = dp + xi + np.array([-spacing, spacing])
347: (12)                        I = np.interp(xi, pl, np.arange(len(pl)), left=-1, right=-1)
348: (12)                        I = [np.floor(I[0]).astype(int), np.ceil(I[1]).astype(int)]
349: (12)                        if I[0] != -1:
350: (16)                            xy1 = [np.interp(xi[0], pl, lc_col) for lc_col in lc.T]
351: (12)                        if I[1] != -1:
352: (16)                            xy2 = [np.interp(xi[1], pl, lc_col) for lc_col in lc.T]
353: (12)                        if closed:
354: (16)                            if all(i != -1 for i in I):
355: (20)                                nlc.append(np.vstack([xy2, lc[I[1]:I[0]+1], xy1]))
356: (12)                        else:
357: (16)                            if I[0] != -1:
358: (20)                                nlc.append(np.vstack([lc[:I[0]+1], xy1]))
359: (16)                            if I[1] != -1:
360: (20)                                nlc.append(np.vstack([xy2, lc[I[1]:]]))
361: (8)                    return rotation, nlc
362: (4)                def add_label(self, x, y, rotation, lev, cvalue):
363: (8)                    """Add a contour label, respecting whether *use_clabeltext* was
set."""
364: (8)                    data_x, data_y = self.axes.transData.inverted().transform((x, y))
365: (8)                    t = Text(
366: (12)                        data_x, data_y,
367: (12)                        text=self.get_text(lev, self.labelFmt),
368: (12)                        rotation=rotation,
369: (12)                        horizontalalignment='center', verticalalignment='center',
370: (12)                        zorder=self._clabel_zorder,
371: (12)                        color=self.labelMappable.to_rgba(cvalue, alpha=self.get_alpha()),
372: (12)                        fontproperties=self._label_font_props,
373: (12)                        clip_box=self.axes.bbox)
374: (8)                    if self._use_clabeltext:
375: (12)                        data_rotation, = self.axes.transData.inverted().transform_angles(
376: (16)                            [rotation], [[x, y]])
377: (12)                        t.set(rotation=data_rotation, transform_rotates_text=True)
378: (8)                    self.labelTexts.append(t)
379: (8)                    self.labelCValues.append(cvalue)
380: (8)                    self.labelXYs.append((x, y))
381: (8)                    self.axes.add_artist(t)
382: (4)                @_api.deprecated("3.8", alternative="add_label")
383: (4)                def add_label_clabeltext(self, x, y, rotation, lev, cvalue):
384: (8)                    """Add contour label with `.Text.set_transform_rotates_text`."""
385: (8)                    with cbook._setattr_cm(self, _use_clabeltext=True):
386: (12)                        self.add_label(x, y, rotation, lev, cvalue)
387: (4)                def add_label_near(self, x, y, inline=True, inline_spacing=5,
388: (23)                                   transform=None):
389: (8)                    """
390: (8)                    Add a label near the point ``(x, y)``.
391: (8)                    Parameters
392: (8)                    ----------
393: (8)                    x, y : float
394: (12)                        The approximate location of the label.
```

```
395: (8)                    inline : bool, default: True
396: (12)                       If *True* remove the segment of the contour beneath the label.
397: (8)                    inline_spacing : int, default: 5
398: (12)                       Space in pixels to leave on each side of label when placing
399: (12)                       inline. This spacing will be exact for labels at locations where
400: (12)                       the contour is straight, less so for labels on curved contours.
401: (8)                    transform : `.Transform` or `False`, default: ``self.axes.transData``
402: (12)                       A transform applied to ``(x, y)`` before labeling.  The default
403: (12)                       causes ``(x, y)`` to be interpreted as data coordinates.  `False`
404: (12)                       is a synonym for `.IdentityTransform`; i.e. ``(x, y)`` should be
405: (12)                       interpreted as display coordinates.
406: (8)                    """
407: (8)                    if transform is None:
408: (12)                       transform = self.axes.transData
409: (8)                    if transform:
410: (12)                       x, y = transform.transform((x, y))
411: (8)                    idx_level_min, idx_vtx_min, proj = self._find_nearest_contour(
412: (12)                       (x, y), self.labelIndiceList)
413: (8)                    path = self._paths[idx_level_min]
414: (8)                    level = self.labelIndiceList.index(idx_level_min)
415: (8)                    label_width = self._get_nth_label_width(level)
416: (8)                    rotation, path = self._split_path_and_get_label_rotation(
417: (12)                       path, idx_vtx_min, proj, label_width, inline_spacing)
418: (8)                    self.add_label(*proj, rotation, self.labelLevelList[idx_level_min],
419: (23)                                  self.labelCValueList[idx_level_min])
420: (8)                    if inline:
421: (12)                       self._paths[idx_level_min] = path
422: (4)                def pop_label(self, index=-1):
423: (8)                    """Defaults to removing last label, but any index can be supplied"""
424: (8)                    self.labelCValues.pop(index)
425: (8)                    t = self.labelTexts.pop(index)
426: (8)                    t.remove()
427: (4)                def labels(self, inline, inline_spacing):
428: (8)                    for idx, (icon, lev, cvalue) in enumerate(zip(
429: (16)                           self.labelIndiceList,
430: (16)                           self.labelLevelList,
431: (16)                           self.labelCValueList,
432: (8)                    )):
433: (12)                       trans = self.get_transform()
434: (12)                       label_width = self._get_nth_label_width(idx)
435: (12)                       additions = []
436: (12)                       for subpath in self._paths[icon]._iter_connected_components():
437: (16)                           screen_xys = trans.transform(subpath.vertices)
438: (16)                           if self.print_label(screen_xys, label_width):
439: (20)                               x, y, idx = self.locate_label(screen_xys, label_width)
440: (20)                               rotation, path = self._split_path_and_get_label_rotation(
441: (24)                                   subpath, idx, (x, y),
442: (24)                                   label_width, inline_spacing)
443: (20)                               self.add_label(x, y, rotation, lev, cvalue)  # Really add
label.
444: (20)                               if inline:  # If inline, add new contours
445: (24)                                   additions.append(path)
446: (16)                           else:  # If not adding label, keep old path
447: (20)                               additions.append(subpath)
448: (12)                       if inline:
449: (16)                           self._paths[icon] = Path.make_compound_path(*additions)
450: (4)                def remove(self):
451: (8)                    super().remove()
452: (8)                    for text in self.labelTexts:
453: (12)                       text.remove()
454: (0)            def _is_closed_polygon(X):
455: (4)                """
456: (4)                Return whether first and last object in a sequence are the same. These are
457: (4)                presumably coordinates on a polygonal curve, in which case this function
458: (4)                tests if that curve is closed.
459: (4)                """
460: (4)                return np.allclose(X[0], X[-1], rtol=1e-10, atol=1e-13)
461: (0)            def _find_closest_point_on_path(xys, p):
462: (4)                """
```

```
(4)                 Parameters
(4)                 ----------
(4)                 xys : (N, 2) array-like
(8)                     Coordinates of vertices.
(4)                 p : (float, float)
(8)                     Coordinates of point.
(4)                 Returns
(4)                 -------
(4)                 d2min : float
(8)                     Minimum square distance of *p* to *xys*.
(4)                 proj : (float, float)
(8)                     Projection of *p* onto *xys*.
(4)                 imin : (int, int)
(8)                     Consecutive indices of vertices of segment in *xys* where *proj* is.
(8)                     Segments are considered as including their end-points; i.e. if the
(8)                     closest point on the path is a node in *xys* with index *i*, this
(8)                     returns ``(i-1, i)``.  For the special case where *xys* is a single
(8)                     point, this returns ``(0, 0)``.
(4)                 """
(4)                 if len(xys) == 1:
(8)                     return (((p - xys[0]) ** 2).sum(), xys[0], (0, 0))
(4)                 dxys = xys[1:] - xys[:-1]  # Individual segment vectors.
(4)                 norms = (dxys ** 2).sum(axis=1)
(4)                 norms[norms == 0] = 1  # For zero-length segment, replace 0/0 by 0/1.
(4)                 rel_projs = np.clip(  # Project onto each segment in relative 0-1 coords.
(8)                     ((p - xys[:-1]) * dxys).sum(axis=1) / norms,
(8)                     0, 1)[:, None]
(4)                 projs = xys[:-1] + rel_projs * dxys  # Projs. onto each segment, in (x,
y).
(4)                 d2s = ((projs - p) ** 2).sum(axis=1)  # Squared distances.
(4)                 imin = np.argmin(d2s)
(4)                 return (d2s[imin], projs[imin], (imin, imin+1))
(0)             _docstring.interpd.update(contour_set_attributes=r"""
(0)             Attributes
(0)             ----------
(0)             ax : `~matplotlib.axes.Axes`
(4)                 The Axes object in which the contours are drawn.
(0)             collections : `.silent_list` of `.PathCollection`\s
(4)                 The `.Artist`\s representing the contour. This is a list of
(4)                 `.PathCollection`\s for both line and filled contours.
(0)             levels : array
(4)                 The values of the contour levels.
(0)             layers : array
(4)                 Same as levels for line contours; half-way between
(4)                 levels for filled contours.  See ``ContourSet._process_colors``.
(0)             """)
(0)             @_docstring.dedent_interpd
(0)             class ContourSet(ContourLabeler, mcoll.Collection):
(4)                 """
(4)                 Store a set of contour lines or filled regions.
(4)                 User-callable method: `~.Axes.clabel`
(4)                 Parameters
(4)                 ----------
(4)                 ax : `~matplotlib.axes.Axes`
(4)                 levels : [level0, level1, ..., leveln]
(8)                     A list of floating point numbers indicating the contour levels.
(4)                 allsegs : [level0segs, level1segs, ...]
(8)                     List of all the polygon segments for all the *levels*.
(8)                     For contour lines ``len(allsegs) == len(levels)``, and for
(8)                     filled contour regions ``len(allsegs) = len(levels)-1``. The lists
(8)                     should look like ::
(12)                        level0segs = [polygon0, polygon1, ...]
(12)                        polygon0 = [[x0, y0], [x1, y1], ...]
(4)                 allkinds : ``None`` or [level0kinds, level1kinds, ...]
(8)                     Optional list of all the polygon vertex kinds (code types), as
(8)                     described and used in Path. This is used to allow multiply-
(8)                     connected paths such as holes within filled polygons.
(8)                     If not ``None``, ``len(allkinds) == len(allsegs)``. The lists
(8)                     should look like ::
```

```
(12)                        level0kinds = [polygon0kinds, ...]
(12)                        polygon0kinds = [vertexcode0, vertexcode1, ...]
(8)                     If *allkinds* is not ``None``, usually all polygons for a
(8)                     particular contour level are grouped together so that
(8)                     ``level0segs = [polygon0]`` and ``level0kinds = [polygon0kinds]``.
(4)                 **kwargs
(8)                     Keyword arguments are as described in the docstring of
(8)                     `~.Axes.contour`.
(4)                 %(contour_set_attributes)s
(4)                 """
(4)                 def __init__(self, ax, *args,
(17)                             levels=None, filled=False, linewidths=None, linestyles=None,
(17)                             hatches=(None,), alpha=None, origin=None, extent=None,
(17)                             cmap=None, colors=None, norm=None, vmin=None, vmax=None,
(17)                             extend='neither', antialiased=None, nchunk=0, locator=None,
(17)                             transform=None, negative_linestyles=None, clip_path=None,
(17)                             **kwargs):
(8)                     """
(8)                     Draw contour lines or filled regions, depending on
(8)                     whether keyword arg *filled* is ``False`` (default) or ``True``.
(8)                     Call signature::
(12)                        ContourSet(ax, levels, allsegs, [allkinds], **kwargs)
(8)                     Parameters
(8)                     ----------
(8)                     ax : `~matplotlib.axes.Axes`
(12)                        The `~.axes.Axes` object to draw on.
(8)                     levels : [level0, level1, ..., leveln]
(12)                        A list of floating point numbers indicating the contour
(12)                        levels.
(8)                     allsegs : [level0segs, level1segs, ...]
(12)                        List of all the polygon segments for all the *levels*.
(12)                        For contour lines ``len(allsegs) == len(levels)``, and for
(12)                        filled contour regions ``len(allsegs) = len(levels)-1``. The lists
(12)                        should look like ::
(16)                            level0segs = [polygon0, polygon1, ...]
(16)                            polygon0 = [[x0, y0], [x1, y1], ...]
(8)                     allkinds : [level0kinds, level1kinds, ...], optional
(12)                        Optional list of all the polygon vertex kinds (code types), as
(12)                        described and used in Path. This is used to allow multiply-
(12)                        connected paths such as holes within filled polygons.
(12)                        If not ``None``, ``len(allkinds) == len(allsegs)``. The lists
(12)                        should look like ::
(16)                            level0kinds = [polygon0kinds, ...]
(16)                            polygon0kinds = [vertexcode0, vertexcode1, ...]
(12)                        If *allkinds* is not ``None``, usually all polygons for a
(12)                        particular contour level are grouped together so that
(12)                        ``level0segs = [polygon0]`` and ``level0kinds = [polygon0kinds]``.
(8)                     **kwargs
(12)                        Keyword arguments are as described in the docstring of
(12)                        `~.Axes.contour`.
(8)                     """
(8)                     if antialiased is None and filled:
(12)                        antialiased = False
(8)                     super().__init__(
(12)                        antialiaseds=antialiased,
(12)                        alpha=alpha,
(12)                        clip_path=clip_path,
(12)                        transform=transform,
(8)                     )
(8)                     self.axes = ax
(8)                     self.levels = levels
(8)                     self.filled = filled
(8)                     self.hatches = hatches
(8)                     self.origin = origin
(8)                     self.extent = extent
(8)                     self.colors = colors
(8)                     self.extend = extend
(8)                     self.nchunk = nchunk
(8)                     self.locator = locator
```

```
(8)                     if (isinstance(norm, mcolors.LogNorm)
(16)                            or isinstance(self.locator, ticker.LogLocator)):
(12)                        self.logscale = True
(12)                        if norm is None:
(16)                            norm = mcolors.LogNorm()
(8)                     else:
(12)                        self.logscale = False
(8)                     _api.check_in_list([None, 'lower', 'upper', 'image'], origin=origin)
(8)                     if self.extent is not None and len(self.extent) != 4:
(12)                        raise ValueError(
(16)                            "If given, 'extent' must be None or (x0, x1, y0, y1)")
(8)                     if self.colors is not None and cmap is not None:
(12)                        raise ValueError('Either colors or cmap must be None')
(8)                     if self.origin == 'image':
(12)                        self.origin = mpl.rcParams['image.origin']
(8)                     self._orig_linestyles = linestyles  # Only kept for user access.
(8)                     self.negative_linestyles = negative_linestyles
(8)                     if self.negative_linestyles is None:
(12)                        self.negative_linestyles = \
(16)                            mpl.rcParams['contour.negative_linestyle']
(8)                     kwargs = self._process_args(*args, **kwargs)
(8)                     self._process_levels()
(8)                     self._extend_min = self.extend in ['min', 'both']
(8)                     self._extend_max = self.extend in ['max', 'both']
(8)                     if self.colors is not None:
(12)                        ncolors = len(self.levels)
(12)                        if self.filled:
(16)                            ncolors -= 1
(12)                        i0 = 0
(12)                        use_set_under_over = False
(12)                        total_levels = (ncolors +
(28)                                        int(self._extend_min) +
(28)                                        int(self._extend_max))
(12)                        if (len(self.colors) == total_levels and
(20)                                (self._extend_min or self._extend_max)):
(16)                            use_set_under_over = True
(16)                            if self._extend_min:
(20)                                i0 = 1
(12)                        cmap = mcolors.ListedColormap(self.colors[i0:None], N=ncolors)
(12)                        if use_set_under_over:
(16)                            if self._extend_min:
(20)                                cmap.set_under(self.colors[0])
(16)                            if self._extend_max:
(20)                                cmap.set_over(self.colors[-1])
(8)                     self.labelTexts = []
(8)                     self.labelCValues = []
(8)                     self.set_cmap(cmap)
(8)                     if norm is not None:
(12)                        self.set_norm(norm)
(8)                     with self.norm.callbacks.blocked(signal="changed"):
(12)                        if vmin is not None:
(16)                            self.norm.vmin = vmin
(12)                        if vmax is not None:
(16)                            self.norm.vmax = vmax
(8)                     self.norm._changed()
(8)                     self._process_colors()
(8)                     if self._paths is None:
(12)                        self._paths = self._make_paths_from_contour_generator()
(8)                     if self.filled:
(12)                        if linewidths is not None:
(16)                            _api.warn_external('linewidths is ignored by contourf')
(12)                        lowers, uppers = self._get_lowers_and_uppers()
(12)                        self.set(
(16)                            edgecolor="none",
(16)                            zorder=kwargs.pop("zorder", 1),
(12)                        )
(8)                     else:
(12)                        self.set(
(16)                            facecolor="none",
```

```
669: (16)                            linewidths=self._process_linewidths(linewidths),
670: (16)                            linestyle=self._process_linestyles(linestyles),
671: (16)                            zorder=kwargs.pop("zorder", 2),
672: (16)                            label="_nolegend_",
673: (12)                        )
674: (8)                    self.axes.add_collection(self, autolim=False)
675: (8)                    self.sticky_edges.x[:] = [self._mins[0], self._maxs[0]]
676: (8)                    self.sticky_edges.y[:] = [self._mins[1], self._maxs[1]]
677: (8)                    self.axes.update_datalim([self._mins, self._maxs])
678: (8)                    self.axes.autoscale_view(tight=True)
679: (8)                    self.changed()  # set the colors
680: (8)                    if kwargs:
681: (12)                       _api.warn_external(
682: (16)                           'The following kwargs were not used by contour: ' +
683: (16)                           ", ".join(map(repr, kwargs))
684: (12)                       )
685: (4)                allsegs = property(lambda self: [
686: (8)                    [subp.vertices for subp in p._iter_connected_components()]
687: (8)                    for p in self.get_paths()])
688: (4)                allkinds = property(lambda self: [
689: (8)                    [subp.codes for subp in p._iter_connected_components()]
690: (8)                    for p in self.get_paths()])
691: (4)                tcolors = _api.deprecated("3.8")(property(lambda self: [
692: (8)                    (tuple(rgba),) for rgba in self.to_rgba(self.cvalues, self.alpha)]))
693: (4)                tlinewidths = _api.deprecated("3.8")(property(lambda self: [
694: (8)                    (w,) for w in self.get_linewidths()]))
695: (4)                alpha = property(lambda self: self.get_alpha())
696: (4)                linestyles = property(lambda self: self._orig_linestyles)
697: (4)                @_api.deprecated("3.8", alternative="set_antialiased or get_antialiased",
698: (21)                                 addendum="Note that get_antialiased returns an array.")
699: (4)                @property
700: (4)                def antialiased(self):
701: (8)                    return all(self.get_antialiased())
702: (4)                @antialiased.setter
703: (4)                def antialiased(self, aa):
704: (8)                    self.set_antialiased(aa)
705: (4)                @_api.deprecated("3.8")
706: (4)                @property
707: (4)                def collections(self):
708: (8)                    if not hasattr(self, "_old_style_split_collections"):
709: (12)                       self.set_visible(False)
710: (12)                       fcs = self.get_facecolor()
711: (12)                       ecs = self.get_edgecolor()
712: (12)                       lws = self.get_linewidth()
713: (12)                       lss = self.get_linestyle()
714: (12)                       self._old_style_split_collections = []
715: (12)                       for idx, path in enumerate(self._paths):
716: (16)                           pc = mcoll.PathCollection(
717: (20)                               [path] if len(path.vertices) else [],
718: (20)                               alpha=self.get_alpha(),
719: (20)                               antialiaseds=self._antialiaseds[idx %
len(self._antialiaseds)],
720: (20)                               transform=self.get_transform(),
721: (20)                               zorder=self.get_zorder(),
722: (20)                               label="_nolegend_",
723: (20)                               facecolor=fcs[idx] if len(fcs) else "none",
724: (20)                               edgecolor=ecs[idx] if len(ecs) else "none",
725: (20)                               linewidths=[lws[idx % len(lws)]],
726: (20)                               linestyles=[lss[idx % len(lss)]],
727: (16)                           )
728: (16)                           if self.filled:
729: (20)                               pc.set(hatch=self.hatches[idx % len(self.hatches)])
730: (16)                           self._old_style_split_collections.append(pc)
731: (12)                       for col in self._old_style_split_collections:
732: (16)                           self.axes.add_collection(col)
733: (8)                    return self._old_style_split_collections
734: (4)                def get_transform(self):
735: (8)                    """Return the `.Transform` instance used by this ContourSet."""
736: (8)                    if self._transform is None:
```

```
(12)                        self._transform = self.axes.transData
(8)                     elif (not isinstance(self._transform, mtransforms.Transform)
(14)                          and hasattr(self._transform, '_as_mpl_transform')):
(12)                        self._transform = self._transform._as_mpl_transform(self.axes)
(8)                     return self._transform
(4)                 def __getstate__(self):
(8)                     state = self.__dict__.copy()
(8)                     state['_contour_generator'] = None
(8)                     return state
(4)                 def legend_elements(self, variable_name='x', str_format=str):
(8)                     """
(8)                     Return a list of artists and labels suitable for passing through
(8)                     to `~.Axes.legend` which represent this ContourSet.
(8)                     The labels have the form "0 < x <= 1" stating the data ranges which
(8)                     the artists represent.
(8)                     Parameters
(8)                     ----------
(8)                     variable_name : str
(12)                        The string used inside the inequality used on the labels.
(8)                     str_format : function: float -> str
(12)                        Function used to format the numbers in the labels.
(8)                     Returns
(8)                     -------
(8)                     artists : list[`.Artist`]
(12)                        A list of the artists.
(8)                     labels : list[str]
(12)                        A list of the labels.
(8)                     """
(8)                     artists = []
(8)                     labels = []
(8)                     if self.filled:
(12)                        lowers, uppers = self._get_lowers_and_uppers()
(12)                        n_levels = len(self._paths)
(12)                        for idx in range(n_levels):
(16)                            artists.append(mpatches.Rectangle(
(20)                                (0, 0), 1, 1,
(20)                                facecolor=self.get_facecolor()[idx],
(20)                                hatch=self.hatches[idx % len(self.hatches)],
(16)                            ))
(16)                            lower = str_format(lowers[idx])
(16)                            upper = str_format(uppers[idx])
(16)                            if idx == 0 and self.extend in ('min', 'both'):
(20)                                labels.append(fr'${variable_name} \leq {lower}s$')
(16)                            elif idx == n_levels - 1 and self.extend in ('max', 'both'):
(20)                                labels.append(fr'${variable_name} > {upper}s$')
(16)                            else:
(20)                                labels.append(fr'${lower} < {variable_name} \leq
{upper}$')
(8)                     else:
(12)                        for idx, level in enumerate(self.levels):
(16)                            artists.append(Line2D(
(20)                                [], [],
(20)                                color=self.get_edgecolor()[idx],
(20)                                linewidth=self.get_linewidths()[idx],
(20)                                linestyle=self.get_linestyles()[idx],
(16)                            ))
(16)                            labels.append(fr'${variable_name} = {str_format(level)}$')
(8)                     return artists, labels
(4)                 def _process_args(self, *args, **kwargs):
(8)                     """
(8)                     Process *args* and *kwargs*; override in derived classes.
(8)                     Must set self.levels, self.zmin and self.zmax, and update Axes limits.
(8)                     """
(8)                     self.levels = args[0]
(8)                     allsegs = args[1]
(8)                     allkinds = args[2] if len(args) > 2 else None
(8)                     self.zmax = np.max(self.levels)
(8)                     self.zmin = np.min(self.levels)
(8)                     if allkinds is None:
```

```
805: (12)                        allkinds = [[None] * len(segs) for segs in allsegs]
806: (8)                     if self.filled:
807: (12)                        if len(allsegs) != len(self.levels) - 1:
808: (16)                            raise ValueError('must be one less number of segments as '
809: (33)                                             'levels')
810: (8)                     else:
811: (12)                        if len(allsegs) != len(self.levels):
812: (16)                            raise ValueError('must be same number of segments as levels')
813: (8)                     if len(allkinds) != len(allsegs):
814: (12)                        raise ValueError('allkinds has different length to allsegs')
815: (8)                     flatseglist = [s for seg in allsegs for s in seg]
816: (8)                     points = np.concatenate(flatseglist, axis=0)
817: (8)                     self._mins = points.min(axis=0)
818: (8)                     self._maxs = points.max(axis=0)
819: (8)                     self._paths = [Path.make_compound_path(*map(Path, segs, kinds))
820: (23)                                   for segs, kinds in zip(allsegs, allkinds)]
821: (8)                     return kwargs
822: (4)                 def _make_paths_from_contour_generator(self):
823: (8)                     """Compute ``paths`` using C extension."""
824: (8)                     if self._paths is not None:
825: (12)                        return self._paths
826: (8)                     cg = self._contour_generator
827: (8)                     empty_path = Path(np.empty((0, 2)))
828: (8)                     vertices_and_codes = (
829: (12)                        map(cg.create_filled_contour, *self._get_lowers_and_uppers())
830: (12)                        if self.filled else
831: (12)                        map(cg.create_contour, self.levels))
832: (8)                     return [Path(np.concatenate(vs), np.concatenate(cs)) if len(vs) else
empty_path
833: (16)                            for vs, cs in vertices_and_codes]
834: (4)                 def _get_lowers_and_uppers(self):
835: (8)                     """
836: (8)                     Return ``(lowers, uppers)`` for filled contours.
837: (8)                     """
838: (8)                     lowers = self._levels[:-1]
839: (8)                     if self.zmin == lowers[0]:
840: (12)                        lowers = lowers.copy()  # so we don't change self._levels
841: (12)                        if self.logscale:
842: (16)                            lowers[0] = 0.99 * self.zmin
843: (12)                        else:
844: (16)                            lowers[0] -= 1
845: (8)                     uppers = self._levels[1:]
846: (8)                     return (lowers, uppers)
847: (4)                 def changed(self):
848: (8)                     if not hasattr(self, "cvalues"):
849: (12)                        self._process_colors()  # Sets cvalues.
850: (8)                     self.norm.autoscale_None(self.levels)
851: (8)                     self.set_array(self.cvalues)
852: (8)                     self.update_scalarmappable()
853: (8)                     alphas = np.broadcast_to(self.get_alpha(), len(self.cvalues))
854: (8)                     for label, cv, alpha in zip(self.labelTexts, self.labelCValues,
alphas):
855: (12)                        label.set_alpha(alpha)
856: (12)                        label.set_color(self.labelMappable.to_rgba(cv))
857: (8)                     super().changed()
858: (4)                 def _autolev(self, N):
859: (8)                     """
860: (8)                     Select contour levels to span the data.
861: (8)                     The target number of levels, *N*, is used only when the
862: (8)                     scale is not log and default locator is used.
863: (8)                     We need two more levels for filled contours than for
864: (8)                     line contours, because for the latter we need to specify
865: (8)                     the lower and upper boundary of each range. For example,
866: (8)                     a single contour boundary, say at z = 0, requires only
867: (8)                     one contour line, but two filled regions, and therefore
868: (8)                     three levels to provide boundaries for both regions.
869: (8)                     """
870: (8)                     if self.locator is None:
871: (12)                        if self.logscale:
```

```
872: (16)                            self.locator = ticker.LogLocator()
873: (12)                        else:
874: (16)                            self.locator = ticker.MaxNLocator(N + 1, min_n_ticks=1)
875: (8)                    lev = self.locator.tick_values(self.zmin, self.zmax)
876: (8)                    try:
877: (12)                        if self.locator._symmetric:
878: (16)                            return lev
879: (8)                    except AttributeError:
880: (12)                        pass
881: (8)                    under = np.nonzero(lev < self.zmin)[0]
882: (8)                    i0 = under[-1] if len(under) else 0
883: (8)                    over = np.nonzero(lev > self.zmax)[0]
884: (8)                    i1 = over[0] + 1 if len(over) else len(lev)
885: (8)                    if self.extend in ('min', 'both'):
886: (12)                        i0 += 1
887: (8)                    if self.extend in ('max', 'both'):
888: (12)                        i1 -= 1
889: (8)                    if i1 - i0 < 3:
890: (12)                        i0, i1 = 0, len(lev)
891: (8)                    return lev[i0:i1]
892: (4)                def _process_contour_level_args(self, args, z_dtype):
893: (8)                    """
894: (8)                    Determine the contour levels and store in self.levels.
895: (8)                    """
896: (8)                    if self.levels is None:
897: (12)                        if args:
898: (16)                            levels_arg = args[0]
899: (12)                        elif np.issubdtype(z_dtype, bool):
900: (16)                            if self.filled:
901: (20)                                levels_arg = [0, .5, 1]
902: (16)                            else:
903: (20)                                levels_arg = [.5]
904: (12)                        else:
905: (16)                            levels_arg = 7  # Default, hard-wired.
906: (8)                    else:
907: (12)                        levels_arg = self.levels
908: (8)                    if isinstance(levels_arg, Integral):
909: (12)                        self.levels = self._autolev(levels_arg)
910: (8)                    else:
911: (12)                        self.levels = np.asarray(levels_arg, np.float64)
912: (8)                    if self.filled and len(self.levels) < 2:
913: (12)                        raise ValueError("Filled contours require at least 2 levels.")
914: (8)                    if len(self.levels) > 1 and np.min(np.diff(self.levels)) <= 0.0:
915: (12)                        raise ValueError("Contour levels must be increasing")
916: (4)                def _process_levels(self):
917: (8)                    """
918: (8)                    Assign values to :attr:`layers` based on :attr:`levels`,
919: (8)                    adding extended layers as needed if contours are filled.
920: (8)                    For line contours, layers simply coincide with levels;
921: (8)                    a line is a thin layer.  No extended levels are needed
922: (8)                    with line contours.
923: (8)                    """
924: (8)                    self._levels = list(self.levels)
925: (8)                    if self.logscale:
926: (12)                        lower, upper = 1e-250, 1e250
927: (8)                    else:
928: (12)                        lower, upper = -1e250, 1e250
929: (8)                    if self.extend in ('both', 'min'):
930: (12)                        self._levels.insert(0, lower)
931: (8)                    if self.extend in ('both', 'max'):
932: (12)                        self._levels.append(upper)
933: (8)                    self._levels = np.asarray(self._levels)
934: (8)                    if not self.filled:
935: (12)                        self.layers = self.levels
936: (12)                        return
937: (8)                    if self.logscale:
938: (12)                        self.layers = (np.sqrt(self._levels[:-1])
939: (27)                                       * np.sqrt(self._levels[1:]))
940: (8)                    else:
```

```
(12)                        self.layers = 0.5 * (self._levels[:-1] + self._levels[1:])
(4)                 def _process_colors(self):
(8)                     """
(8)                     Color argument processing for contouring.
(8)                     Note that we base the colormapping on the contour levels
(8)                     and layers, not on the actual range of the Z values.  This
(8)                     means we don't have to worry about bad values in Z, and we
(8)                     always have the full dynamic range available for the selected
(8)                     levels.
(8)                     The color is based on the midpoint of the layer, except for
(8)                     extended end layers.  By default, the norm vmin and vmax
(8)                     are the extreme values of the non-extended levels.  Hence,
(8)                     the layer color extremes are not the extreme values of
(8)                     the colormap itself, but approach those values as the number
(8)                     of levels increases.  An advantage of this scheme is that
(8)                     line contours, when added to filled contours, take on
(8)                     colors that are consistent with those of the filled regions;
(8)                     for example, a contour line on the boundary between two
(8)                     regions will have a color intermediate between those
(8)                     of the regions.
(8)                     """
(8)                     self.monochrome = self.cmap.monochrome
(8)                     if self.colors is not None:
(12)                        i0, i1 = 0, len(self.levels)
(12)                        if self.filled:
(16)                            i1 -= 1
(16)                            if self.extend in ('both', 'min'):
(20)                                i0 -= 1
(16)                            if self.extend in ('both', 'max'):
(20)                                i1 += 1
(12)                        self.cvalues = list(range(i0, i1))
(12)                        self.set_norm(mcolors.NoNorm())
(8)                     else:
(12)                        self.cvalues = self.layers
(8)                     self.norm.autoscale_None(self.levels)
(8)                     self.set_array(self.cvalues)
(8)                     self.update_scalarmappable()
(8)                     if self.extend in ('both', 'max', 'min'):
(12)                        self.norm.clip = False
(4)                 def _process_linewidths(self, linewidths):
(8)                     Nlev = len(self.levels)
(8)                     if linewidths is None:
(12)                        default_linewidth = mpl.rcParams['contour.linewidth']
(12)                        if default_linewidth is None:
(16)                            default_linewidth = mpl.rcParams['lines.linewidth']
(12)                        return [default_linewidth] * Nlev
(8)                     elif not np.iterable(linewidths):
(12)                        return [linewidths] * Nlev
(8)                     else:
(12)                        linewidths = list(linewidths)
(12)                        return (linewidths * math.ceil(Nlev / len(linewidths)))[:Nlev]
(4)                 def _process_linestyles(self, linestyles):
(8)                     Nlev = len(self.levels)
(8)                     if linestyles is None:
(12)                        tlinestyles = ['solid'] * Nlev
(12)                        if self.monochrome:
(16)                            eps = - (self.zmax - self.zmin) * 1e-15
(16)                            for i, lev in enumerate(self.levels):
(20)                                if lev < eps:
(24)                                    tlinestyles[i] = self.negative_linestyles
(8)                     else:
(12)                        if isinstance(linestyles, str):
(16)                            tlinestyles = [linestyles] * Nlev
(12)                        elif np.iterable(linestyles):
(16)                            tlinestyles = list(linestyles)
(16)                            if len(tlinestyles) < Nlev:
(20)                                nreps = int(np.ceil(Nlev / len(linestyles)))
(20)                                tlinestyles = tlinestyles * nreps
(16)                            if len(tlinestyles) > Nlev:
```

```
1010: (20)                                tlinestyles = tlinestyles[:Nlev]
1011: (12)                        else:
1012: (16)                            raise ValueError("Unrecognized type for linestyles kwarg")
1013: (8)                    return tlinestyles
1014: (4)                def _find_nearest_contour(self, xy, indices=None):
1015: (8)                    """
1016: (8)                    Find the point in the unfilled contour plot that is closest (in screen
1017: (8)                    space) to point *xy*.
1018: (8)                    Parameters
1019: (8)                    ----------
1020: (8)                    xy : tuple[float, float]
1021: (12)                        The reference point (in screen space).
1022: (8)                    indices : list of int or None, default: None
1023: (12)                        Indices of contour levels to consider.  If None (the default), all
levels
1024: (12)                        are considered.
1025: (8)                    Returns
1026: (8)                    -------
1027: (8)                    idx_level_min : int
1028: (12)                        The index of the contour level closest to *xy*.
1029: (8)                    idx_vtx_min : int
1030: (12)                        The index of the `.Path` segment closest to *xy* (at that level).
1031: (8)                    proj : (float, float)
1032: (12)                        The point in the contour plot closest to *xy*.
1033: (8)                    """
1034: (8)                    if self.filled:
1035: (12)                        raise ValueError("Method does not support filled contours")
1036: (8)                    if indices is None:
1037: (12)                        indices = range(len(self._paths))
1038: (8)                    d2min = np.inf
1039: (8)                    idx_level_min = idx_vtx_min = proj_min = None
1040: (8)                    for idx_level in indices:
1041: (12)                        path = self._paths[idx_level]
1042: (12)                        idx_vtx_start = 0
1043: (12)                        for subpath in path._iter_connected_components():
1044: (16)                            if not len(subpath.vertices):
1045: (20)                                continue
1046: (16)                            lc = self.get_transform().transform(subpath.vertices)
1047: (16)                            d2, proj, leg = _find_closest_point_on_path(lc, xy)
1048: (16)                            if d2 < d2min:
1049: (20)                                d2min = d2
1050: (20)                                idx_level_min = idx_level
1051: (20)                                idx_vtx_min = leg[1] + idx_vtx_start
1052: (20)                                proj_min = proj
1053: (16)                            idx_vtx_start += len(subpath)
1054: (8)                    return idx_level_min, idx_vtx_min, proj_min
1055: (4)                def find_nearest_contour(self, x, y, indices=None, pixel=True):
1056: (8)                    """
1057: (8)                    Find the point in the contour plot that is closest to ``(x, y)``.
1058: (8)                    This method does not support filled contours.
1059: (8)                    Parameters
1060: (8)                    ----------
1061: (8)                    x, y : float
1062: (12)                        The reference point.
1063: (8)                    indices : list of int or None, default: None
1064: (12)                        Indices of contour levels to consider.  If None (the default), all
1065: (12)                        levels are considered.
1066: (8)                    pixel : bool, default: True
1067: (12)                        If *True*, measure distance in pixel (screen) space, which is
1068: (12)                        useful for manual contour labeling; else, measure distance in axes
1069: (12)                        space.
1070: (8)                    Returns
1071: (8)                    -------
1072: (8)                    path : int
1073: (12)                        The index of the path that is closest to ``(x, y)``.  Each path
corresponds
1074: (12)                        to one contour level.
1075: (8)                    subpath : int
1076: (12)                        The index within that closest path of the subpath that is closest
```

```
to
1077: (12)                      ``(x, y)``.  Each subpath corresponds to one unbroken contour
line.
1078: (8)                   index : int
1079: (12)                      The index of the vertices within that subpath that are closest to
1080: (12)                      ``(x, y)``.
1081: (8)                   xmin, ymin : float
1082: (12)                      The point in the contour plot that is closest to ``(x, y)``.
1083: (8)                   d2 : float
1084: (12)                      The squared distance from ``(xmin, ymin)`` to ``(x, y)``.
1085: (8)                   """
1086: (8)                   segment = index = d2 = None
1087: (8)                   with ExitStack() as stack:
1088: (12)                      if not pixel:
1089: (16)                          stack.enter_context(self._cm_set(
1090: (20)                              transform=mtransforms.IdentityTransform()))
1091: (12)                      i_level, i_vtx, (xmin, ymin) = self._find_nearest_contour((x, y),
indices)
1092: (8)                   if i_level is not None:
1093: (12)                      cc_cumlens = np.cumsum(
1094: (16)                          [*map(len,
self._paths[i_level]._iter_connected_components())])
1095: (12)                      segment = cc_cumlens.searchsorted(i_vtx, "right")
1096: (12)                      index = i_vtx if segment == 0 else i_vtx - cc_cumlens[segment - 1]
1097: (12)                      d2 = (xmin-x)**2 + (ymin-y)**2
1098: (8)                   return (i_level, segment, index, xmin, ymin, d2)
1099: (4)               def draw(self, renderer):
1100: (8)                   paths = self._paths
1101: (8)                   n_paths = len(paths)
1102: (8)                   if not self.filled or all(hatch is None for hatch in self.hatches):
1103: (12)                      super().draw(renderer)
1104: (12)                      return
1105: (8)                   for idx in range(n_paths):
1106: (12)                      with cbook._setattr_cm(self, _paths=[paths[idx]]), self._cm_set(
1107: (16)                          hatch=self.hatches[idx % len(self.hatches)],
1108: (16)                          array=[self.get_array()[idx]],
1109: (16)                          linewidths=[self.get_linewidths()[idx %
len(self.get_linewidths())]],
1110: (16)                          linestyles=[self.get_linestyles()[idx %
len(self.get_linestyles())]],
1111: (12)                      ):
1112: (16)                          super().draw(renderer)
1113: (0)           @_docstring.dedent_interpd
1114: (0)           class QuadContourSet(ContourSet):
1115: (4)               """
1116: (4)               Create and store a set of contour lines or filled regions.
1117: (4)               This class is typically not instantiated directly by the user but by
1118: (4)               `~.Axes.contour` and `~.Axes.contourf`.
1119: (4)               %(contour_set_attributes)s
1120: (4)               """
1121: (4)               def _process_args(self, *args, corner_mask=None, algorithm=None,
**kwargs):
1122: (8)                   """
1123: (8)                   Process args and kwargs.
1124: (8)                   """
1125: (8)                   if args and isinstance(args[0], QuadContourSet):
1126: (12)                      if self.levels is None:
1127: (16)                          self.levels = args[0].levels
1128: (12)                      self.zmin = args[0].zmin
1129: (12)                      self.zmax = args[0].zmax
1130: (12)                      self._corner_mask = args[0]._corner_mask
1131: (12)                      contour_generator = args[0]._contour_generator
1132: (12)                      self._mins = args[0]._mins
1133: (12)                      self._maxs = args[0]._maxs
1134: (12)                      self._algorithm = args[0]._algorithm
1135: (8)                   else:
1136: (12)                      import contourpy
1137: (12)                      if algorithm is None:
1138: (16)                          algorithm = mpl.rcParams['contour.algorithm']
```

```
1139: (12)                        mpl.rcParams.validate["contour.algorithm"](algorithm)
1140: (12)                        self._algorithm = algorithm
1141: (12)                        if corner_mask is None:
1142: (16)                            if self._algorithm == "mpl2005":
1143: (20)                                corner_mask = False
1144: (16)                            else:
1145: (20)                                corner_mask = mpl.rcParams['contour.corner_mask']
1146: (12)                        self._corner_mask = corner_mask
1147: (12)                        x, y, z = self._contour_args(args, kwargs)
1148: (12)                        contour_generator = contourpy.contour_generator(
1149: (16)                            x, y, z, name=self._algorithm, corner_mask=self._corner_mask,
1150: (16)                            line_type=contourpy.LineType.SeparateCode,
1151: (16)                            fill_type=contourpy.FillType.OuterCode,
1152: (16)                            chunk_size=self.nchunk)
1153: (12)                        t = self.get_transform()
1154: (12)                        if (t != self.axes.transData and
1155: (20)                                any(t.contains_branch_seperately(self.axes.transData))):
1156: (16)                            trans_to_data = t - self.axes.transData
1157: (16)                            pts = np.vstack([x.flat, y.flat]).T
1158: (16)                            transformed_pts = trans_to_data.transform(pts)
1159: (16)                            x = transformed_pts[..., 0]
1160: (16)                            y = transformed_pts[..., 1]
1161: (12)                        self._mins = [ma.min(x), ma.min(y)]
1162: (12)                        self._maxs = [ma.max(x), ma.max(y)]
1163: (8)                    self._contour_generator = contour_generator
1164: (8)                    return kwargs
1165: (4)                def _contour_args(self, args, kwargs):
1166: (8)                    if self.filled:
1167: (12)                        fn = 'contourf'
1168: (8)                    else:
1169: (12)                        fn = 'contour'
1170: (8)                    nargs = len(args)
1171: (8)                    if 0 < nargs <= 2:
1172: (12)                        z, *args = args
1173: (12)                        z = ma.asarray(z)
1174: (12)                        x, y = self._initialize_x_y(z)
1175: (8)                    elif 2 < nargs <= 4:
1176: (12)                        x, y, z_orig, *args = args
1177: (12)                        x, y, z = self._check_xyz(x, y, z_orig, kwargs)
1178: (8)                    else:
1179: (12)                        raise _api.nargs_error(fn, takes="from 1 to 4", given=nargs)
1180: (8)                    z = ma.masked_invalid(z, copy=False)
1181: (8)                    self.zmax = z.max().astype(float)
1182: (8)                    self.zmin = z.min().astype(float)
1183: (8)                    if self.logscale and self.zmin <= 0:
1184: (12)                        z = ma.masked_where(z <= 0, z)
1185: (12)                        _api.warn_external('Log scale: values of z <= 0 have been masked')
1186: (12)                        self.zmin = z.min().astype(float)
1187: (8)                    self._process_contour_level_args(args, z.dtype)
1188: (8)                    return (x, y, z)
1189: (4)                def _check_xyz(self, x, y, z, kwargs):
1190: (8)                    """
1191: (8)                    Check that the shapes of the input arrays match; if x and y are 1D,
1192: (8)                    convert them to 2D using meshgrid.
1193: (8)                    """
1194: (8)                    x, y = self.axes._process_unit_info([("x", x), ("y", y)], kwargs)
1195: (8)                    x = np.asarray(x, dtype=np.float64)
1196: (8)                    y = np.asarray(y, dtype=np.float64)
1197: (8)                    z = ma.asarray(z)
1198: (8)                    if z.ndim != 2:
1199: (12)                        raise TypeError(f"Input z must be 2D, not {z.ndim}D")
1200: (8)                    if z.shape[0] < 2 or z.shape[1] < 2:
1201: (12)                        raise TypeError(f"Input z must be at least a (2, 2) shaped array,
"
1202: (28)                                        f"but has shape {z.shape}")
1203: (8)                    Ny, Nx = z.shape
1204: (8)                    if x.ndim != y.ndim:
1205: (12)                        raise TypeError(f"Number of dimensions of x ({x.ndim}) and y "
1206: (28)                                        f"({y.ndim}) do not match")
```

```
(8)                 if x.ndim == 1:
(12)                    nx, = x.shape
(12)                    ny, = y.shape
(12)                    if nx != Nx:
(16)                        raise TypeError(f"Length of x ({nx}) must match number of "
(32)                                        f"columns in z ({Nx})")
(12)                    if ny != Ny:
(16)                        raise TypeError(f"Length of y ({ny}) must match number of "
(32)                                        f"rows in z ({Ny})")
(12)                    x, y = np.meshgrid(x, y)
(8)                 elif x.ndim == 2:
(12)                    if x.shape != z.shape:
(16)                        raise TypeError(
(20)                            f"Shapes of x {x.shape} and z {z.shape} do not match")
(12)                    if y.shape != z.shape:
(16)                        raise TypeError(
(20)                            f"Shapes of y {y.shape} and z {z.shape} do not match")
(8)                 else:
(12)                    raise TypeError(f"Inputs x and y must be 1D or 2D, not {x.ndim}D")
(8)                 return x, y, z
(4)             def _initialize_x_y(self, z):
(8)                 """
(8)                 Return X, Y arrays such that contour(Z) will match imshow(Z)
(8)                 if origin is not None.
(8)                 The center of pixel Z[i, j] depends on origin:
(8)                 if origin is None, x = j, y = i;
(8)                 if origin is 'lower', x = j + 0.5, y = i + 0.5;
(8)                 if origin is 'upper', x = j + 0.5, y = Nrows - i - 0.5
(8)                 If extent is not None, x and y will be scaled to match,
(8)                 as in imshow.
(8)                 If origin is None and extent is not None, then extent
(8)                 will give the minimum and maximum values of x and y.
(8)                 """
(8)                 if z.ndim != 2:
(12)                    raise TypeError(f"Input z must be 2D, not {z.ndim}D")
(8)                 elif z.shape[0] < 2 or z.shape[1] < 2:
(12)                    raise TypeError(f"Input z must be at least a (2, 2) shaped array,
"
(28)                                    f"but has shape {z.shape}")
(8)                 else:
(12)                    Ny, Nx = z.shape
(8)                 if self.origin is None:  # Not for image-matching.
(12)                    if self.extent is None:
(16)                        return np.meshgrid(np.arange(Nx), np.arange(Ny))
(12)                    else:
(16)                        x0, x1, y0, y1 = self.extent
(16)                        x = np.linspace(x0, x1, Nx)
(16)                        y = np.linspace(y0, y1, Ny)
(16)                        return np.meshgrid(x, y)
(8)                 if self.extent is None:
(12)                    x0, x1, y0, y1 = (0, Nx, 0, Ny)
(8)                 else:
(12)                    x0, x1, y0, y1 = self.extent
(8)                 dx = (x1 - x0) / Nx
(8)                 dy = (y1 - y0) / Ny
(8)                 x = x0 + (np.arange(Nx) + 0.5) * dx
(8)                 y = y0 + (np.arange(Ny) + 0.5) * dy
(8)                 if self.origin == 'upper':
(12)                    y = y[::-1]
(8)                 return np.meshgrid(x, y)
(0)         _docstring.interpd.update(contour_doc="""
(0)         `.contour` and `.contourf` draw contour lines and filled contours,
(0)         respectively.  Except as noted, function signatures and return values
(0)         are the same for both versions.
(0)         Parameters
(0)         ----------
(0)         X, Y : array-like, optional
(4)             The coordinates of the values in *Z*.
(4)             *X* and *Y* must both be 2D with the same shape as *Z* (e.g.
```

```
(4)             created via `numpy.meshgrid`), or they must both be 1-D such
(4)             that ``len(X) == N`` is the number of columns in *Z* and
(4)             ``len(Y) == M`` is the number of rows in *Z*.
(4)             *X* and *Y* must both be ordered monotonically.
(4)             If not given, they are assumed to be integer indices, i.e.
(4)             ``X = range(N)``, ``Y = range(M)``.
(0)         Z : (M, N) array-like
(4)             The height values over which the contour is drawn.  Color-mapping is
(4)             controlled by *cmap*, *norm*, *vmin*, and *vmax*.
(0)         levels : int or array-like, optional
(4)             Determines the number and positions of the contour lines / regions.
(4)             If an int *n*, use `~matplotlib.ticker.MaxNLocator`, which tries
(4)             to automatically choose no more than *n+1* "nice" contour levels
(4)             between minimum and maximum numeric values of *Z*.
(4)             If array-like, draw contour lines at the specified levels.
(4)             The values must be in increasing order.
(0)         Returns
(0)         -------
(0)         `~.contour.QuadContourSet`
(0)         Other Parameters
(0)         ----------------
(0)         corner_mask : bool, default: :rc:`contour.corner_mask`
(4)             Enable/disable corner masking, which only has an effect if *Z* is
(4)             a masked array.  If ``False``, any quad touching a masked point is
(4)             masked out.  If ``True``, only the triangular corners of quads
(4)             nearest those points are always masked out, other triangular
(4)             corners comprising three unmasked points are contoured as usual.
(0)         colors : :mpltype:`color` or list of :mpltype:`color`, optional
(4)             The colors of the levels, i.e. the lines for `.contour` and the
(4)             areas for `.contourf`.
(4)             The sequence is cycled for the levels in ascending order. If the
(4)             sequence is shorter than the number of levels, it's repeated.
(4)             As a shortcut, single color strings may be used in place of
(4)             one-element lists, i.e. ``'red'`` instead of ``['red']`` to color
(4)             all levels with the same color. This shortcut does only work for
(4)             color strings, not for other ways of specifying colors.
(4)             By default (value *None*), the colormap specified by *cmap*
(4)             will be used.
(0)         alpha : float, default: 1
(4)             The alpha blending value, between 0 (transparent) and 1 (opaque).
(0)         %(cmap_doc)s
(4)             This parameter is ignored if *colors* is set.
(0)         %(norm_doc)s
(4)             This parameter is ignored if *colors* is set.
(0)         %(vmin_vmax_doc)s
(4)             If *vmin* or *vmax* are not given, the default color scaling is based on
(4)             *levels*.
(4)             This parameter is ignored if *colors* is set.
(0)         origin : {*None*, 'upper', 'lower', 'image'}, default: None
(4)             Determines the orientation and exact position of *Z* by specifying
(4)             the position of ``Z[0, 0]``.  This is only relevant, if *X*, *Y*
(4)             are not given.
(4)             - *None*: ``Z[0, 0]`` is at X=0, Y=0 in the lower left corner.
(4)             - 'lower': ``Z[0, 0]`` is at X=0.5, Y=0.5 in the lower left corner.
(4)             - 'upper': ``Z[0, 0]`` is at X=N+0.5, Y=0.5 in the upper left
(6)               corner.
(4)             - 'image': Use the value from :rc:`image.origin`.
(0)         extent : (x0, x1, y0, y1), optional
(4)             If *origin* is not *None*, then *extent* is interpreted as in
(4)             `.imshow`: it gives the outer pixel boundaries. In this case, the
(4)             position of Z[0, 0] is the center of the pixel, not a corner. If
(4)             *origin* is *None*, then (*x0*, *y0*) is the position of Z[0, 0],
(4)             and (*x1*, *y1*) is the position of Z[-1, -1].
(4)             This argument is ignored if *X* and *Y* are specified in the call
(4)             to contour.
(0)         locator : ticker.Locator subclass, optional
(4)             The locator is used to determine the contour levels if they
(4)             are not given explicitly via *levels*.
(4)             Defaults to `~.ticker.MaxNLocator`.
```

```
(0)           extend : {'neither', 'both', 'min', 'max'}, default: 'neither'
(4)               Determines the ``contourf``-coloring of values that are outside the
(4)               *levels* range.
(4)               If 'neither', values outside the *levels* range are not colored.
(4)               If 'min', 'max' or 'both', color the values below, above or below
(4)               and above the *levels* range.
(4)               Values below ``min(levels)`` and above ``max(levels)`` are mapped
(4)               to the under/over values of the `.Colormap`. Note that most
(4)               colormaps do not have dedicated colors for these by default, so
(4)               that the over and under values are the edge values of the colormap.
(4)               You may want to set these values explicitly using
(4)               `.Colormap.set_under` and `.Colormap.set_over`.
(4)               .. note::
(8)                   An existing `.QuadContourSet` does not get notified if
(8)                   properties of its colormap are changed. Therefore, an explicit
(8)                   call `.QuadContourSet.changed()` is needed after modifying the
(8)                   colormap. The explicit call can be left out, if a colorbar is
(8)                   assigned to the `.QuadContourSet` because it internally calls
(8)                   `.QuadContourSet.changed()`.
(4)               Example::
(8)                   x = np.arange(1, 10)
(8)                   y = x.reshape(-1, 1)
(8)                   h = x * y
(8)                   cs = plt.contourf(h, levels=[10, 30, 50],
(12)                      colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both')
(8)                   cs.cmap.set_over('red')
(8)                   cs.cmap.set_under('blue')
(8)                   cs.changed()
(0)           xunits, yunits : registered units, optional
(4)               Override axis units by specifying an instance of a
(4)               :class:`matplotlib.units.ConversionInterface`.
(0)           antialiased : bool, optional
(4)               Enable antialiasing, overriding the defaults.  For
(4)               filled contours, the default is *False*.  For line contours,
(4)               it is taken from :rc:`lines.antialiased`.
(0)           nchunk : int >= 0, optional
(4)               If 0, no subdivision of the domain.  Specify a positive integer to
(4)               divide the domain into subdomains of *nchunk* by *nchunk* quads.
(4)               Chunking reduces the maximum length of polygons generated by the
(4)               contouring algorithm which reduces the rendering workload passed
(4)               on to the backend and also requires slightly less RAM.  It can
(4)               however introduce rendering artifacts at chunk boundaries depending
(4)               on the backend, the *antialiased* flag and value of *alpha*.
(0)           linewidths : float or array-like, default: :rc:`contour.linewidth`
(4)               *Only applies to* `.contour`.
(4)               The line width of the contour lines.
(4)               If a number, all levels will be plotted with this linewidth.
(4)               If a sequence, the levels in ascending order will be plotted with
(4)               the linewidths in the order specified.
(4)               If None, this falls back to :rc:`lines.linewidth`.
(0)           linestyles : {*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional
(4)               *Only applies to* `.contour`.
(4)               If *linestyles* is *None*, the default is 'solid' unless the lines are
(4)               monochrome. In that case, negative contours will instead take their
(4)               linestyle from the *negative_linestyles* argument.
(4)               *linestyles* can also be an iterable of the above strings specifying a set
(4)               of linestyles to be used. If this iterable is shorter than the number of
(4)               contour levels it will be repeated as necessary.
(0)           negative_linestyles : {*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, \
(23)                                optional
(4)               *Only applies to* `.contour`.
(4)               If *linestyles* is *None* and the lines are monochrome, this argument
(4)               specifies the line style for negative contours.
(4)               If *negative_linestyles* is *None*, the default is taken from
(4)               :rc:`contour.negative_linestyles`.
(4)               *negative_linestyles* can also be an iterable of the above strings
(4)               specifying a set of linestyles to be used. If this iterable is shorter
than
(4)               the number of contour levels it will be repeated as necessary.
```

```
1412: (0)           hatches : list[str], optional
1413: (4)               *Only applies to* `.contourf`.
1414: (4)               A list of cross hatch patterns to use on the filled areas.
1415: (4)               If None, no hatching will be added to the contour.
1416: (0)           algorithm : {'mpl2005', 'mpl2014', 'serial', 'threaded'}, optional
1417: (4)               Which contouring algorithm to use to calculate the contour lines and
1418: (4)               polygons. The algorithms are implemented in
1419: (4)               `ContourPy <https://github.com/contourpy/contourpy>`_, consult the
1420: (4)               `ContourPy documentation <https://contourpy.readthedocs.io>`_ for
1421: (4)               further information.
1422: (4)               The default is taken from :rc:`contour.algorithm`.
1423: (0)           clip_path : `~matplotlib.patches.Patch` or `.Path` or `.TransformedPath`
1424: (4)               Set the clip path.  See `~matplotlib.artist.Artist.set_clip_path`.
1425: (4)               .. versionadded:: 3.8
1426: (0)           data : indexable object, optional
1427: (4)               DATA_PARAMETER_PLACEHOLDER
1428: (0)           Notes
1429: (0)           -----
1430: (0)           1. `.contourf` differs from the MATLAB version in that it does not draw
1431: (3)              the polygon edges. To draw edges, add line contours with calls to
1432: (3)              `.contour`.
1433: (0)           2. `.contourf` fills intervals that are closed at the top; that is, for
1434: (3)              boundaries *z1* and *z2*, the filled region is::
1435: (6)                 z1 < Z <= z2
1436: (3)              except for the lowest interval, which is closed on both sides (i.e.
1437: (3)              it includes the lowest value).
1438: (0)           3. `.contour` and `.contourf` use a `marching squares
1439: (3)              <https://en.wikipedia.org/wiki/Marching_squares>`_ algorithm to
1440: (3)              compute contour locations.  More information can be found in
1441: (3)              `ContourPy documentation <https://contourpy.readthedocs.io>`_.
1442: (0)           """ % _docstring.interpd.params)
```

File 36 - dates.py:

```
1: (0)              """
2: (0)              Matplotlib provides sophisticated date plotting capabilities, standing on the
3: (0)              shoulders of python :mod:`datetime` and the add-on module dateutil_.
4: (0)              By default, Matplotlib uses the units machinery described in
5: (0)              `~matplotlib.units` to convert `datetime.datetime`, and `numpy.datetime64`
6: (0)              objects when plotted on an x- or y-axis. The user does not
7: (0)              need to do anything for dates to be formatted, but dates often have strict
8: (0)              formatting needs, so this module provides many tick locators and formatters.
9: (0)              A basic example using `numpy.datetime64` is::
10: (4)                 import numpy as np
11: (4)                 times = np.arange(np.datetime64('2001-01-02'),
12: (22)                                  np.datetime64('2002-02-03'), np.timedelta64(75, 'm'))
13: (4)                 y = np.random.randn(len(times))
14: (4)                 fig, ax = plt.subplots()
15: (4)                 ax.plot(times, y)
16: (0)             .. seealso::
17: (4)                 - :doc:`/gallery/text_labels_and_annotations/date`
18: (4)                 - :doc:`/gallery/ticks/date_concise_formatter`
19: (4)                 - :doc:`/gallery/ticks/date_demo_convert`
20: (0)             .. _date-format:
21: (0)             Matplotlib date format
22: (0)             ----------------------
23: (0)             Matplotlib represents dates using floating point numbers specifying the number
24: (0)             of days since a default epoch of 1970-01-01 UTC; for example,
25: (0)             1970-01-01, 06:00 is the floating point number 0.25. The formatters and
26: (0)             locators require the use of `datetime.datetime` objects, so only dates between
27: (0)             year 0001 and 9999 can be represented.  Microsecond precision
28: (0)             is achievable for (approximately) 70 years on either side of the epoch, and
29: (0)             20 microseconds for the rest of the allowable range of dates (year 0001 to
30: (0)             9999). The epoch can be changed at import time via `.dates.set_epoch` or
31: (0)             :rc:`dates.epoch` to other dates if necessary; see
32: (0)             :doc:`/gallery/ticks/date_precision_and_epochs` for a discussion.
33: (0)             .. note::
```

```
  34: (3)                Before Matplotlib 3.3, the epoch was 0000-12-31 which lost modern
  35: (3)                microsecond precision and also made the default axis limit of 0 an invalid
  36: (3)                datetime.  In 3.3 the epoch was changed as above.  To convert old
  37: (3)                ordinal floats to the new epoch, users can do::
  38: (5)                  new_ordinal = old_ordinal + mdates.date2num(np.datetime64('0000-12-31'))
  39: (0)             There are a number of helper functions to convert between :mod:`datetime`
  40: (0)             objects and Matplotlib dates:
  41: (0)             .. currentmodule:: matplotlib.dates
  42: (0)             .. autosummary::
  43: (3)                :nosignatures:
  44: (3)                datestr2num
  45: (3)                date2num
  46: (3)                num2date
  47: (3)                num2timedelta
  48: (3)                drange
  49: (3)                set_epoch
  50: (3)                get_epoch
  51: (0)             .. note::
  52: (3)                Like Python's `datetime.datetime`, Matplotlib uses the Gregorian calendar
  53: (3)                for all conversions between dates and floating point numbers. This practice
  54: (3)                is not universal, and calendar differences can cause confusing
  55: (3)                differences between what Python and Matplotlib give as the number of days
  56: (3)                since 0001-01-01 and what other software and databases yield.  For
  57: (3)                example, the US Naval Observatory uses a calendar that switches
  58: (3)                from Julian to Gregorian in October, 1582.  Hence, using their
  59: (3)                calculator, the number of days between 0001-01-01 and 2006-04-01 is
  60: (3)                732403, whereas using the Gregorian calendar via the datetime
  61: (3)                module we find::
  62: (5)                  In [1]: date(2006, 4, 1).toordinal() - date(1, 1, 1).toordinal()
  63: (5)                  Out[1]: 732401
  64: (0)             All the Matplotlib date converters, locators and formatters are timezone
  aware.
  65: (0)             If no explicit timezone is provided, :rc:`timezone` is assumed, provided as a
  66: (0)             string.  If you want to use a different timezone, pass the *tz* keyword
  67: (0)             argument of `num2date` to any date tick locators or formatters you create.
  This
  68: (0)             can be either a `datetime.tzinfo` instance or a string with the timezone name
  69: (0)             that can be parsed by `~dateutil.tz.gettz`.
  70: (0)             A wide range of specific and general purpose date tick locators and
  71: (0)             formatters are provided in this module.  See
  72: (0)             :mod:`matplotlib.ticker` for general information on tick locators
  73: (0)             and formatters.  These are described below.
  74: (0)             The dateutil_ module provides additional code to handle date ticking, making
  it
  75: (0)             easy to place ticks on any kinds of dates.  See examples below.
  76: (0)             .. _dateutil: https://dateutil.readthedocs.io
  77: (0)             .. _date-locators:
  78: (0)             Date tick locators
  79: (0)             ------------------
  80: (0)             Most of the date tick locators can locate single or multiple ticks. For
  example::
  81: (4)                 from matplotlib.dates import MO, TU, WE, TH, FR, SA, SU
  82: (4)                 loc = WeekdayLocator(byweekday=MO, tz=tz)
  83: (4)                 loc = WeekdayLocator(byweekday=(MO, SA))
  84: (0)             In addition, most of the constructors take an interval argument::
  85: (4)                 loc = WeekdayLocator(byweekday=MO, interval=2)
  86: (0)             The rrule locator allows completely general date ticking::
  87: (4)                 rule = rrulewrapper(YEARLY, byeaster=1, interval=5)
  88: (4)                 loc = RRuleLocator(rule)
  89: (0)             The available date tick locators are:
  90: (0)             * `MicrosecondLocator`: Locate microseconds.
  91: (0)             * `SecondLocator`: Locate seconds.
  92: (0)             * `MinuteLocator`: Locate minutes.
  93: (0)             * `HourLocator`: Locate hours.
  94: (0)             * `DayLocator`: Locate specified days of the month.
  95: (0)             * `WeekdayLocator`: Locate days of the week, e.g., MO, TU.
  96: (0)             * `MonthLocator`: Locate months, e.g., 7 for July.
  97: (0)             * `YearLocator`: Locate years that are multiples of base.
  98: (0)             * `RRuleLocator`: Locate using a `rrulewrapper`.
```

```
 99: (2)                `rrulewrapper` is a simple wrapper around dateutil_'s `dateutil.rrule`
100: (2)                which allow almost arbitrary date tick specifications.
101: (2)                See :doc:`rrule example </gallery/ticks/date_demo_rrule>`.
102: (0)              * `AutoDateLocator`: On autoscale, this class picks the best `DateLocator`
103: (2)                (e.g., `RRuleLocator`) to set the view limits and the tick locations.  If
104: (2)                called with ``interval_multiples=True`` it will make ticks line up with
105: (2)                sensible multiples of the tick intervals.  For example, if the interval is
106: (2)                4 hours, it will pick hours 0, 4, 8, etc. as ticks.  This behaviour is not
107: (2)                guaranteed by default.
108: (0)              .. _date-formatters:
109: (0)              Date formatters
110: (0)              ---------------
111: (0)              The available date formatters are:
112: (0)              * `AutoDateFormatter`: attempts to figure out the best format to use.  This is
113: (2)                most useful when used with the `AutoDateLocator`.
114: (0)              * `ConciseDateFormatter`: also attempts to figure out the best format to use,
115: (2)                and to make the format as compact as possible while still having complete
116: (2)                date information.  This is most useful when used with the `AutoDateLocator`.
117: (0)              * `DateFormatter`: use `~datetime.datetime.strftime` format strings.
118: (0)              """
119: (0)              import datetime
120: (0)              import functools
121: (0)              import logging
122: (0)              import re
123: (0)              from dateutil.rrule import (rrule, MO, TU, WE, TH, FR, SA, SU, YEARLY,
124: (28)                                          MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY,
125: (28)                                          SECONDLY)
126: (0)              from dateutil.relativedelta import relativedelta
127: (0)              import dateutil.parser
128: (0)              import dateutil.tz
129: (0)              import numpy as np
130: (0)              import matplotlib as mpl
131: (0)              from matplotlib import _api, cbook, ticker, units
132: (0)              __all__ = ('datestr2num', 'date2num', 'num2date', 'num2timedelta', 'drange',
133: (11)                         'set_epoch', 'get_epoch', 'DateFormatter', 'ConciseDateFormatter',
134: (11)                         'AutoDateFormatter', 'DateLocator', 'RRuleLocator',
135: (11)                         'AutoDateLocator', 'YearLocator', 'MonthLocator', 'WeekdayLocator',
136: (11)                         'DayLocator', 'HourLocator', 'MinuteLocator',
137: (11)                         'SecondLocator', 'MicrosecondLocator',
138: (11)                         'rrule', 'MO', 'TU', 'WE', 'TH', 'FR', 'SA', 'SU',
139: (11)                         'YEARLY', 'MONTHLY', 'WEEKLY', 'DAILY',
140: (11)                         'HOURLY', 'MINUTELY', 'SECONDLY', 'MICROSECONDLY', 'relativedelta',
141: (11)                         'DateConverter', 'ConciseDateConverter', 'rrulewrapper')
142: (0)              _log = logging.getLogger(__name__)
143: (0)              UTC = datetime.timezone.utc
144: (0)              def _get_tzinfo(tz=None):
145: (4)                  """
146: (4)                  Generate `~datetime.tzinfo` from a string or return `~datetime.tzinfo`.
147: (4)                  If None, retrieve the preferred timezone from the rcParams dictionary.
148: (4)                  """
149: (4)                  tz = mpl._val_or_rc(tz, 'timezone')
150: (4)                  if tz == 'UTC':
151: (8)                      return UTC
152: (4)                  if isinstance(tz, str):
153: (8)                      tzinfo = dateutil.tz.gettz(tz)
154: (8)                      if tzinfo is None:
155: (12)                         raise ValueError(f"{tz} is not a valid timezone as parsed by"
156: (29)                                          " dateutil.tz.gettz.")
157: (8)                      return tzinfo
158: (4)                  if isinstance(tz, datetime.tzinfo):
159: (8)                      return tz
160: (4)                  raise TypeError(f"tz must be string or tzinfo subclass, not {tz!r}.")
161: (0)              EPOCH_OFFSET = float(datetime.datetime(1970, 1, 1).toordinal())
162: (0)              MICROSECONDLY = SECONDLY + 1
163: (0)              HOURS_PER_DAY = 24.
164: (0)              MIN_PER_HOUR = 60.
165: (0)              SEC_PER_MIN = 60.
166: (0)              MONTHS_PER_YEAR = 12.
167: (0)              DAYS_PER_WEEK = 7.
```

```
(0)             DAYS_PER_MONTH = 30.
(0)             DAYS_PER_YEAR = 365.0
(0)             MINUTES_PER_DAY = MIN_PER_HOUR * HOURS_PER_DAY
(0)             SEC_PER_HOUR = SEC_PER_MIN * MIN_PER_HOUR
(0)             SEC_PER_DAY = SEC_PER_HOUR * HOURS_PER_DAY
(0)             SEC_PER_WEEK = SEC_PER_DAY * DAYS_PER_WEEK
(0)             MUSECONDS_PER_DAY = 1e6 * SEC_PER_DAY
(0)             MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY = (
(4)                 MO, TU, WE, TH, FR, SA, SU)
(0)             WEEKDAYS = (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY)
(0)             _epoch = None
(0)             def _reset_epoch_test_example():
(4)                 """
(4)                 Reset the Matplotlib date epoch so it can be set again.
(4)                 Only for use in tests and examples.
(4)                 """
(4)                 global _epoch
(4)                 _epoch = None
(0)             def set_epoch(epoch):
(4)                 """
(4)                 Set the epoch (origin for dates) for datetime calculations.
(4)                 The default epoch is :rc:`dates.epoch` (by default 1970-01-01T00:00).
(4)                 If microsecond accuracy is desired, the date being plotted needs to be
(4)                 within approximately 70 years of the epoch. Matplotlib internally
(4)                 represents dates as days since the epoch, so floating point dynamic
(4)                 range needs to be within a factor of 2^52.
(4)                 `~.dates.set_epoch` must be called before any dates are converted
(4)                 (i.e. near the import section) or a RuntimeError will be raised.
(4)                 See also :doc:`/gallery/ticks/date_precision_and_epochs`.
(4)                 Parameters
(4)                 ----------
(4)                 epoch : str
(8)                     valid UTC date parsable by `numpy.datetime64` (do not include
(8)                     timezone).
(4)                 """
(4)                 global _epoch
(4)                 if _epoch is not None:
(8)                     raise RuntimeError('set_epoch must be called before dates plotted.')
(4)                 _epoch = epoch
(0)             def get_epoch():
(4)                 """
(4)                 Get the epoch used by `.dates`.
(4)                 Returns
(4)                 -------
(4)                 epoch : str
(8)                     String for the epoch (parsable by `numpy.datetime64`).
(4)                 """
(4)                 global _epoch
(4)                 _epoch = mpl._val_or_rc(_epoch, 'date.epoch')
(4)                 return _epoch
(0)             def _dt64_to_ordinalf(d):
(4)                 """
(4)                 Convert `numpy.datetime64` or an `numpy.ndarray` of those types to
(4)                 Gregorian date as UTC float relative to the epoch (see `.get_epoch`).
(4)                 Roundoff is float64 precision.  Practically: microseconds for dates
(4)                 between 290301 BC, 294241 AD, milliseconds for larger dates
(4)                 (see `numpy.datetime64`).
(4)                 """
(4)                 dseconds = d.astype('datetime64[s]')
(4)                 extra = (d - dseconds).astype('timedelta64[ns]')
(4)                 t0 = np.datetime64(get_epoch(), 's')
(4)                 dt = (dseconds - t0).astype(np.float64)
(4)                 dt += extra.astype(np.float64) / 1.0e9
(4)                 dt = dt / SEC_PER_DAY
(4)                 NaT_int = np.datetime64('NaT').astype(np.int64)
(4)                 d_int = d.astype(np.int64)
(4)                 dt[d_int == NaT_int] = np.nan
(4)                 return dt
(0)             def _from_ordinalf(x, tz=None):
```

```
237: (4)                 """
238: (4)                 Convert Gregorian float of the date, preserving hours, minutes,
239: (4)                 seconds and microseconds.  Return value is a `.datetime`.
240: (4)                 The input date *x* is a float in ordinal days at UTC, and the output will
241: (4)                 be the specified `.datetime` object corresponding to that time in
242: (4)                 timezone *tz*, or if *tz* is ``None``, in the timezone specified in
243: (4)                 :rc:`timezone`.
244: (4)                 """
245: (4)                 tz = _get_tzinfo(tz)
246: (4)                 dt = (np.datetime64(get_epoch()) +
247: (10)                      np.timedelta64(int(np.round(x * MUSECONDS_PER_DAY)), 'us'))
248: (4)                 if dt < np.datetime64('0001-01-01') or dt >= np.datetime64('10000-01-01'):
249: (8)                     raise ValueError(f'Date ordinal {x} converts to {dt} (using '
250: (25)                                      f'epoch {get_epoch()}), but Matplotlib dates must be
'
251: (26)                                       'between year 0001 and 9999.')
252: (4)                 dt = dt.tolist()
253: (4)                 dt = dt.replace(tzinfo=dateutil.tz.gettz('UTC'))
254: (4)                 dt = dt.astimezone(tz)
255: (4)                 if np.abs(x) > 70 * 365:
256: (8)                     ms = round(dt.microsecond / 20) * 20
257: (8)                     if ms == 1000000:
258: (12)                        dt = dt.replace(microsecond=0) + datetime.timedelta(seconds=1)
259: (8)                     else:
260: (12)                        dt = dt.replace(microsecond=ms)
261: (4)                 return dt
262: (0)             _from_ordinalf_np_vectorized = np.vectorize(_from_ordinalf, otypes="O")
263: (0)             _dateutil_parser_parse_np_vectorized = np.vectorize(dateutil.parser.parse)
264: (0)             def datestr2num(d, default=None):
265: (4)                 """
266: (4)                 Convert a date string to a datenum using `dateutil.parser.parse`.
267: (4)                 Parameters
268: (4)                 ----------
269: (4)                 d : str or sequence of str
270: (8)                     The dates to convert.
271: (4)                 default : datetime.datetime, optional
272: (8)                     The default date to use when fields are missing in *d*.
273: (4)                 """
274: (4)                 if isinstance(d, str):
275: (8)                     dt = dateutil.parser.parse(d, default=default)
276: (8)                     return date2num(dt)
277: (4)                 else:
278: (8)                     if default is not None:
279: (12)                        d = [date2num(dateutil.parser.parse(s, default=default))
280: (17)                             for s in d]
281: (12)                        return np.asarray(d)
282: (8)                     d = np.asarray(d)
283: (8)                     if not d.size:
284: (12)                        return d
285: (8)                     return date2num(_dateutil_parser_parse_np_vectorized(d))
286: (0)             def date2num(d):
287: (4)                 """
288: (4)                 Convert datetime objects to Matplotlib dates.
289: (4)                 Parameters
290: (4)                 ----------
291: (4)                 d : `datetime.datetime` or `numpy.datetime64` or sequences of these
292: (4)                 Returns
293: (4)                 -------
294: (4)                 float or sequence of floats
295: (8)                     Number of days since the epoch.  See `.get_epoch` for the
296: (8)                     epoch, which can be changed by :rc:`date.epoch` or `.set_epoch`.  If
297: (8)                     the epoch is "1970-01-01T00:00:00" (default) then noon Jan 1 1970
298: (8)                     ("1970-01-01T12:00:00") returns 0.5.
299: (4)                 Notes
300: (4)                 -----
301: (4)                 The Gregorian calendar is assumed; this is not universal practice.
302: (4)                 For details see the module docstring.
303: (4)                 """
304: (4)                 d = cbook._unpack_to_numpy(d)
```

```
(4)                 iterable = np.iterable(d)
(4)                 if not iterable:
(8)                     d = [d]
(4)                 masked = np.ma.is_masked(d)
(4)                 mask = np.ma.getmask(d)
(4)                 d = np.asarray(d)
(4)                 if not np.issubdtype(d.dtype, np.datetime64):
(8)                     if not d.size:
(12)                        return d
(8)                     tzi = getattr(d[0], 'tzinfo', None)
(8)                     if tzi is not None:
(12)                        d = [dt.astimezone(UTC).replace(tzinfo=None) for dt in d]
(12)                        d = np.asarray(d)
(8)                     d = d.astype('datetime64[us]')
(4)                 d = np.ma.masked_array(d, mask=mask) if masked else d
(4)                 d = _dt64_to_ordinalf(d)
(4)                 return d if iterable else d[0]
(0)             def num2date(x, tz=None):
(4)                 """
(4)                 Convert Matplotlib dates to `~datetime.datetime` objects.
(4)                 Parameters
(4)                 ----------
(4)                 x : float or sequence of floats
(8)                     Number of days (fraction part represents hours, minutes, seconds)
(8)                     since the epoch.  See `.get_epoch` for the
(8)                     epoch, which can be changed by :rc:`date.epoch` or `.set_epoch`.
(4)                 tz : str or `~datetime.tzinfo`, default: :rc:`timezone`
(8)                     Timezone of *x*. If a string, *tz* is passed to `dateutil.tz`.
(4)                 Returns
(4)                 -------
(4)                 `~datetime.datetime` or sequence of `~datetime.datetime`
(8)                     Dates are returned in timezone *tz*.
(8)                     If *x* is a sequence, a sequence of `~datetime.datetime` objects will
(8)                     be returned.
(4)                 Notes
(4)                 -----
(4)                 The Gregorian calendar is assumed; this is not universal practice.
(4)                 For details, see the module docstring.
(4)                 """
(4)                 tz = _get_tzinfo(tz)
(4)                 return _from_ordinalf_np_vectorized(x, tz).tolist()
(0)             _ordinalf_to_timedelta_np_vectorized = np.vectorize(
(4)                 lambda x: datetime.timedelta(days=x), otypes="O")
(0)             def num2timedelta(x):
(4)                 """
(4)                 Convert number of days to a `~datetime.timedelta` object.
(4)                 If *x* is a sequence, a sequence of `~datetime.timedelta` objects will
(4)                 be returned.
(4)                 Parameters
(4)                 ----------
(4)                 x : float, sequence of floats
(8)                     Number of days. The fraction part represents hours, minutes, seconds.
(4)                 Returns
(4)                 -------
(4)                 `datetime.timedelta` or list[`datetime.timedelta`]
(4)                 """
(4)                 return _ordinalf_to_timedelta_np_vectorized(x).tolist()
(0)             def drange(dstart, dend, delta):
(4)                 """
(4)                 Return a sequence of equally spaced Matplotlib dates.
(4)                 The dates start at *dstart* and reach up to, but not including *dend*.
(4)                 They are spaced by *delta*.
(4)                 Parameters
(4)                 ----------
(4)                 dstart, dend : `~datetime.datetime`
(8)                     The date limits.
(4)                 delta : `datetime.timedelta`
(8)                     Spacing of the dates.
(4)                 Returns
```

```
374: (4)                -------
375: (4)                `numpy.array`
376: (8)                    A list floats representing Matplotlib dates.
377: (4)                """
378: (4)                f1 = date2num(dstart)
379: (4)                f2 = date2num(dend)
380: (4)                step = delta.total_seconds() / SEC_PER_DAY
381: (4)                num = int(np.ceil((f2 - f1) / step))
382: (4)                dinterval_end = dstart + num * delta
383: (4)                if dinterval_end >= dend:
384: (8)                    dinterval_end -= delta
385: (8)                    num -= 1
386: (4)                f2 = date2num(dinterval_end)  # new float-endpoint
387: (4)                return np.linspace(f1, f2, num + 1)
388: (0)            def _wrap_in_tex(text):
389: (4)                p = r'([a-zA-Z]+)'
390: (4)                ret_text = re.sub(p, r'}$\1$\\mathdefault{', text)
391: (4)                ret_text = ret_text.replace('-', '{-}').replace(':', '{:}')
392: (4)                ret_text = ret_text.replace(' ', r'\;')
393: (4)                ret_text = '$\\mathdefault{' + ret_text + '}$'
394: (4)                ret_text = ret_text.replace('$\\mathdefault{}$', '')
395: (4)                return ret_text
396: (0)            class DateFormatter(ticker.Formatter):
397: (4)                """
398: (4)                Format a tick (in days since the epoch) with a
399: (4)                `~datetime.datetime.strftime` format string.
400: (4)                """
401: (4)                def __init__(self, fmt, tz=None, *, usetex=None):
402: (8)                    """
403: (8)                    Parameters
404: (8)                    ----------
405: (8)                    fmt : str
406: (12)                       `~datetime.datetime.strftime` format string
407: (8)                    tz : str or `~datetime.tzinfo`, default: :rc:`timezone`
408: (12)                       Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.
409: (8)                    usetex : bool, default: :rc:`text.usetex`
410: (12)                       To enable/disable the use of TeX's math mode for rendering the
411: (12)                       results of the formatter.
412: (8)                    """
413: (8)                    self.tz = _get_tzinfo(tz)
414: (8)                    self.fmt = fmt
415: (8)                    self._usetex = mpl._val_or_rc(usetex, 'text.usetex')
416: (4)                def __call__(self, x, pos=0):
417: (8)                    result = num2date(x, self.tz).strftime(self.fmt)
418: (8)                    return _wrap_in_tex(result) if self._usetex else result
419: (4)                def set_tzinfo(self, tz):
420: (8)                    self.tz = _get_tzinfo(tz)
421: (0)            class ConciseDateFormatter(ticker.Formatter):
422: (4)                """
423: (4)                A `.Formatter` which attempts to figure out the best format to use for the
424: (4)                date, and to make it as compact as possible, but still be complete. This
is
425: (4)                most useful when used with the `AutoDateLocator`::
426: (4)                >>> locator = AutoDateLocator()
427: (4)                >>> formatter = ConciseDateFormatter(locator)
428: (4)                Parameters
429: (4)                ----------
430: (4)                locator : `.ticker.Locator`
431: (8)                    Locator that this axis is using.
432: (4)                tz : str or `~datetime.tzinfo`, default: :rc:`timezone`
433: (8)                    Ticks timezone, passed to `.dates.num2date`.
434: (4)                formats : list of 6 strings, optional
435: (8)                    Format strings for 6 levels of tick labelling: mostly years,
436: (8)                    months, days, hours, minutes, and seconds.  Strings use
437: (8)                    the same format codes as `~datetime.datetime.strftime`.  Default is
438: (8)                    ``['%Y', '%b', '%d', '%H:%M', '%H:%M', '%S.%f']``
439: (4)                zero_formats : list of 6 strings, optional
440: (8)                    Format strings for tick labels that are "zeros" for a given tick
441: (8)                    level.  For instance, if most ticks are months, ticks around 1 Jan
```

```
2005
                    will be labeled "Dec", "2005", "Feb".  The default is
                    ``['', '%Y', '%b', '%b-%d', '%H:%M', '%H:%M']``
                offset_formats : list of 6 strings, optional
                    Format strings for the 6 levels that is applied to the "offset"
                    string found on the right side of an x-axis, or top of a y-axis.
                    Combined with the tick labels this should completely specify the
                    date.  The default is::
                        ['', '%Y', '%Y-%b', '%Y-%b-%d', '%Y-%b-%d', '%Y-%b-%d %H:%M']
                show_offset : bool, default: True
                    Whether to show the offset or not.
                usetex : bool, default: :rc:`text.usetex`
                    To enable/disable the use of TeX's math mode for rendering the results
                    of the formatter.
                Examples
                --------
                See :doc:`/gallery/ticks/date_concise_formatter`
                .. plot::
                    import datetime
                    import matplotlib.dates as mdates
                    base = datetime.datetime(2005, 2, 1)
                    dates = np.array([base + datetime.timedelta(hours=(2 * i))
                                      for i in range(732)])
                    N = len(dates)
                    np.random.seed(19680801)
                    y = np.cumsum(np.random.randn(N))
                    fig, ax = plt.subplots(constrained_layout=True)
                    locator = mdates.AutoDateLocator()
                    formatter = mdates.ConciseDateFormatter(locator)
                    ax.xaxis.set_major_locator(locator)
                    ax.xaxis.set_major_formatter(formatter)
                    ax.plot(dates, y)
                    ax.set_title('Concise Date Formatter')
                """
                def __init__(self, locator, tz=None, formats=None, offset_formats=None,
                             zero_formats=None, show_offset=True, *, usetex=None):
                    """
                    Autoformat the date labels.  The default format is used to form an
                    initial string, and then redundant elements are removed.
                    """
                    self._locator = locator
                    self._tz = tz
                    self.defaultfmt = '%Y'
                    if formats:
                        if len(formats) != 6:
                            raise ValueError('formats argument must be a list of '
                                             '6 format strings (or None)')
                        self.formats = formats
                    else:
                        self.formats = ['%Y',  # ticks are mostly years
                                        '%b',          # ticks are mostly months
                                        '%d',          # ticks are mostly days
                                        '%H:%M',       # hrs
                                        '%H:%M',       # min
                                        '%S.%f',       # secs
                                        ]
                    if zero_formats:
                        if len(zero_formats) != 6:
                            raise ValueError('zero_formats argument must be a list of '
                                             '6 format strings (or None)')
                        self.zero_formats = zero_formats
                    elif formats:
                        self.zero_formats = [''] + self.formats[:-1]
                    else:
                        self.zero_formats = [''] + self.formats[:-1]
                        self.zero_formats[3] = '%b-%d'
                    if offset_formats:
                        if len(offset_formats) != 6:
                            raise ValueError('offset_formats argument must be a list of '
```

```
(33)                                        '6 format strings (or None)')
(12)                        self.offset_formats = offset_formats
(8)                     else:
(12)                        self.offset_formats = ['',
(35)                                               '%Y',
(35)                                               '%Y-%b',
(35)                                               '%Y-%b-%d',
(35)                                               '%Y-%b-%d',
(35)                                               '%Y-%b-%d %H:%M']
(8)                     self.offset_string = ''
(8)                     self.show_offset = show_offset
(8)                     self._usetex = mpl._val_or_rc(usetex, 'text.usetex')
(4)                 def __call__(self, x, pos=None):
(8)                     formatter = DateFormatter(self.defaultfmt, self._tz,
(34)                                              usetex=self._usetex)
(8)                     return formatter(x, pos=pos)
(4)                 def format_ticks(self, values):
(8)                     tickdatetime = [num2date(value, tz=self._tz) for value in values]
(8)                     tickdate = np.array([tdt.timetuple()[:6] for tdt in tickdatetime])
(8)                     fmts = self.formats
(8)                     zerofmts = self.zero_formats
(8)                     offsetfmts = self.offset_formats
(8)                     show_offset = self.show_offset
(8)                     for level in range(5, -1, -1):
(12)                        unique = np.unique(tickdate[:, level])
(12)                        if len(unique) > 1:
(16)                            if level < 2 and np.any(unique == 1):
(20)                                show_offset = False
(16)                            break
(12)                        elif level == 0:
(16)                            level = 5
(8)                     zerovals = [0, 1, 1, 0, 0, 0, 0]
(8)                     labels = [''] * len(tickdate)
(8)                     for nn in range(len(tickdate)):
(12)                        if level < 5:
(16)                            if tickdate[nn][level] == zerovals[level]:
(20)                                fmt = zerofmts[level]
(16)                            else:
(20)                                fmt = fmts[level]
(12)                        else:
(16)                            if (tickdatetime[nn].second == tickdatetime[nn].microsecond
(24)                                    == 0):
(20)                                fmt = zerofmts[level]
(16)                            else:
(20)                                fmt = fmts[level]
(12)                        labels[nn] = tickdatetime[nn].strftime(fmt)
(8)                     if level >= 5:
(12)                        trailing_zeros = min(
(16)                            (len(s) - len(s.rstrip('0')) for s in labels if '.' in s),
(16)                            default=None)
(12)                        if trailing_zeros:
(16)                            for nn in range(len(labels)):
(20)                                if '.' in labels[nn]:
(24)                                    labels[nn] = labels[nn][:-trailing_zeros].rstrip('.')
(8)                     if show_offset:
(12)                        self.offset_string = tickdatetime[-1].strftime(offsetfmts[level])
(12)                        if self._usetex:
(16)                            self.offset_string = _wrap_in_tex(self.offset_string)
(8)                     else:
(12)                        self.offset_string = ''
(8)                     if self._usetex:
(12)                        return [_wrap_in_tex(l) for l in labels]
(8)                     else:
(12)                        return labels
(4)                 def get_offset(self):
(8)                     return self.offset_string
(4)                 def format_data_short(self, value):
(8)                     return num2date(value, tz=self._tz).strftime('%Y-%m-%d %H:%M:%S')
(0)             class AutoDateFormatter(ticker.Formatter):
```

```
(4)                 """
(4)                 A `.Formatter` which attempts to figure out the best format to use.  This
(4)                 is most useful when used with the `AutoDateLocator`.
(4)                 `.AutoDateFormatter` has a ``.scale`` dictionary that maps tick scales (the
(4)                 interval in days between one major tick) to format strings; this dictionary
(4)                 defaults to ::
(8)                     self.scaled = {
(12)                        DAYS_PER_YEAR: rcParams['date.autoformatter.year'],
(12)                        DAYS_PER_MONTH: rcParams['date.autoformatter.month'],
(12)                        1: rcParams['date.autoformatter.day'],
(12)                        1 / HOURS_PER_DAY: rcParams['date.autoformatter.hour'],
(12)                        1 / MINUTES_PER_DAY: rcParams['date.autoformatter.minute'],
(12)                        1 / SEC_PER_DAY: rcParams['date.autoformatter.second'],
(12)                        1 / MUSECONDS_PER_DAY: rcParams['date.autoformatter.microsecond'],
(8)                     }
(4)                 The formatter uses the format string corresponding to the lowest key in
(4)                 the dictionary that is greater or equal to the current scale.  Dictionary
(4)                 entries can be customized::
(8)                     locator = AutoDateLocator()
(8)                     formatter = AutoDateFormatter(locator)
(8)                     formatter.scaled[1/(24*60)] = '%M:%S' # only show min and sec
(4)                 Custom callables can also be used instead of format strings.  The following
(4)                 example shows how to use a custom format function to strip trailing zeros
(4)                 from decimal seconds and adds the date to the first ticklabel::
(8)                     def my_format_function(x, pos=None):
(12)                        x = matplotlib.dates.num2date(x)
(12)                        if pos == 0:
(16)                            fmt = '%D %H:%M:%S.%f'
(12)                        else:
(16)                            fmt = '%H:%M:%S.%f'
(12)                        label = x.strftime(fmt)
(12)                        label = label.rstrip("0")
(12)                        label = label.rstrip(".")
(12)                        return label
(8)                     formatter.scaled[1/(24*60)] = my_format_function
(4)                 """
(4)                 def __init__(self, locator, tz=None, defaultfmt='%Y-%m-%d', *,
(17)                             usetex=None):
(8)                     """
(8)                     Autoformat the date labels.
(8)                     Parameters
(8)                     ----------
(8)                     locator : `.ticker.Locator`
(12)                        Locator that this axis is using.
(8)                     tz : str or `~datetime.tzinfo`, default: :rc:`timezone`
(12)                        Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.
(8)                     defaultfmt : str
(12)                        The default format to use if none of the values in ``self.scaled``
(12)                        are greater than the unit returned by ``locator._get_unit()``.
(8)                     usetex : bool, default: :rc:`text.usetex`
(12)                        To enable/disable the use of TeX's math mode for rendering the
(12)                        results of the formatter. If any entries in ``self.scaled`` are set
(12)                        as functions, then it is up to the customized function to enable or
(12)                        disable TeX's math mode itself.
(8)                     """
(8)                     self._locator = locator
(8)                     self._tz = tz
(8)                     self.defaultfmt = defaultfmt
(8)                     self._formatter = DateFormatter(self.defaultfmt, tz)
(8)                     rcParams = mpl.rcParams
(8)                     self._usetex = mpl._val_or_rc(usetex, 'text.usetex')
(8)                     self.scaled = {
(12)                        DAYS_PER_YEAR: rcParams['date.autoformatter.year'],
(12)                        DAYS_PER_MONTH: rcParams['date.autoformatter.month'],
```

```
643: (12)                        1: rcParams['date.autoformatter.day'],
644: (12)                        1 / HOURS_PER_DAY: rcParams['date.autoformatter.hour'],
645: (12)                        1 / MINUTES_PER_DAY: rcParams['date.autoformatter.minute'],
646: (12)                        1 / SEC_PER_DAY: rcParams['date.autoformatter.second'],
647: (12)                        1 / MUSECONDS_PER_DAY: rcParams['date.autoformatter.microsecond']
648: (8)                    }
649: (4)                def _set_locator(self, locator):
650: (8)                    self._locator = locator
651: (4)                def __call__(self, x, pos=None):
652: (8)                    try:
653: (12)                        locator_unit_scale = float(self._locator._get_unit())
654: (8)                    except AttributeError:
655: (12)                        locator_unit_scale = 1
656: (8)                    fmt = next((fmt for scale, fmt in sorted(self.scaled.items())
657: (20)                               if scale >= locator_unit_scale),
658: (19)                              self.defaultfmt)
659: (8)                    if isinstance(fmt, str):
660: (12)                        self._formatter = DateFormatter(fmt, self._tz,
usetex=self._usetex)
661: (12)                        result = self._formatter(x, pos)
662: (8)                    elif callable(fmt):
663: (12)                        result = fmt(x, pos)
664: (8)                    else:
665: (12)                        raise TypeError(f'Unexpected type passed to {self!r}.')
666: (8)                    return result
667: (0)            class rrulewrapper:
668: (4)                """
669: (4)                A simple wrapper around a `dateutil.rrule` allowing flexible
670: (4)                date tick specifications.
671: (4)                """
672: (4)                def __init__(self, freq, tzinfo=None, **kwargs):
673: (8)                    """
674: (8)                    Parameters
675: (8)                    ----------
676: (8)                    freq : {YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY, SECONDLY}
677: (12)                       Tick frequency. These constants are defined in `dateutil.rrule`,
678: (12)                       but they are accessible from `matplotlib.dates` as well.
679: (8)                    tzinfo : `datetime.tzinfo`, optional
680: (12)                       Time zone information. The default is None.
681: (8)                    **kwargs
682: (12)                       Additional keyword arguments are passed to the `dateutil.rrule`.
683: (8)                    """
684: (8)                    kwargs['freq'] = freq
685: (8)                    self._base_tzinfo = tzinfo
686: (8)                    self._update_rrule(**kwargs)
687: (4)                def set(self, **kwargs):
688: (8)                    """Set parameters for an existing wrapper."""
689: (8)                    self._construct.update(kwargs)
690: (8)                    self._update_rrule(**self._construct)
691: (4)                def _update_rrule(self, **kwargs):
692: (8)                    tzinfo = self._base_tzinfo
693: (8)                    if 'dtstart' in kwargs:
694: (12)                       dtstart = kwargs['dtstart']
695: (12)                       if dtstart.tzinfo is not None:
696: (16)                           if tzinfo is None:
697: (20)                               tzinfo = dtstart.tzinfo
698: (16)                           else:
699: (20)                               dtstart = dtstart.astimezone(tzinfo)
700: (16)                           kwargs['dtstart'] = dtstart.replace(tzinfo=None)
701: (8)                    if 'until' in kwargs:
702: (12)                       until = kwargs['until']
703: (12)                       if until.tzinfo is not None:
704: (16)                           if tzinfo is not None:
705: (20)                               until = until.astimezone(tzinfo)
706: (16)                           else:
707: (20)                               raise ValueError('until cannot be aware if dtstart '
708: (37)                                                'is naive and tzinfo is None')
709: (16)                           kwargs['until'] = until.replace(tzinfo=None)
710: (8)                    self._construct = kwargs.copy()
```

```
(8)                 self._tzinfo = tzinfo
(8)                 self._rrule = rrule(**self._construct)
(4)             def _attach_tzinfo(self, dt, tzinfo):
(8)                 if hasattr(tzinfo, 'localize'):
(12)                    return tzinfo.localize(dt, is_dst=True)
(8)                 return dt.replace(tzinfo=tzinfo)
(4)             def _aware_return_wrapper(self, f, returns_list=False):
(8)                 """Decorator function that allows rrule methods to handle tzinfo."""
(8)                 if self._tzinfo is None:
(12)                    return f
(8)                 def normalize_arg(arg):
(12)                    if isinstance(arg, datetime.datetime) and arg.tzinfo is not None:
(16)                        if arg.tzinfo is not self._tzinfo:
(20)                            arg = arg.astimezone(self._tzinfo)
(16)                        return arg.replace(tzinfo=None)
(12)                    return arg
(8)                 def normalize_args(args, kwargs):
(12)                    args = tuple(normalize_arg(arg) for arg in args)
(12)                    kwargs = {kw: normalize_arg(arg) for kw, arg in kwargs.items()}
(12)                    return args, kwargs
(8)                 if not returns_list:
(12)                    def inner_func(*args, **kwargs):
(16)                        args, kwargs = normalize_args(args, kwargs)
(16)                        dt = f(*args, **kwargs)
(16)                        return self._attach_tzinfo(dt, self._tzinfo)
(8)                 else:
(12)                    def inner_func(*args, **kwargs):
(16)                        args, kwargs = normalize_args(args, kwargs)
(16)                        dts = f(*args, **kwargs)
(16)                        return [self._attach_tzinfo(dt, self._tzinfo) for dt in dts]
(8)                 return functools.wraps(f)(inner_func)
(4)             def __getattr__(self, name):
(8)                 if name in self.__dict__:
(12)                    return self.__dict__[name]
(8)                 f = getattr(self._rrule, name)
(8)                 if name in {'after', 'before'}:
(12)                    return self._aware_return_wrapper(f)
(8)                 elif name in {'xafter', 'xbefore', 'between'}:
(12)                    return self._aware_return_wrapper(f, returns_list=True)
(8)                 else:
(12)                    return f
(4)             def __setstate__(self, state):
(8)                 self.__dict__.update(state)
(0)         class DateLocator(ticker.Locator):
(4)             """
(4)             Determines the tick locations when plotting dates.
(4)             This class is subclassed by other Locators and
(4)             is not meant to be used on its own.
(4)             """
(4)             hms0d = {'byhour': 0, 'byminute': 0, 'bysecond': 0}
(4)             def __init__(self, tz=None):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 tz : str or `~datetime.tzinfo`, default: :rc:`timezone`
(12)                    Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.
(8)                 """
(8)                 self.tz = _get_tzinfo(tz)
(4)             def set_tzinfo(self, tz):
(8)                 """
(8)                 Set timezone info.
(8)                 Parameters
(8)                 ----------
(8)                 tz : str or `~datetime.tzinfo`, default: :rc:`timezone`
(12)                    Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.
(8)                 """
(8)                 self.tz = _get_tzinfo(tz)
(4)             def datalim_to_dt(self):
(8)                 """Convert axis data interval to datetime objects."""
```

```
(8)                     dmin, dmax = self.axis.get_data_interval()
(8)                     if dmin > dmax:
(12)                        dmin, dmax = dmax, dmin
(8)                     return num2date(dmin, self.tz), num2date(dmax, self.tz)
(4)                 def viewlim_to_dt(self):
(8)                     """Convert the view interval to datetime objects."""
(8)                     vmin, vmax = self.axis.get_view_interval()
(8)                     if vmin > vmax:
(12)                        vmin, vmax = vmax, vmin
(8)                     return num2date(vmin, self.tz), num2date(vmax, self.tz)
(4)                 def _get_unit(self):
(8)                     """
(8)                     Return how many days a unit of the locator is; used for
(8)                     intelligent autoscaling.
(8)                     """
(8)                     return 1
(4)                 def _get_interval(self):
(8)                     """
(8)                     Return the number of units for each tick.
(8)                     """
(8)                     return 1
(4)                 def nonsingular(self, vmin, vmax):
(8)                     """
(8)                     Given the proposed upper and lower extent, adjust the range
(8)                     if it is too close to being singular (i.e. a range of ~0).
(8)                     """
(8)                     if not np.isfinite(vmin) or not np.isfinite(vmax):
(12)                        return (date2num(datetime.date(1970, 1, 1)),
(20)                                date2num(datetime.date(1970, 1, 2)))
(8)                     if vmax < vmin:
(12)                        vmin, vmax = vmax, vmin
(8)                     unit = self._get_unit()
(8)                     interval = self._get_interval()
(8)                     if abs(vmax - vmin) < 1e-6:
(12)                        vmin -= 2 * unit * interval
(12)                        vmax += 2 * unit * interval
(8)                     return vmin, vmax
(0)             class RRuleLocator(DateLocator):
(4)                 def __init__(self, o, tz=None):
(8)                     super().__init__(tz)
(8)                     self.rule = o
(4)                 def __call__(self):
(8)                     try:
(12)                        dmin, dmax = self.viewlim_to_dt()
(8)                     except ValueError:
(12)                        return []
(8)                     return self.tick_values(dmin, dmax)
(4)                 def tick_values(self, vmin, vmax):
(8)                     start, stop = self._create_rrule(vmin, vmax)
(8)                     dates = self.rule.between(start, stop, True)
(8)                     if len(dates) == 0:
(12)                        return date2num([vmin, vmax])
(8)                     return self.raise_if_exceeds(date2num(dates))
(4)                 def _create_rrule(self, vmin, vmax):
(8)                     delta = relativedelta(vmax, vmin)
(8)                     try:
(12)                        start = vmin - delta
(8)                     except (ValueError, OverflowError):
(12)                        start = datetime.datetime(1, 1, 1, 0, 0, 0,
(38)                                                  tzinfo=datetime.timezone.utc)
(8)                     try:
(12)                        stop = vmax + delta
(8)                     except (ValueError, OverflowError):
(12)                        stop = datetime.datetime(9999, 12, 31, 23, 59, 59,
(37)                                                 tzinfo=datetime.timezone.utc)
(8)                     self.rule.set(dtstart=start, until=stop)
(8)                     return vmin, vmax
(4)                 def _get_unit(self):
(8)                     freq = self.rule._rrule._freq
```

```
849: (8)                    return self.get_unit_generic(freq)
850: (4)                @staticmethod
851: (4)                def get_unit_generic(freq):
852: (8)                    if freq == YEARLY:
853: (12)                       return DAYS_PER_YEAR
854: (8)                    elif freq == MONTHLY:
855: (12)                       return DAYS_PER_MONTH
856: (8)                    elif freq == WEEKLY:
857: (12)                       return DAYS_PER_WEEK
858: (8)                    elif freq == DAILY:
859: (12)                       return 1.0
860: (8)                    elif freq == HOURLY:
861: (12)                       return 1.0 / HOURS_PER_DAY
862: (8)                    elif freq == MINUTELY:
863: (12)                       return 1.0 / MINUTES_PER_DAY
864: (8)                    elif freq == SECONDLY:
865: (12)                       return 1.0 / SEC_PER_DAY
866: (8)                    else:
867: (12)                       return -1  # or should this just return '1'?
868: (4)                def _get_interval(self):
869: (8)                    return self.rule._rrule._interval
870: (0)            class AutoDateLocator(DateLocator):
871: (4)                """
872: (4)                On autoscale, this class picks the best `DateLocator` to set the view
873: (4)                limits and the tick locations.
874: (4)                Attributes
875: (4)                ----------
876: (4)                intervald : dict
877: (8)                    Mapping of tick frequencies to multiples allowed for that ticking.
878: (8)                    The default is ::
879: (12)                       self.intervald = {
880: (16)                           YEARLY  : [1, 2, 4, 5, 10, 20, 40, 50, 100, 200, 400, 500,
881: (27)                                      1000, 2000, 4000, 5000, 10000],
882: (16)                           MONTHLY : [1, 2, 3, 4, 6],
883: (16)                           DAILY   : [1, 2, 3, 7, 14, 21],
884: (16)                           HOURLY  : [1, 2, 3, 4, 6, 12],
885: (16)                           MINUTELY: [1, 5, 10, 15, 30],
886: (16)                           SECONDLY: [1, 5, 10, 15, 30],
887: (16)                           MICROSECONDLY: [1, 2, 5, 10, 20, 50, 100, 200, 500,
888: (32)                                          1000, 2000, 5000, 10000, 20000, 50000,
889: (32)                                          100000, 200000, 500000, 1000000],
890: (12)                       }
891: (8)                    where the keys are defined in `dateutil.rrule`.
892: (8)                    The interval is used to specify multiples that are appropriate for
893: (8)                    the frequency of ticking. For instance, every 7 days is sensible
894: (8)                    for daily ticks, but for minutes/seconds, 15 or 30 make sense.
895: (8)                    When customizing, you should only modify the values for the existing
896: (8)                    keys. You should not add or delete entries.
897: (8)                    Example for forcing ticks every 3 hours::
898: (12)                       locator = AutoDateLocator()
899: (12)                       locator.intervald[HOURLY] = [3]  # only show every 3 hours
900: (4)                """
901: (4)                def __init__(self, tz=None, minticks=5, maxticks=None,
902: (17)                            interval_multiples=True):
903: (8)                    """
904: (8)                    Parameters
905: (8)                    ----------
906: (8)                    tz : str or `~datetime.tzinfo`, default: :rc:`timezone`
907: (12)                       Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.
908: (8)                    minticks : int
909: (12)                       The minimum number of ticks desired; controls whether ticks occur
910: (12)                       yearly, monthly, etc.
911: (8)                    maxticks : int
912: (12)                       The maximum number of ticks desired; controls the interval between
913: (12)                       ticks (ticking every other, every 3, etc.).  For fine-grained
914: (12)                       control, this can be a dictionary mapping individual rrule
915: (12)                       frequency constants (YEARLY, MONTHLY, etc.) to their own maximum
916: (12)                       number of ticks.  This can be used to keep the number of ticks
917: (12)                       appropriate to the format chosen in `AutoDateFormatter`. Any
```

```
(12)                        frequency not specified in this dictionary is given a default
(12)                        value.
(8)                     interval_multiples : bool, default: True
(12)                        Whether ticks should be chosen to be multiple of the interval,
(12)                        locking them to 'nicer' locations.  For example, this will force
(12)                        the ticks to be at hours 0, 6, 12, 18 when hourly ticking is done
(12)                        at 6 hour intervals.
(8)                     """
(8)                     super().__init__(tz=tz)
(8)                     self._freq = YEARLY
(8)                     self._freqs = [YEARLY, MONTHLY, DAILY, HOURLY, MINUTELY,
(23)                                   SECONDLY, MICROSECONDLY]
(8)                     self.minticks = minticks
(8)                     self.maxticks = {YEARLY: 11, MONTHLY: 12, DAILY: 11, HOURLY: 12,
(25)                                     MINUTELY: 11, SECONDLY: 11, MICROSECONDLY: 8}
(8)                     if maxticks is not None:
(12)                        try:
(16)                            self.maxticks.update(maxticks)
(12)                        except TypeError:
(16)                            self.maxticks = dict.fromkeys(self._freqs, maxticks)
(8)                     self.interval_multiples = interval_multiples
(8)                     self.intervald = {
(12)                        YEARLY:   [1, 2, 4, 5, 10, 20, 40, 50, 100, 200, 400, 500,
(23)                                   1000, 2000, 4000, 5000, 10000],
(12)                        MONTHLY:  [1, 2, 3, 4, 6],
(12)                        DAILY:    [1, 2, 3, 7, 14, 21],
(12)                        HOURLY:   [1, 2, 3, 4, 6, 12],
(12)                        MINUTELY: [1, 5, 10, 15, 30],
(12)                        SECONDLY: [1, 5, 10, 15, 30],
(12)                        MICROSECONDLY: [1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000,
(28)                                        5000, 10000, 20000, 50000, 100000, 200000, 500000,
(28)                                        1000000],
(28)                                        }
(8)                     if interval_multiples:
(12)                        self.intervald[DAILY] = [1, 2, 4, 7, 14]
(8)                     self._byranges = [None, range(1, 13), range(1, 32),
(26)                                      range(0, 24), range(0, 60), range(0, 60), None]
(4)                 def __call__(self):
(8)                     dmin, dmax = self.viewlim_to_dt()
(8)                     locator = self.get_locator(dmin, dmax)
(8)                     return locator()
(4)                 def tick_values(self, vmin, vmax):
(8)                     return self.get_locator(vmin, vmax).tick_values(vmin, vmax)
(4)                 def nonsingular(self, vmin, vmax):
(8)                     if not np.isfinite(vmin) or not np.isfinite(vmax):
(12)                        return (date2num(datetime.date(1970, 1, 1)),
(20)                                date2num(datetime.date(1970, 1, 2)))
(8)                     if vmax < vmin:
(12)                        vmin, vmax = vmax, vmin
(8)                     if vmin == vmax:
(12)                        vmin = vmin - DAYS_PER_YEAR * 2
(12)                        vmax = vmax + DAYS_PER_YEAR * 2
(8)                     return vmin, vmax
(4)                 def _get_unit(self):
(8)                     if self._freq in [MICROSECONDLY]:
(12)                        return 1. / MUSECONDS_PER_DAY
(8)                     else:
(12)                        return RRuleLocator.get_unit_generic(self._freq)
(4)                 def get_locator(self, dmin, dmax):
(8)                     """Pick the best locator based on a distance."""
(8)                     delta = relativedelta(dmax, dmin)
(8)                     tdelta = dmax - dmin
(8)                     if dmin > dmax:
(12)                        delta = -delta
(12)                        tdelta = -tdelta
(8)                     numYears = float(delta.years)
(8)                     numMonths = numYears * MONTHS_PER_YEAR + delta.months
(8)                     numDays = tdelta.days  # Avoids estimates of days/month, days/year.
(8)                     numHours = numDays * HOURS_PER_DAY + delta.hours
```

```
(8)                     numMinutes = numHours * MIN_PER_HOUR + delta.minutes
(8)                     numSeconds = np.floor(tdelta.total_seconds())
(8)                     numMicroseconds = np.floor(tdelta.total_seconds() * 1e6)
(8)                     nums = [numYears, numMonths, numDays, numHours, numMinutes,
(16)                            numSeconds, numMicroseconds]
(8)                     use_rrule_locator = [True] * 6 + [False]
(8)                     byranges = [None, 1, 1, 0, 0, 0, None]
(8)                     for i, (freq, num) in enumerate(zip(self._freqs, nums)):
(12)                        if num < self.minticks:
(16)                            byranges[i] = None
(16)                            continue
(12)                        for interval in self.intervald[freq]:
(16)                            if num <= interval * (self.maxticks[freq] - 1):
(20)                                break
(12)                        else:
(16)                            if not (self.interval_multiples and freq == DAILY):
(20)                                _api.warn_external(
(24)                                    f"AutoDateLocator was unable to pick an appropriate "
(24)                                    f"interval for this date range. It may be necessary "
(24)                                    f"to add an interval value to the AutoDateLocator's "
(24)                                    f"intervald dictionary. Defaulting to {interval}.")
(12)                        self._freq = freq
(12)                        if self._byranges[i] and self.interval_multiples:
(16)                            byranges[i] = self._byranges[i][::interval]
(16)                            if i in (DAILY, WEEKLY):
(20)                                if interval == 14:
(24)                                    byranges[i] = [1, 15]
(20)                                elif interval == 7:
(24)                                    byranges[i] = [1, 8, 15, 22]
(16)                            interval = 1
(12)                        else:
(16)                            byranges[i] = self._byranges[i]
(12)                        break
(8)                     else:
(12)                        interval = 1
(8)                     if (freq == YEARLY) and self.interval_multiples:
(12)                        locator = YearLocator(interval, tz=self.tz)
(8)                     elif use_rrule_locator[i]:
(12)                        _, bymonth, bymonthday, byhour, byminute, bysecond, _ = byranges
(12)                        rrule = rrulewrapper(self._freq, interval=interval,
(33)                                             dtstart=dmin, until=dmax,
(33)                                             bymonth=bymonth, bymonthday=bymonthday,
(33)                                             byhour=byhour, byminute=byminute,
(33)                                             bysecond=bysecond)
(12)                        locator = RRuleLocator(rrule, tz=self.tz)
(8)                     else:
(12)                        locator = MicrosecondLocator(interval, tz=self.tz)
(12)                        if date2num(dmin) > 70 * 365 and interval < 1000:
(16)                            _api.warn_external(
(20)                                'Plotting microsecond time intervals for dates far from '
(20)                                f'the epoch (time origin: {get_epoch()}) is not well-'
(20)                                'supported. See matplotlib.dates.set_epoch to change the '
(20)                                'epoch.')
(8)                     locator.set_axis(self.axis)
(8)                     return locator
(0)             class YearLocator(RRuleLocator):
(4)                 """
(4)                 Make ticks on a given day of each year that is a multiple of base.
(4)                 Examples::
(6)                   locator = YearLocator()
(6)                   locator = YearLocator(5, month=7, day=4)
(4)                 """
(4)                 def __init__(self, base=1, month=1, day=1, tz=None):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     base : int, default: 1
(12)                        Mark ticks every *base* years.
(8)                     month : int, default: 1
```

```
(12)                        The month on which to place the ticks, starting from 1. Default is
(12)                        January.
(8)                     day : int, default: 1
(12)                        The day on which to place the ticks.
(8)                     tz : str or `~datetime.tzinfo`, default: :rc:`timezone`
(12)                        Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.
(8)                     """
(8)                     rule = rrulewrapper(YEARLY, interval=base, bymonth=month,
(28)                                        bymonthday=day, **self.hms0d)
(8)                     super().__init__(rule, tz=tz)
(8)                     self.base = ticker._Edge_integer(base, 0)
(4)                 def _create_rrule(self, vmin, vmax):
(8)                     ymin = max(self.base.le(vmin.year) * self.base.step, 1)
(8)                     ymax = min(self.base.ge(vmax.year) * self.base.step, 9999)
(8)                     c = self.rule._construct
(8)                     replace = {'year': ymin,
(19)                               'month': c.get('bymonth', 1),
(19)                               'day': c.get('bymonthday', 1),
(19)                               'hour': 0, 'minute': 0, 'second': 0}
(8)                     start = vmin.replace(**replace)
(8)                     stop = start.replace(year=ymax)
(8)                     self.rule.set(dtstart=start, until=stop)
(8)                     return start, stop
(0)             class MonthLocator(RRuleLocator):
(4)                 """
(4)                 Make ticks on occurrences of each month, e.g., 1, 3, 12.
(4)                 """
(4)                 def __init__(self, bymonth=None, bymonthday=1, interval=1, tz=None):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     bymonth : int or list of int, default: all months
(12)                        Ticks will be placed on every month in *bymonth*. Default is
(12)                        ``range(1, 13)``, i.e. every month.
(8)                     bymonthday : int, default: 1
(12)                        The day on which to place the ticks.
(8)                     interval : int, default: 1
(12)                        The interval between each iteration. For example, if
(12)                        ``interval=2``, mark every second occurrence.
(8)                     tz : str or `~datetime.tzinfo`, default: :rc:`timezone`
(12)                        Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.
(8)                     """
(8)                     if bymonth is None:
(12)                        bymonth = range(1, 13)
(8)                     rule = rrulewrapper(MONTHLY, bymonth=bymonth, bymonthday=bymonthday,
(28)                                        interval=interval, **self.hms0d)
(8)                     super().__init__(rule, tz=tz)
(0)             class WeekdayLocator(RRuleLocator):
(4)                 """
(4)                 Make ticks on occurrences of each weekday.
(4)                 """
(4)                 def __init__(self, byweekday=1, interval=1, tz=None):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     byweekday : int or list of int, default: all days
(12)                        Ticks will be placed on every weekday in *byweekday*. Default is
(12)                        every day.
(12)                        Elements of *byweekday* must be one of MO, TU, WE, TH, FR, SA,
(12)                        SU, the constants from :mod:`dateutil.rrule`, which have been
(12)                        imported into the :mod:`matplotlib.dates` namespace.
(8)                     interval : int, default: 1
(12)                        The interval between each iteration. For example, if
(12)                        ``interval=2``, mark every second occurrence.
(8)                     tz : str or `~datetime.tzinfo`, default: :rc:`timezone`
(12)                        Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.
(8)                     """
(8)                     rule = rrulewrapper(DAILY, byweekday=byweekday,
(28)                                        interval=interval, **self.hms0d)
```

```
(8)                 super().__init__(rule, tz=tz)
(0)         class DayLocator(RRuleLocator):
(4)             """
(4)             Make ticks on occurrences of each day of the month.  For example,
(4)             1, 15, 30.
(4)             """
(4)             def __init__(self, bymonthday=None, interval=1, tz=None):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 bymonthday : int or list of int, default: all days
(12)                    Ticks will be placed on every day in *bymonthday*. Default is
(12)                    ``bymonthday=range(1, 32)``, i.e., every day of the month.
(8)                 interval : int, default: 1
(12)                    The interval between each iteration. For example, if
(12)                    ``interval=2``, mark every second occurrence.
(8)                 tz : str or `~datetime.tzinfo`, default: :rc:`timezone`
(12)                    Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.
(8)                 """
(8)                 if interval != int(interval) or interval < 1:
(12)                    raise ValueError("interval must be an integer greater than 0")
(8)                 if bymonthday is None:
(12)                    bymonthday = range(1, 32)
(8)                 rule = rrulewrapper(DAILY, bymonthday=bymonthday,
(28)                                    interval=interval, **self.hms0d)
(8)                 super().__init__(rule, tz=tz)
(0)         class HourLocator(RRuleLocator):
(4)             """
(4)             Make ticks on occurrences of each hour.
(4)             """
(4)             def __init__(self, byhour=None, interval=1, tz=None):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 byhour : int or list of int, default: all hours
(12)                    Ticks will be placed on every hour in *byhour*. Default is
(12)                    ``byhour=range(24)``, i.e., every hour.
(8)                 interval : int, default: 1
(12)                    The interval between each iteration. For example, if
(12)                    ``interval=2``, mark every second occurrence.
(8)                 tz : str or `~datetime.tzinfo`, default: :rc:`timezone`
(12)                    Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.
(8)                 """
(8)                 if byhour is None:
(12)                    byhour = range(24)
(8)                 rule = rrulewrapper(HOURLY, byhour=byhour, interval=interval,
(28)                                    byminute=0, bysecond=0)
(8)                 super().__init__(rule, tz=tz)
(0)         class MinuteLocator(RRuleLocator):
(4)             """
(4)             Make ticks on occurrences of each minute.
(4)             """
(4)             def __init__(self, byminute=None, interval=1, tz=None):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 byminute : int or list of int, default: all minutes
(12)                    Ticks will be placed on every minute in *byminute*. Default is
(12)                    ``byminute=range(60)``, i.e., every minute.
(8)                 interval : int, default: 1
(12)                    The interval between each iteration. For example, if
(12)                    ``interval=2``, mark every second occurrence.
(8)                 tz : str or `~datetime.tzinfo`, default: :rc:`timezone`
(12)                    Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.
(8)                 """
(8)                 if byminute is None:
(12)                    byminute = range(60)
(8)                 rule = rrulewrapper(MINUTELY, byminute=byminute, interval=interval,
(28)                                    bysecond=0)
```

```
(8)                 super().__init__(rule, tz=tz)
(0)         class SecondLocator(RRuleLocator):
(4)             """
(4)             Make ticks on occurrences of each second.
(4)             """
(4)             def __init__(self, bysecond=None, interval=1, tz=None):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 bysecond : int or list of int, default: all seconds
(12)                    Ticks will be placed on every second in *bysecond*. Default is
(12)                    ``bysecond = range(60)``, i.e., every second.
(8)                 interval : int, default: 1
(12)                    The interval between each iteration. For example, if
(12)                    ``interval=2``, mark every second occurrence.
(8)                 tz : str or `~datetime.tzinfo`, default: :rc:`timezone`
(12)                    Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.
(8)                 """
(8)                 if bysecond is None:
(12)                    bysecond = range(60)
(8)                 rule = rrulewrapper(SECONDLY, bysecond=bysecond, interval=interval)
(8)                 super().__init__(rule, tz=tz)
(0)         class MicrosecondLocator(DateLocator):
(4)             """
(4)             Make ticks on regular intervals of one or more microsecond(s).
(4)             .. note::
(8)                 By default, Matplotlib uses a floating point representation of time in
(8)                 days since the epoch, so plotting data with
(8)                 microsecond time resolution does not work well for
(8)                 dates that are far (about 70 years) from the epoch (check with
(8)                 `~.dates.get_epoch`).
(8)                 If you want sub-microsecond resolution time plots, it is strongly
(8)                 recommended to use floating point seconds, not datetime-like
(8)                 time representation.
(8)                 If you really must use datetime.datetime() or similar and still
(8)                 need microsecond precision, change the time origin via
(8)                 `.dates.set_epoch` to something closer to the dates being plotted.
(8)                 See :doc:`/gallery/ticks/date_precision_and_epochs`.
(4)             """
(4)             def __init__(self, interval=1, tz=None):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 interval : int, default: 1
(12)                    The interval between each iteration. For example, if
(12)                    ``interval=2``, mark every second occurrence.
(8)                 tz : str or `~datetime.tzinfo`, default: :rc:`timezone`
(12)                    Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.
(8)                 """
(8)                 super().__init__(tz=tz)
(8)                 self._interval = interval
(8)                 self._wrapped_locator = ticker.MultipleLocator(interval)
(4)             def set_axis(self, axis):
(8)                 self._wrapped_locator.set_axis(axis)
(8)                 return super().set_axis(axis)
(4)             def __call__(self):
(8)                 try:
(12)                    dmin, dmax = self.viewlim_to_dt()
(8)                 except ValueError:
(12)                    return []
(8)                 return self.tick_values(dmin, dmax)
(4)             def tick_values(self, vmin, vmax):
(8)                 nmin, nmax = date2num((vmin, vmax))
(8)                 t0 = np.floor(nmin)
(8)                 nmax = nmax - t0
(8)                 nmin = nmin - t0
(8)                 nmin *= MUSECONDS_PER_DAY
(8)                 nmax *= MUSECONDS_PER_DAY
(8)                 ticks = self._wrapped_locator.tick_values(nmin, nmax)
```

```
1263: (8)                   ticks = ticks / MUSECONDS_PER_DAY + t0
1264: (8)                   return ticks
1265: (4)               def _get_unit(self):
1266: (8)                   return 1. / MUSECONDS_PER_DAY
1267: (4)               def _get_interval(self):
1268: (8)                   return self._interval
1269: (0)           class DateConverter(units.ConversionInterface):
1270: (4)               """
1271: (4)               Converter for `datetime.date` and `datetime.datetime` data, or for
1272: (4)               date/time data represented as it would be converted by `date2num`.
1273: (4)               The 'unit' tag for such data is None or a `~datetime.tzinfo` instance.
1274: (4)               """
1275: (4)               def __init__(self, *, interval_multiples=True):
1276: (8)                   self._interval_multiples = interval_multiples
1277: (8)                   super().__init__()
1278: (4)               def axisinfo(self, unit, axis):
1279: (8)                   """
1280: (8)                   Return the `~matplotlib.units.AxisInfo` for *unit*.
1281: (8)                   *unit* is a `~datetime.tzinfo` instance or None.
1282: (8)                   The *axis* argument is required but not used.
1283: (8)                   """
1284: (8)                   tz = unit
1285: (8)                   majloc = AutoDateLocator(tz=tz,
1286: (33)                                           interval_multiples=self._interval_multiples)
1287: (8)                   majfmt = AutoDateFormatter(majloc, tz=tz)
1288: (8)                   datemin = datetime.date(1970, 1, 1)
1289: (8)                   datemax = datetime.date(1970, 1, 2)
1290: (8)                   return units.AxisInfo(majloc=majloc, majfmt=majfmt, label='',
1291: (30)                                        default_limits=(datemin, datemax))
1292: (4)               @staticmethod
1293: (4)               def convert(value, unit, axis):
1294: (8)                   """
1295: (8)                   If *value* is not already a number or sequence of numbers, convert it
1296: (8)                   with `date2num`.
1297: (8)                   The *unit* and *axis* arguments are not used.
1298: (8)                   """
1299: (8)                   return date2num(value)
1300: (4)               @staticmethod
1301: (4)               def default_units(x, axis):
1302: (8)                   """
1303: (8)                   Return the `~datetime.tzinfo` instance of *x* or of its first element,
1304: (8)                   or None
1305: (8)                   """
1306: (8)                   if isinstance(x, np.ndarray):
1307: (12)                      x = x.ravel()
1308: (8)                   try:
1309: (12)                      x = cbook._safe_first_finite(x)
1310: (8)                   except (TypeError, StopIteration):
1311: (12)                      pass
1312: (8)                   try:
1313: (12)                      return x.tzinfo
1314: (8)                   except AttributeError:
1315: (12)                      pass
1316: (8)                   return None
1317: (0)           class ConciseDateConverter(DateConverter):
1318: (4)               def __init__(self, formats=None, zero_formats=None, offset_formats=None,
1319: (17)                           show_offset=True, *, interval_multiples=True):
1320: (8)                   self._formats = formats
1321: (8)                   self._zero_formats = zero_formats
1322: (8)                   self._offset_formats = offset_formats
1323: (8)                   self._show_offset = show_offset
1324: (8)                   self._interval_multiples = interval_multiples
1325: (8)                   super().__init__()
1326: (4)               def axisinfo(self, unit, axis):
1327: (8)                   tz = unit
1328: (8)                   majloc = AutoDateLocator(tz=tz,
1329: (33)                                           interval_multiples=self._interval_multiples)
1330: (8)                   majfmt = ConciseDateFormatter(majloc, tz=tz, formats=self._formats,
1331: (38)                                                zero_formats=self._zero_formats,
```

```
1332: (38)                                                offset_formats=self._offset_formats,
1333: (38)                                                show_offset=self._show_offset)
1334: (8)                   datemin = datetime.date(1970, 1, 1)
1335: (8)                   datemax = datetime.date(1970, 1, 2)
1336: (8)                   return units.AxisInfo(majloc=majloc, majfmt=majfmt, label='',
1337: (30)                                        default_limits=(datemin, datemax))
1338: (0)           class _SwitchableDateConverter:
1339: (4)               """
1340: (4)               Helper converter-like object that generates and dispatches to
1341: (4)               temporary ConciseDateConverter or DateConverter instances based on
1342: (4)               :rc:`date.converter` and :rc:`date.interval_multiples`.
1343: (4)               """
1344: (4)               @staticmethod
1345: (4)               def _get_converter():
1346: (8)                   converter_cls = {
1347: (12)                      "concise": ConciseDateConverter, "auto": DateConverter}[
1348: (16)                          mpl.rcParams["date.converter"]]
1349: (8)                   interval_multiples = mpl.rcParams["date.interval_multiples"]
1350: (8)                   return converter_cls(interval_multiples=interval_multiples)
1351: (4)               def axisinfo(self, *args, **kwargs):
1352: (8)                   return self._get_converter().axisinfo(*args, **kwargs)
1353: (4)               def default_units(self, *args, **kwargs):
1354: (8)                   return self._get_converter().default_units(*args, **kwargs)
1355: (4)               def convert(self, *args, **kwargs):
1356: (8)                   return self._get_converter().convert(*args, **kwargs)
1357: (0)           units.registry[np.datetime64] = \
1358: (4)               units.registry[datetime.date] = \
1359: (4)               units.registry[datetime.datetime] = \
1360: (4)               _SwitchableDateConverter()
```

File 37 - dviread.py:

```
1: (0)              """
2: (0)              A module for reading dvi files output by TeX. Several limitations make
3: (0)              this not (currently) useful as a general-purpose dvi preprocessor, but
4: (0)              it is currently used by the pdf backend for processing usetex text.
5: (0)              Interface::
6: (2)                with Dvi(filename, 72) as dvi:
7: (6)                    for page in dvi:
8: (10)                       w, h, d = page.width, page.height, page.descent
9: (10)                       for x, y, font, glyph, width in page.text:
10: (14)                          fontname = font.texname
11: (14)                          pointsize = font.size
12: (14)                          ...
13: (10)                      for x, y, height, width in page.boxes:
14: (14)                          ...
15: (0)             """
16: (0)             from collections import namedtuple
17: (0)             import enum
18: (0)             from functools import lru_cache, partial, wraps
19: (0)             import logging
20: (0)             import os
21: (0)             from pathlib import Path
22: (0)             import re
23: (0)             import struct
24: (0)             import subprocess
25: (0)             import sys
26: (0)             import numpy as np
27: (0)             from matplotlib import _api, cbook
28: (0)             _log = logging.getLogger(__name__)
29: (0)             _dvistate = enum.Enum('DviState', 'pre outer inpage post_post finale')
30: (0)             Page = namedtuple('Page', 'text boxes height width descent')
31: (0)             Box = namedtuple('Box', 'x y height width')
32: (0)             class Text(namedtuple('Text', 'x y font glyph width')):
33: (4)                 """
34: (4)                 A glyph in the dvi file.
35: (4)                 The *x* and *y* attributes directly position the glyph.  The *font*,
```

```
 36: (4)                 *glyph*, and *width* attributes are kept public for back-compatibility,
 37: (4)                 but users wanting to draw the glyph themselves are encouraged to instead
 38: (4)                 load the font specified by `font_path` at `font_size`, warp it with the
 39: (4)                 effects specified by `font_effects`, and load the glyph specified by
 40: (4)                 `glyph_name_or_index`.
 41: (4)                 """
 42: (4)                 def _get_pdftexmap_entry(self):
 43: (8)                     return PsfontsMap(find_tex_file("pdftex.map"))[self.font.texname]
 44: (4)                 @property
 45: (4)                 def font_path(self):
 46: (8)                     """The `~pathlib.Path` to the font for this glyph."""
 47: (8)                     psfont = self._get_pdftexmap_entry()
 48: (8)                     if psfont.filename is None:
 49: (12)                        raise ValueError("No usable font file found for {} ({}); "
 50: (29)                                         "the font may lack a Type-1 version"
 51: (29)                                         .format(psfont.psname.decode("ascii"),
 52: (37)                                                 psfont.texname.decode("ascii")))
 53: (8)                     return Path(psfont.filename)
 54: (4)                 @property
 55: (4)                 def font_size(self):
 56: (8)                     """The font size."""
 57: (8)                     return self.font.size
 58: (4)                 @property
 59: (4)                 def font_effects(self):
 60: (8)                     """
 61: (8)                     The "font effects" dict for this glyph.
 62: (8)                     This dict contains the values for this glyph of SlantFont and
 63: (8)                     ExtendFont (if any), read off :file:`pdftex.map`.
 64: (8)                     """
 65: (8)                     return self._get_pdftexmap_entry().effects
 66: (4)                 @property
 67: (4)                 def glyph_name_or_index(self):
 68: (8)                     """
 69: (8)                     Either the glyph name or the native charmap glyph index.
 70: (8)                     If :file:`pdftex.map` specifies an encoding for this glyph's font,
that
 71: (8)                     is a mapping of glyph indices to Adobe glyph names; use it to convert
 72: (8)                     dvi indices to glyph names.  Callers can then convert glyph names to
 73: (8)                     glyph indices (with FT_Get_Name_Index/get_name_index), and load the
 74: (8)                     glyph using FT_Load_Glyph/load_glyph.
 75: (8)                     If :file:`pdftex.map` specifies no encoding, the indices directly map
 76: (8)                     to the font's "native" charmap; glyphs should directly load using
 77: (8)                     FT_Load_Char/load_char after selecting the native charmap.
 78: (8)                     """
 79: (8)                     entry = self._get_pdftexmap_entry()
 80: (8)                     return (_parse_enc(entry.encoding)[self.glyph]
 81: (16)                            if entry.encoding is not None else self.glyph)
 82: (0)             _arg_mapping = dict(
 83: (4)                 raw=lambda dvi, delta: delta,
 84: (4)                 u1=lambda dvi, delta: dvi._arg(1, signed=False),
 85: (4)                 u4=lambda dvi, delta: dvi._arg(4, signed=False),
 86: (4)                 s4=lambda dvi, delta: dvi._arg(4, signed=True),
 87: (4)                 slen=lambda dvi, delta: dvi._arg(delta, signed=True) if delta else None,
 88: (4)                 slen1=lambda dvi, delta: dvi._arg(delta + 1, signed=True),
 89: (4)                 ulen1=lambda dvi, delta: dvi._arg(delta + 1, signed=False),
 90: (4)                 olen1=lambda dvi, delta: dvi._arg(delta + 1, signed=(delta == 3)),
 91: (0)             )
 92: (0)             def _dispatch(table, min, max=None, state=None, args=('raw',)):
 93: (4)                 """
 94: (4)                 Decorator for dispatch by opcode. Sets the values in *table*
 95: (4)                 from *min* to *max* to this method, adds a check that the Dvi state
 96: (4)                 matches *state* if not None, reads arguments from the file according
 97: (4)                 to *args*.
 98: (4)                 Parameters
 99: (4)                 ----------
100: (4)                 table : dict[int, callable]
101: (8)                     The dispatch table to be filled in.
102: (4)                 min, max : int
103: (8)                     Range of opcodes that calls the registered function; *max* defaults to
```

```
104: (8)                    *min*.
105: (4)                state : _dvistate, optional
106: (8)                    State of the Dvi object in which these opcodes are allowed.
107: (4)                args : list[str], default: ['raw']
108: (8)                    Sequence of argument specifications:
109: (8)                    - 'raw': opcode minus minimum
110: (8)                    - 'u1': read one unsigned byte
111: (8)                    - 'u4': read four bytes, treat as an unsigned number
112: (8)                    - 's4': read four bytes, treat as a signed number
113: (8)                    - 'slen': read (opcode - minimum) bytes, treat as signed
114: (8)                    - 'slen1': read (opcode - minimum + 1) bytes, treat as signed
115: (8)                    - 'ulen1': read (opcode - minimum + 1) bytes, treat as unsigned
116: (8)                    - 'olen1': read (opcode - minimum + 1) bytes, treat as unsigned
117: (10)                     if under four bytes, signed if four bytes
118: (4)                """
119: (4)                def decorate(method):
120: (8)                    get_args = [_arg_mapping[x] for x in args]
121: (8)                    @wraps(method)
122: (8)                    def wrapper(self, byte):
123: (12)                       if state is not None and self.state != state:
124: (16)                           raise ValueError("state precondition failed")
125: (12)                       return method(self, *[f(self, byte-min) for f in get_args])
126: (8)                    if max is None:
127: (12)                       table[min] = wrapper
128: (8)                    else:
129: (12)                       for i in range(min, max+1):
130: (16)                           assert table[i] is None
131: (16)                           table[i] = wrapper
132: (8)                    return wrapper
133: (4)                return decorate
134: (0)            class Dvi:
135: (4)                """
136: (4)                A reader for a dvi ("device-independent") file, as produced by TeX.
137: (4)                The current implementation can only iterate through pages in order,
138: (4)                and does not even attempt to verify the postamble.
139: (4)                This class can be used as a context manager to close the underlying
140: (4)                file upon exit. Pages can be read via iteration. Here is an overly
141: (4)                simple way to extract text without trying to detect whitespace::
142: (8)                    >>> with matplotlib.dviread.Dvi('input.dvi', 72) as dvi:
143: (8)                    ...     for page in dvi:
144: (8)                    ...         print(''.join(chr(t.glyph) for t in page.text))
145: (4)                """
146: (4)                _dtable = [None] * 256
147: (4)                _dispatch = partial(_dispatch, _dtable)
148: (4)                def __init__(self, filename, dpi):
149: (8)                    """
150: (8)                    Read the data from the file named *filename* and convert
151: (8)                    TeX's internal units to units of *dpi* per inch.
152: (8)                    *dpi* only sets the units and does not limit the resolution.
153: (8)                    Use None to return TeX's internal units.
154: (8)                    """
155: (8)                    _log.debug('Dvi: %s', filename)
156: (8)                    self.file = open(filename, 'rb')
157: (8)                    self.dpi = dpi
158: (8)                    self.fonts = {}
159: (8)                    self.state = _dvistate.pre
160: (4)                def __enter__(self):
161: (8)                    """Context manager enter method, does nothing."""
162: (8)                    return self
163: (4)                def __exit__(self, etype, evalue, etrace):
164: (8)                    """
165: (8)                    Context manager exit method, closes the underlying file if it is open.
166: (8)                    """
167: (8)                    self.close()
168: (4)                def __iter__(self):
169: (8)                    """
170: (8)                    Iterate through the pages of the file.
171: (8)                    Yields
172: (8)                    ------
```

```
(8)                     Page
(12)                        Details of all the text and box objects on the page.
(12)                        The Page tuple contains lists of Text and Box tuples and
(12)                        the page dimensions, and the Text and Box tuples contain
(12)                        coordinates transformed into a standard Cartesian
(12)                        coordinate system at the dpi value given when initializing.
(12)                        The coordinates are floating point numbers, but otherwise
(12)                        precision is not lost and coordinate values are not clipped to
(12)                        integers.
(8)                     """
(8)                     while self._read():
(12)                        yield self._output()
(4)                 def close(self):
(8)                     """Close the underlying file if it is open."""
(8)                     if not self.file.closed:
(12)                        self.file.close()
(4)                 def _output(self):
(8)                     """
(8)                     Output the text and boxes belonging to the most recent page.
(8)                     page = dvi._output()
(8)                     """
(8)                     minx, miny, maxx, maxy = np.inf, np.inf, -np.inf, -np.inf
(8)                     maxy_pure = -np.inf
(8)                     for elt in self.text + self.boxes:
(12)                        if isinstance(elt, Box):
(16)                            x, y, h, w = elt
(16)                            e = 0  # zero depth
(12)                        else:  # glyph
(16)                            x, y, font, g, w = elt
(16)                            h, e = font._height_depth_of(g)
(12)                        minx = min(minx, x)
(12)                        miny = min(miny, y - h)
(12)                        maxx = max(maxx, x + w)
(12)                        maxy = max(maxy, y + e)
(12)                        maxy_pure = max(maxy_pure, y)
(8)                     if self._baseline_v is not None:
(12)                        maxy_pure = self._baseline_v  # This should normally be the case.
(12)                        self._baseline_v = None
(8)                     if not self.text and not self.boxes:  # Avoid infs/nans from inf+/-
inf.
(12)                        return Page(text=[], boxes=[], width=0, height=0, descent=0)
(8)                     if self.dpi is None:
(12)                        return Page(text=self.text, boxes=self.boxes,
(24)                                    width=maxx-minx, height=maxy_pure-miny,
(24)                                    descent=maxy-maxy_pure)
(8)                     d = self.dpi / (72.27 * 2**16)
(8)                     descent = (maxy - maxy_pure) * d
(8)                     text = [Text((x-minx)*d, (maxy-y)*d - descent, f, g, w*d)
(16)                            for (x, y, f, g, w) in self.text]
(8)                     boxes = [Box((x-minx)*d, (maxy-y)*d - descent, h*d, w*d)
(17)                             for (x, y, h, w) in self.boxes]
(8)                     return Page(text=text, boxes=boxes, width=(maxx-minx)*d,
(20)                                height=(maxy_pure-miny)*d, descent=descent)
(4)                 def _read(self):
(8)                     """
(8)                     Read one page from the file. Return True if successful,
(8)                     False if there were no more pages.
(8)                     """
(8)                     down_stack = [0]
(8)                     self._baseline_v = None
(8)                     while True:
(12)                        byte = self.file.read(1)[0]
(12)                        self._dtable[byte](self, byte)
(12)                        name = self._dtable[byte].__name__
(12)                        if name == "_push":
(16)                            down_stack.append(down_stack[-1])
(12)                        elif name == "_pop":
(16)                            down_stack.pop()
(12)                        elif name == "_down":
```

```
(16)                            down_stack[-1] += 1
(12)                        if (self._baseline_v is None
(20)                                and len(getattr(self, "stack", [])) == 3
(20)                                and down_stack[-1] >= 4):
(16)                            self._baseline_v = self.v
(12)                        if byte == 140:                         # end of page
(16)                            return True
(12)                        if self.state is _dvistate.post_post:   # end of file
(16)                            self.close()
(16)                            return False
(4)                 def _arg(self, nbytes, signed=False):
(8)                     """
(8)                     Read and return a big-endian integer *nbytes* long.
(8)                     Signedness is determined by the *signed* keyword.
(8)                     """
(8)                     return int.from_bytes(self.file.read(nbytes), "big", signed=signed)
(4)                 @_dispatch(min=0, max=127, state=_dvistate.inpage)
(4)                 def _set_char_immediate(self, char):
(8)                     self._put_char_real(char)
(8)                     self.h += self.fonts[self.f]._width_of(char)
(4)                 @_dispatch(min=128, max=131, state=_dvistate.inpage, args=('olen1',))
(4)                 def _set_char(self, char):
(8)                     self._put_char_real(char)
(8)                     self.h += self.fonts[self.f]._width_of(char)
(4)                 @_dispatch(132, state=_dvistate.inpage, args=('s4', 's4'))
(4)                 def _set_rule(self, a, b):
(8)                     self._put_rule_real(a, b)
(8)                     self.h += b
(4)                 @_dispatch(min=133, max=136, state=_dvistate.inpage, args=('olen1',))
(4)                 def _put_char(self, char):
(8)                     self._put_char_real(char)
(4)                 def _put_char_real(self, char):
(8)                     font = self.fonts[self.f]
(8)                     if font._vf is None:
(12)                        self.text.append(Text(self.h, self.v, font, char,
(34)                                              font._width_of(char)))
(8)                     else:
(12)                        scale = font._scale
(12)                        for x, y, f, g, w in font._vf[char].text:
(16)                            newf = DviFont(scale=_mul2012(scale, f._scale),
(31)                                           tfm=f._tfm, texname=f.texname, vf=f._vf)
(16)                            self.text.append(Text(self.h + _mul2012(x, scale),
(38)                                                  self.v + _mul2012(y, scale),
(38)                                                  newf, g, newf._width_of(g)))
(12)                        self.boxes.extend([Box(self.h + _mul2012(x, scale),
(35)                                               self.v + _mul2012(y, scale),
(35)                                               _mul2012(a, scale), _mul2012(b, scale))
(31)                                           for x, y, a, b in font._vf[char].boxes])
(4)                 @_dispatch(137, state=_dvistate.inpage, args=('s4', 's4'))
(4)                 def _put_rule(self, a, b):
(8)                     self._put_rule_real(a, b)
(4)                 def _put_rule_real(self, a, b):
(8)                     if a > 0 and b > 0:
(12)                        self.boxes.append(Box(self.h, self.v, a, b))
(4)                 @_dispatch(138)
(4)                 def _nop(self, _):
(8)                     pass
(4)                 @_dispatch(139, state=_dvistate.outer, args=('s4',)*11)
(4)                 def _bop(self, c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, p):
(8)                     self.state = _dvistate.inpage
(8)                     self.h, self.v, self.w, self.x, self.y, self.z = 0, 0, 0, 0, 0, 0
(8)                     self.stack = []
(8)                     self.text = []          # list of Text objects
(8)                     self.boxes = []         # list of Box objects
(4)                 @_dispatch(140, state=_dvistate.inpage)
(4)                 def _eop(self, _):
(8)                     self.state = _dvistate.outer
(8)                     del self.h, self.v, self.w, self.x, self.y, self.z, self.stack
(4)                 @_dispatch(141, state=_dvistate.inpage)
```

```
(4)     def _push(self, _):
(8)         self.stack.append((self.h, self.v, self.w, self.x, self.y, self.z))
(4)     @_dispatch(142, state=_dvistate.inpage)
(4)     def _pop(self, _):
(8)         self.h, self.v, self.w, self.x, self.y, self.z = self.stack.pop()
(4)     @_dispatch(min=143, max=146, state=_dvistate.inpage, args=('slen1',))
(4)     def _right(self, b):
(8)         self.h += b
(4)     @_dispatch(min=147, max=151, state=_dvistate.inpage, args=('slen',))
(4)     def _right_w(self, new_w):
(8)         if new_w is not None:
(12)            self.w = new_w
(8)         self.h += self.w
(4)     @_dispatch(min=152, max=156, state=_dvistate.inpage, args=('slen',))
(4)     def _right_x(self, new_x):
(8)         if new_x is not None:
(12)            self.x = new_x
(8)         self.h += self.x
(4)     @_dispatch(min=157, max=160, state=_dvistate.inpage, args=('slen1',))
(4)     def _down(self, a):
(8)         self.v += a
(4)     @_dispatch(min=161, max=165, state=_dvistate.inpage, args=('slen',))
(4)     def _down_y(self, new_y):
(8)         if new_y is not None:
(12)            self.y = new_y
(8)         self.v += self.y
(4)     @_dispatch(min=166, max=170, state=_dvistate.inpage, args=('slen',))
(4)     def _down_z(self, new_z):
(8)         if new_z is not None:
(12)            self.z = new_z
(8)         self.v += self.z
(4)     @_dispatch(min=171, max=234, state=_dvistate.inpage)
(4)     def _fnt_num_immediate(self, k):
(8)         self.f = k
(4)     @_dispatch(min=235, max=238, state=_dvistate.inpage, args=('olen1',))
(4)     def _fnt_num(self, new_f):
(8)         self.f = new_f
(4)     @_dispatch(min=239, max=242, args=('ulen1',))
(4)     def _xxx(self, datalen):
(8)         special = self.file.read(datalen)
(8)         _log.debug(
(12)            'Dvi._xxx: encountered special: %s',
(12)            ''.join([chr(ch) if 32 <= ch < 127 else '<%02x>' % ch
(21)                     for ch in special]))
(4)     @_dispatch(min=243, max=246, args=('olen1', 'u4', 'u4', 'u4', 'u1', 'u1'))
(4)     def _fnt_def(self, k, c, s, d, a, l):
(8)         self._fnt_def_real(k, c, s, d, a, l)
(4)     def _fnt_def_real(self, k, c, s, d, a, l):
(8)         n = self.file.read(a + l)
(8)         fontname = n[-l:].decode('ascii')
(8)         tfm = _tfmfile(fontname)
(8)         if c != 0 and tfm.checksum != 0 and c != tfm.checksum:
(12)            raise ValueError('tfm checksum mismatch: %s' % n)
(8)         try:
(12)            vf = _vffile(fontname)
(8)         except FileNotFoundError:
(12)            vf = None
(8)         self.fonts[k] = DviFont(scale=s, tfm=tfm, texname=n, vf=vf)
(4)     @_dispatch(247, state=_dvistate.pre, args=('u1', 'u4', 'u4', 'u4', 'u1'))
(4)     def _pre(self, i, num, den, mag, k):
(8)         self.file.read(k)  # comment in the dvi file
(8)         if i != 2:
(12)            raise ValueError("Unknown dvi format %d" % i)
(8)         if num != 25400000 or den != 7227 * 2**16:
(12)            raise ValueError("Nonstandard units in dvi file")
(8)         if mag != 1000:
(12)            raise ValueError("Nonstandard magnification in dvi file")
(8)         self.state = _dvistate.outer
(4)     @_dispatch(248, state=_dvistate.outer)
```

```
(4)     def _post(self, _):
(8)         self.state = _dvistate.post_post
(4)     @_dispatch(249)
(4)     def _post_post(self, _):
(8)         raise NotImplementedError
(4)     @_dispatch(min=250, max=255)
(4)     def _malformed(self, offset):
(8)         raise ValueError(f"unknown command: byte {250 + offset}")
(0)  class DviFont:
(4)     """
(4)     Encapsulation of a font that a DVI file can refer to.
(4)     This class holds a font's texname and size, supports comparison,
(4)     and knows the widths of glyphs in the same units as the AFM file.
(4)     There are also internal attributes (for use by dviread.py) that
(4)     are *not* used for comparison.
(4)     The size is in Adobe points (converted from TeX points).
(4)     Parameters
(4)     ----------
(4)     scale : float
(8)         Factor by which the font is scaled from its natural size.
(4)     tfm : Tfm
(8)         TeX font metrics for this font
(4)     texname : bytes
(7)        Name of the font as used internally by TeX and friends, as an ASCII
(7)        bytestring.  This is usually very different from any external font
(7)        names; `PsfontsMap` can be used to find the external name of the font.
(4)     vf : Vf
(7)        A TeX "virtual font" file, or None if this font is not virtual.
(4)     Attributes
(4)     ----------
(4)     texname : bytes
(4)     size : float
(7)        Size of the font in Adobe points, converted from the slightly
(7)        smaller TeX points.
(4)     widths : list
(7)        Widths of glyphs in glyph-space units, typically 1/1000ths of
(7)        the point size.
(4)     """
(4)     __slots__ = ('texname', 'size', 'widths', '_scale', '_vf', '_tfm')
(4)     def __init__(self, scale, tfm, texname, vf):
(8)         _api.check_isinstance(bytes, texname=texname)
(8)         self._scale = scale
(8)         self._tfm = tfm
(8)         self.texname = texname
(8)         self._vf = vf
(8)         self.size = scale * (72.0 / (72.27 * 2**16))
(8)         try:
(12)            nchars = max(tfm.width) + 1
(8)         except ValueError:
(12)            nchars = 0
(8)         self.widths = [(1000*tfm.width.get(char, 0)) >> 20
(23)                       for char in range(nchars)]
(4)     def __eq__(self, other):
(8)         return (type(self) is type(other)
(16)                and self.texname == other.texname and self.size == other.size)
(4)     def __ne__(self, other):
(8)         return not self.__eq__(other)
(4)     def __repr__(self):
(8)         return f"<{type(self).__name__}: {self.texname}>"
(4)     def _width_of(self, char):
(8)         """Width of char in dvi units."""
(8)         width = self._tfm.width.get(char, None)
(8)         if width is not None:
(12)            return _mul2012(width, self._scale)
(8)         _log.debug('No width for char %d in font %s.', char, self.texname)
(8)         return 0
(4)     def _height_depth_of(self, char):
(8)         """Height and depth of char in dvi units."""
(8)         result = []
```

```
448: (8)                    for metric, name in ((self._tfm.height, "height"),
449: (29)                                         (self._tfm.depth, "depth")):
450: (12)                       value = metric.get(char, None)
451: (12)                       if value is None:
452: (16)                           _log.debug('No %s for char %d in font %s',
453: (27)                                      name, char, self.texname)
454: (16)                           result.append(0)
455: (12)                       else:
456: (16)                           result.append(_mul2012(value, self._scale))
457: (8)                    if re.match(br'^cmsy\d+$', self.texname) and char == 0:
458: (12)                       result[-1] = 0
459: (8)                    return result
460: (0)            class Vf(Dvi):
461: (4)                r"""
462: (4)                A virtual font (\*.vf file) containing subroutines for dvi files.
463: (4)                Parameters
464: (4)                ----------
465: (4)                filename : str or path-like
466: (4)                Notes
467: (4)                -----
468: (4)                The virtual font format is a derivative of dvi:
469: (4)                http://mirrors.ctan.org/info/knuth/virtual-fonts
470: (4)                This class reuses some of the machinery of `Dvi`
471: (4)                but replaces the `_read` loop and dispatch mechanism.
472: (4)                Examples
473: (4)                --------
474: (4)                ::
475: (8)                    vf = Vf(filename)
476: (8)                    glyph = vf[code]
477: (8)                    glyph.text, glyph.boxes, glyph.width
478: (4)                """
479: (4)                def __init__(self, filename):
480: (8)                    super().__init__(filename, 0)
481: (8)                    try:
482: (12)                       self._first_font = None
483: (12)                       self._chars = {}
484: (12)                       self._read()
485: (8)                    finally:
486: (12)                       self.close()
487: (4)                def __getitem__(self, code):
488: (8)                    return self._chars[code]
489: (4)                def _read(self):
490: (8)                    """
491: (8)                    Read one page from the file. Return True if successful,
492: (8)                    False if there were no more pages.
493: (8)                    """
494: (8)                    packet_char, packet_ends = None, None
495: (8)                    packet_len, packet_width = None, None
496: (8)                    while True:
497: (12)                       byte = self.file.read(1)[0]
498: (12)                       if self.state is _dvistate.inpage:
499: (16)                           byte_at = self.file.tell()-1
500: (16)                           if byte_at == packet_ends:
501: (20)                               self._finalize_packet(packet_char, packet_width)
502: (20)                               packet_len, packet_char, packet_width = None, None, None
503: (16)                           elif byte_at > packet_ends:
504: (20)                               raise ValueError("Packet length mismatch in vf file")
505: (16)                           else:
506: (20)                               if byte in (139, 140) or byte >= 243:
507: (24)                                   raise ValueError(
508: (28)                                       "Inappropriate opcode %d in vf file" % byte)
509: (20)                               Dvi._dtable[byte](self, byte)
510: (20)                               continue
511: (12)                       if byte < 242:          # a short packet (length given by byte)
512: (16)                           packet_len = byte
513: (16)                           packet_char, packet_width = self._arg(1), self._arg(3)
514: (16)                           packet_ends = self._init_packet(byte)
515: (16)                           self.state = _dvistate.inpage
516: (12)                       elif byte == 242:       # a long packet
```

```
517: (16)                            packet_len, packet_char, packet_width = \
518: (28)                                        [self._arg(x) for x in (4, 4, 4)]
519: (16)                            self._init_packet(packet_len)
520: (12)                        elif 243 <= byte <= 246:
521: (16)                            k = self._arg(byte - 242, byte == 246)
522: (16)                            c, s, d, a, l = [self._arg(x) for x in (4, 4, 4, 1, 1)]
523: (16)                            self._fnt_def_real(k, c, s, d, a, l)
524: (16)                            if self._first_font is None:
525: (20)                                self._first_font = k
526: (12)                        elif byte == 247:       # preamble
527: (16)                            i, k = self._arg(1), self._arg(1)
528: (16)                            x = self.file.read(k)
529: (16)                            cs, ds = self._arg(4), self._arg(4)
530: (16)                            self._pre(i, x, cs, ds)
531: (12)                        elif byte == 248:       # postamble (just some number of 248s)
532: (16)                            break
533: (12)                        else:
534: (16)                            raise ValueError("Unknown vf opcode %d" % byte)
535: (4)                    def _init_packet(self, pl):
536: (8)                        if self.state != _dvistate.outer:
537: (12)                           raise ValueError("Misplaced packet in vf file")
538: (8)                        self.h, self.v, self.w, self.x, self.y, self.z = 0, 0, 0, 0, 0, 0
539: (8)                        self.stack, self.text, self.boxes = [], [], []
540: (8)                        self.f = self._first_font
541: (8)                        return self.file.tell() + pl
542: (4)                    def _finalize_packet(self, packet_char, packet_width):
543: (8)                        self._chars[packet_char] = Page(
544: (12)                           text=self.text, boxes=self.boxes, width=packet_width,
545: (12)                           height=None, descent=None)
546: (8)                        self.state = _dvistate.outer
547: (4)                    def _pre(self, i, x, cs, ds):
548: (8)                        if self.state is not _dvistate.pre:
549: (12)                           raise ValueError("pre command in middle of vf file")
550: (8)                        if i != 202:
551: (12)                           raise ValueError("Unknown vf format %d" % i)
552: (8)                        if len(x):
553: (12)                           _log.debug('vf file comment: %s', x)
554: (8)                        self.state = _dvistate.outer
555: (0)                def _mul2012(num1, num2):
556: (4)                    """Multiply two numbers in 20.12 fixed point format."""
557: (4)                    return (num1*num2) >> 20
558: (0)                class Tfm:
559: (4)                    """
560: (4)                    A TeX Font Metric file.
561: (4)                    This implementation covers only the bare minimum needed by the Dvi class.
562: (4)                    Parameters
563: (4)                    ----------
564: (4)                    filename : str or path-like
565: (4)                    Attributes
566: (4)                    ----------
567: (4)                    checksum : int
568: (7)                       Used for verifying against the dvi file.
569: (4)                    design_size : int
570: (7)                       Design size of the font (unknown units)
571: (4)                    width, height, depth : dict
572: (7)                       Dimensions of each character, need to be scaled by the factor
573: (7)                       specified in the dvi file. These are dicts because indexing may
574: (7)                       not start from 0.
575: (4)                    """
576: (4)                    __slots__ = ('checksum', 'design_size', 'width', 'height', 'depth')
577: (4)                    def __init__(self, filename):
578: (8)                        _log.debug('opening tfm file %s', filename)
579: (8)                        with open(filename, 'rb') as file:
580: (12)                           header1 = file.read(24)
581: (12)                           lh, bc, ec, nw, nh, nd = struct.unpack('!6H', header1[2:14])
582: (12)                           _log.debug('lh=%d, bc=%d, ec=%d, nw=%d, nh=%d, nd=%d',
583: (23)                                      lh, bc, ec, nw, nh, nd)
584: (12)                           header2 = file.read(4*lh)
585: (12)                           self.checksum, self.design_size = struct.unpack('!2I',
```

```
header2[:8])
586: (12)                        char_info = file.read(4*(ec-bc+1))
587: (12)                        widths = struct.unpack(f'!{nw}i', file.read(4*nw))
588: (12)                        heights = struct.unpack(f'!{nh}i', file.read(4*nh))
589: (12)                        depths = struct.unpack(f'!{nd}i', file.read(4*nd))
590: (8)                    self.width, self.height, self.depth = {}, {}, {}
591: (8)                    for idx, char in enumerate(range(bc, ec+1)):
592: (12)                        byte0 = char_info[4*idx]
593: (12)                        byte1 = char_info[4*idx+1]
594: (12)                        self.width[char] = widths[byte0]
595: (12)                        self.height[char] = heights[byte1 >> 4]
596: (12)                        self.depth[char] = depths[byte1 & 0xf]
597: (0)            PsFont = namedtuple('PsFont', 'texname psname effects encoding filename')
598: (0)            class PsfontsMap:
599: (4)                """
600: (4)                A psfonts.map formatted file, mapping TeX fonts to PS fonts.
601: (4)                Parameters
602: (4)                ----------
603: (4)                filename : str or path-like
604: (4)                Notes
605: (4)                -----
606: (4)                For historical reasons, TeX knows many Type-1 fonts by different
607: (4)                names than the outside world. (For one thing, the names have to
608: (4)                fit in eight characters.) Also, TeX's native fonts are not Type-1
609: (4)                but Metafont, which is nontrivial to convert to PostScript except
610: (4)                as a bitmap. While high-quality conversions to Type-1 format exist
611: (4)                and are shipped with modern TeX distributions, we need to know
612: (4)                which Type-1 fonts are the counterparts of which native fonts. For
613: (4)                these reasons a mapping is needed from internal font names to font
614: (4)                file names.
615: (4)                A texmf tree typically includes mapping files called e.g.
616: (4)                :file:`psfonts.map`, :file:`pdftex.map`, or :file:`dvipdfm.map`.
617: (4)                The file :file:`psfonts.map` is used by :program:`dvips`,
618: (4)                :file:`pdftex.map` by :program:`pdfTeX`, and :file:`dvipdfm.map`
619: (4)                by :program:`dvipdfm`. :file:`psfonts.map` might avoid embedding
620: (4)                the 35 PostScript fonts (i.e., have no filename for them, as in
621: (4)                the Times-Bold example above), while the pdf-related files perhaps
622: (4)                only avoid the "Base 14" pdf fonts. But the user may have
623: (4)                configured these files differently.
624: (4)                Examples
625: (4)                --------
626: (4)                >>> map = PsfontsMap(find_tex_file('pdftex.map'))
627: (4)                >>> entry = map[b'ptmbo8r']
628: (4)                >>> entry.texname
629: (4)                b'ptmbo8r'
630: (4)                >>> entry.psname
631: (4)                b'Times-Bold'
632: (4)                >>> entry.encoding
633: (4)                '/usr/local/texlive/2008/texmf-dist/fonts/enc/dvips/base/8r.enc'
634: (4)                >>> entry.effects
635: (4)                {'slant': 0.16700000000000001}
636: (4)                >>> entry.filename
637: (4)                """
638: (4)                __slots__ = ('_filename', '_unparsed', '_parsed')
639: (4)                @lru_cache
640: (4)                def __new__(cls, filename):
641: (8)                    self = object.__new__(cls)
642: (8)                    self._filename = os.fsdecode(filename)
643: (8)                    with open(filename, 'rb') as file:
644: (12)                       self._unparsed = {}
645: (12)                       for line in file:
646: (16)                           tfmname = line.split(b' ', 1)[0]
647: (16)                           self._unparsed.setdefault(tfmname, []).append(line)
648: (8)                    self._parsed = {}
649: (8)                    return self
650: (4)                def __getitem__(self, texname):
651: (8)                    assert isinstance(texname, bytes)
652: (8)                    if texname in self._unparsed:
653: (12)                       for line in self._unparsed.pop(texname):
```

```
654: (16)                            if self._parse_and_cache_line(line):
655: (20)                                break
656: (8)                    try:
657: (12)                       return self._parsed[texname]
658: (8)                    except KeyError:
659: (12)                       raise LookupError(
660: (16)                           f"An associated PostScript font (required by Matplotlib) "
661: (16)                           f"could not be found for TeX font {texname.decode('ascii')!r}
"
662: (16)                           f"in {self._filename!r}; this problem can often be solved by "
663: (16)                           f"installing a suitable PostScript font package in your TeX "
664: (16)                           f"package manager") from None
665: (4)                def _parse_and_cache_line(self, line):
666: (8)                    """
667: (8)                    Parse a line in the font mapping file.
668: (8)                    The format is (partially) documented at
669: (8)                    http://mirrors.ctan.org/systems/doc/pdftex/manual/pdftex-a.pdf
670: (8)                    https://tug.org/texinfohtml/dvips.html#psfonts_002emap
671: (8)                    Each line can have the following fields:
672: (8)                    - tfmname (first, only required field),
673: (8)                    - psname (defaults to tfmname, must come immediately after tfmname if
674: (10)                     present),
675: (8)                    - fontflags (integer, must come immediately after psname if present,
676: (10)                     ignored by us),
677: (8)                    - special (SlantFont and ExtendFont, only field that is double-
quoted),
678: (8)                    - fontfile, encodingfile (optional, prefixed by <, <<, or <[; <<
always
679: (10)                     precedes a font, <[ always precedes an encoding, < can precede
either
680: (10)                     but then an encoding file must have extension .enc; < and << also
681: (10)                     request different font subsetting behaviors but we ignore that; <
can
682: (10)                     be separated from the filename by whitespace).
683: (8)                    special, fontfile, and encodingfile can appear in any order.
684: (8)                    """
685: (8)                    if not line or line.startswith((b" ", b"%", b"*", b";", b"#")):
686: (12)                       return
687: (8)                    tfmname = basename = special = encodingfile = fontfile = None
688: (8)                    is_subsetted = is_t1 = is_truetype = False
689: (8)                    matches = re.finditer(br'"([^"]*)(?:"|$)|(\S+)', line)
690: (8)                    for match in matches:
691: (12)                       quoted, unquoted = match.groups()
692: (12)                       if unquoted:
693: (16)                           if unquoted.startswith(b"<<"):  # font
694: (20)                               fontfile = unquoted[2:]
695: (16)                           elif unquoted.startswith(b"<["):  # encoding
696: (20)                               encodingfile = unquoted[2:]
697: (16)                           elif unquoted.startswith(b"<"):  # font or encoding
698: (20)                               word = (
699: (24)                                   unquoted[1:]
700: (24)                                   or next(filter(None, next(matches).groups())))
701: (20)                               if word.endswith(b".enc"):
702: (24)                                   encodingfile = word
703: (20)                               else:
704: (24)                                   fontfile = word
705: (24)                                   is_subsetted = True
706: (16)                           elif tfmname is None:
707: (20)                               tfmname = unquoted
708: (16)                           elif basename is None:
709: (20)                               basename = unquoted
710: (12)                       elif quoted:
711: (16)                           special = quoted
712: (8)                    effects = {}
713: (8)                    if special:
714: (12)                       words = reversed(special.split())
715: (12)                       for word in words:
716: (16)                           if word == b"SlantFont":
717: (20)                               effects["slant"] = float(next(words))
```

```
(16)                            elif word == b"ExtendFont":
(20)                                effects["extend"] = float(next(words))
(8)                     if fontfile is not None:
(12)                        if fontfile.endswith((b".ttf", b".ttc")):
(16)                            is_truetype = True
(12)                        elif not fontfile.endswith(b".otf"):
(16)                            is_t1 = True
(8)                     elif basename is not None:
(12)                        is_t1 = True
(8)                     if is_truetype and is_subsetted and encodingfile is None:
(12)                        return
(8)                     if not is_t1 and ("slant" in effects or "extend" in effects):
(12)                        return
(8)                     if abs(effects.get("slant", 0)) > 1:
(12)                        return
(8)                     if abs(effects.get("extend", 0)) > 2:
(12)                        return
(8)                     if basename is None:
(12)                        basename = tfmname
(8)                     if encodingfile is not None:
(12)                        encodingfile = find_tex_file(encodingfile)
(8)                     if fontfile is not None:
(12)                        fontfile = find_tex_file(fontfile)
(8)                     self._parsed[tfmname] = PsFont(
(12)                        texname=tfmname, psname=basename, effects=effects,
(12)                        encoding=encodingfile, filename=fontfile)
(8)                     return True
(0)             def _parse_enc(path):
(4)                 r"""
(4)                 Parse a \*.enc file referenced from a psfonts.map style file.
(4)                 The format supported by this function is a tiny subset of PostScript.
(4)                 Parameters
(4)                 ----------
(4)                 path : `os.PathLike`
(4)                 Returns
(4)                 -------
(4)                 list
(8)                     The nth entry of the list is the PostScript glyph name of the nth
(8)                     glyph.
(4)                 """
(4)                 no_comments = re.sub("%.*", "", Path(path).read_text(encoding="ascii"))
(4)                 array = re.search(r"(?s)\[(.*)\]", no_comments).group(1)
(4)                 lines = [line for line in array.split() if line]
(4)                 if all(line.startswith("/") for line in lines):
(8)                     return [line[1:] for line in lines]
(4)                 else:
(8)                     raise ValueError(f"Failed to parse {path} as Postscript encoding")
(0)             class _LuatexKpsewhich:
(4)                 @lru_cache  # A singleton.
(4)                 def __new__(cls):
(8)                     self = object.__new__(cls)
(8)                     self._proc = self._new_proc()
(8)                     return self
(4)                 def _new_proc(self):
(8)                     return subprocess.Popen(
(12)                        ["luatex", "--luaonly",
(13)                         str(cbook._get_data_path("kpsewhich.lua"))],
(12)                        stdin=subprocess.PIPE, stdout=subprocess.PIPE)
(4)                 def search(self, filename):
(8)                     if self._proc.poll() is not None:  # Dead, restart it.
(12)                        self._proc = self._new_proc()
(8)                     self._proc.stdin.write(os.fsencode(filename) + b"\n")
(8)                     self._proc.stdin.flush()
(8)                     out = self._proc.stdout.readline().rstrip()
(8)                     return None if out == b"nil" else os.fsdecode(out)
(0)             @lru_cache
(0)             def find_tex_file(filename):
(4)                 """
(4)                 Find a file in the texmf tree using kpathsea_.
```

```
(4)                 The kpathsea library, provided by most existing TeX distributions, both
(4)                 on Unix-like systems and on Windows (MikTeX), is invoked via a long-lived
(4)                 luatex process if luatex is installed, or via kpsewhich otherwise.
(4)                 .. _kpathsea: https://www.tug.org/kpathsea/
(4)                 Parameters
(4)                 ----------
(4)                 filename : str or path-like
(4)                 Raises
(4)                 ------
(4)                 FileNotFoundError
(8)                     If the file is not found.
(4)                 """
(4)                 if isinstance(filename, bytes):
(8)                     filename = filename.decode('utf-8', errors='replace')
(4)                 try:
(8)                     lk = _LuatexKpsewhich()
(4)                 except FileNotFoundError:
(8)                     lk = None  # Fallback to directly calling kpsewhich, as below.
(4)                 if lk:
(8)                     path = lk.search(filename)
(4)                 else:
(8)                     if sys.platform == 'win32':
(12)                        kwargs = {'env': {**os.environ, 'command_line_encoding': 'utf-8'},
(22)                                  'encoding': 'utf-8'}
(8)                     else:  # On POSIX, run through the equivalent of os.fsdecode().
(12)                        kwargs = {'encoding': sys.getfilesystemencoding(),
(22)                                  'errors': 'surrogateescape'}
(8)                     try:
(12)                        path = (cbook._check_and_log_subprocess(['kpsewhich', filename],
(52)                                                                _log, **kwargs)
(20)                                .rstrip('\n'))
(8)                     except (FileNotFoundError, RuntimeError):
(12)                        path = None
(4)                 if path:
(8)                     return path
(4)                 else:
(8)                     raise FileNotFoundError(
(12)                        f"Matplotlib's TeX implementation searched for a file named "
(12)                        f"{filename!r} in your texmf tree, but could not find it")
(0)             @lru_cache
(0)             def _fontfile(cls, suffix, texname):
(4)                 return cls(find_tex_file(texname + suffix))
(0)             _tfmfile = partial(_fontfile, Tfm, ".tfm")
(0)             _vffile = partial(_fontfile, Vf, ".vf")
(0)             if __name__ == '__main__':
(4)                 from argparse import ArgumentParser
(4)                 import itertools
(4)                 parser = ArgumentParser()
(4)                 parser.add_argument("filename")
(4)                 parser.add_argument("dpi", nargs="?", type=float, default=None)
(4)                 args = parser.parse_args()
(4)                 with Dvi(args.filename, args.dpi) as dvi:
(8)                     fontmap = PsfontsMap(find_tex_file('pdftex.map'))
(8)                     for page in dvi:
(12)                        print(f"=== new page === "
(18)                              f"(w: {page.width}, h: {page.height}, d: {page.descent})")
(12)                        for font, group in itertools.groupby(
(20)                                page.text, lambda text: text.font):
(16)                            print(f"font: {font.texname.decode('latin-1')!r}\t"
(22)                                  f"scale: {font._scale / 2 ** 20}")
(16)                            print("x", "y", "glyph", "chr", "w", "(glyphs)", sep="\t")
(16)                            for text in group:
(20)                                print(text.x, text.y, text.glyph,
(26)                                      chr(text.glyph) if chr(text.glyph).isprintable()
(26)                                      else ".",
(26)                                      text.width, sep="\t")
(12)                        if page.boxes:
(16)                            print("x", "y", "h", "w", "", "(boxes)", sep="\t")
(16)                            for box in page.boxes:
```

```
856: (20)                                print(box.x, box.y, box.height, box.width, sep="\t")

----------------------------------------

File 38 - figure.py:

1: (0)              """
2: (0)              `matplotlib.figure` implements the following classes:
3: (0)              `Figure`
4: (4)                  Top level `~matplotlib.artist.Artist`, which holds all plot elements.
5: (4)                  Many methods are implemented in `FigureBase`.
6: (0)              `SubFigure`
7: (4)                  A logical figure inside a figure, usually added to a figure (or parent
8: (4)                  `SubFigure`) with `Figure.add_subfigure` or `Figure.subfigures` methods
9: (4)                  (provisional API v3.4).
10: (0)             Figures are typically created using pyplot methods `~.pyplot.figure`,
11: (0)             `~.pyplot.subplots`, and `~.pyplot.subplot_mosaic`.
12: (0)             .. plot::
13: (4)                 :include-source:
14: (4)                 fig, ax = plt.subplots(figsize=(2, 2), facecolor='lightskyblue',
15: (27)                                        layout='constrained')
16: (4)                 fig.suptitle('Figure')
17: (4)                 ax.set_title('Axes', loc='left', fontstyle='oblique', fontsize='medium')
18: (0)             Some situations call for directly instantiating a `~.figure.Figure` class,
19: (0)             usually inside an application of some sort (see :ref:`user_interfaces` for a
20: (0)             list of examples) .  More information about Figures can be found at
21: (0)             :ref:`figure-intro`.
22: (0)             """
23: (0)             from contextlib import ExitStack
24: (0)             import inspect
25: (0)             import itertools
26: (0)             import logging
27: (0)             from numbers import Integral
28: (0)             import threading
29: (0)             import numpy as np
30: (0)             import matplotlib as mpl
31: (0)             from matplotlib import _blocking_input, backend_bases, _docstring, projections
32: (0)             from matplotlib.artist import (
33: (4)                 Artist, allow_rasterization, _finalize_rasterization)
34: (0)             from matplotlib.backend_bases import (
35: (4)                 DrawEvent, FigureCanvasBase, NonGuiException, MouseButton, _get_renderer)
36: (0)             import matplotlib._api as _api
37: (0)             import matplotlib.cbook as cbook
38: (0)             import matplotlib.colorbar as cbar
39: (0)             import matplotlib.image as mimage
40: (0)             from matplotlib.axes import Axes
41: (0)             from matplotlib.gridspec import GridSpec, SubplotParams
42: (0)             from matplotlib.layout_engine import (
43: (4)                 ConstrainedLayoutEngine, TightLayoutEngine, LayoutEngine,
44: (4)                 PlaceHolderLayoutEngine
45: (0)             )
46: (0)             import matplotlib.legend as mlegend
47: (0)             from matplotlib.patches import Rectangle
48: (0)             from matplotlib.text import Text
49: (0)             from matplotlib.transforms import (Affine2D, Bbox, BboxTransformTo,
50: (35)                                           TransformedBbox)
51: (0)             _log = logging.getLogger(__name__)
52: (0)             def _stale_figure_callback(self, val):
53: (4)                 if self.figure:
54: (8)                     self.figure.stale = val
55: (0)             class _AxesStack:
56: (4)                 """
57: (4)                 Helper class to track Axes in a figure.
58: (4)                 Axes are tracked both in the order in which they have been added
59: (4)                 (``self._axes`` insertion/iteration order) and in the separate "gca" stack
60: (4)                 (which is the index to which they map in the ``self._axes`` dict).
61: (4)                 """
62: (4)                 def __init__(self):
63: (8)                     self._axes = {}  # Mapping of Axes to "gca" order.
```

```
 64: (8)                     self._counter = itertools.count()
 65: (4)                 def as_list(self):
 66: (8)                     """List the Axes that have been added to the figure."""
 67: (8)                     return [*self._axes]  # This relies on dict preserving order.
 68: (4)                 def remove(self, a):
 69: (8)                     """Remove the Axes from the stack."""
 70: (8)                     self._axes.pop(a)
 71: (4)                 def bubble(self, a):
 72: (8)                     """Move an Axes, which must already exist in the stack, to the top."""
 73: (8)                     if a not in self._axes:
 74: (12)                        raise ValueError("Axes has not been added yet")
 75: (8)                     self._axes[a] = next(self._counter)
 76: (4)                 def add(self, a):
 77: (8)                     """Add an Axes to the stack, ignoring it if already present."""
 78: (8)                     if a not in self._axes:
 79: (12)                        self._axes[a] = next(self._counter)
 80: (4)                 def current(self):
 81: (8)                     """Return the active Axes, or None if the stack is empty."""
 82: (8)                     return max(self._axes, key=self._axes.__getitem__, default=None)
 83: (4)                 def __getstate__(self):
 84: (8)                     return {
 85: (12)                        **vars(self),
 86: (12)                        "_counter": max(self._axes.values(), default=0)
 87: (8)                     }
 88: (4)                 def __setstate__(self, state):
 89: (8)                     next_counter = state.pop('_counter')
 90: (8)                     vars(self).update(state)
 91: (8)                     self._counter = itertools.count(next_counter)
 92: (0)             class FigureBase(Artist):
 93: (4)                 """
 94: (4)                 Base class for `.Figure` and `.SubFigure` containing the methods that add
 95: (4)                 artists to the figure or subfigure, create Axes, etc.
 96: (4)                 """
 97: (4)                 def __init__(self, **kwargs):
 98: (8)                     super().__init__()
 99: (8)                     del self._axes
100: (8)                     self._suptitle = None
101: (8)                     self._supxlabel = None
102: (8)                     self._supylabel = None
103: (8)                     self._align_label_groups = {
104: (12)                        "x": cbook.Grouper(),
105: (12)                        "y": cbook.Grouper(),
106: (12)                        "title": cbook.Grouper()
107: (8)                     }
108: (8)                     self._localaxes = []  # track all Axes
109: (8)                     self.artists = []
110: (8)                     self.lines = []
111: (8)                     self.patches = []
112: (8)                     self.texts = []
113: (8)                     self.images = []
114: (8)                     self.legends = []
115: (8)                     self.subfigs = []
116: (8)                     self.stale = True
117: (8)                     self.suppressComposite = None
118: (8)                     self.set(**kwargs)
119: (4)                 def _get_draw_artists(self, renderer):
120: (8)                     """Also runs apply_aspect"""
121: (8)                     artists = self.get_children()
122: (8)                     artists.remove(self.patch)
123: (8)                     artists = sorted(
124: (12)                        (artist for artist in artists if not artist.get_animated()),
125: (12)                        key=lambda artist: artist.get_zorder())
126: (8)                     for ax in self._localaxes:
127: (12)                        locator = ax.get_axes_locator()
128: (12)                        ax.apply_aspect(locator(ax, renderer) if locator else None)
129: (12)                        for child in ax.get_children():
130: (16)                            if hasattr(child, 'apply_aspect'):
131: (20)                                locator = child.get_axes_locator()
132: (20)                                child.apply_aspect(
```

```
(24)                                locator(child, renderer) if locator else None)
(8)                     return artists
(4)                 def autofmt_xdate(
(12)                        self, bottom=0.2, rotation=30, ha='right', which='major'):
(8)                     """
(8)                     Date ticklabels often overlap, so it is useful to rotate them
(8)                     and right align them.  Also, a common use case is a number of
(8)                     subplots with shared x-axis where the x-axis is date data.  The
(8)                     ticklabels are often long, and it helps to rotate them on the
(8)                     bottom subplot and turn them off on other subplots, as well as
(8)                     turn off xlabels.
(8)                     Parameters
(8)                     ----------
(8)                     bottom : float, default: 0.2
(12)                        The bottom of the subplots for `subplots_adjust`.
(8)                     rotation : float, default: 30 degrees
(12)                        The rotation angle of the xtick labels in degrees.
(8)                     ha : {'left', 'center', 'right'}, default: 'right'
(12)                        The horizontal alignment of the xticklabels.
(8)                     which : {'major', 'minor', 'both'}, default: 'major'
(12)                        Selects which ticklabels to rotate.
(8)                     """
(8)                     _api.check_in_list(['major', 'minor', 'both'], which=which)
(8)                     allsubplots = all(ax.get_subplotspec() for ax in self.axes)
(8)                     if len(self.axes) == 1:
(12)                        for label in self.axes[0].get_xticklabels(which=which):
(16)                            label.set_ha(ha)
(16)                            label.set_rotation(rotation)
(8)                     else:
(12)                        if allsubplots:
(16)                            for ax in self.get_axes():
(20)                                if ax.get_subplotspec().is_last_row():
(24)                                    for label in ax.get_xticklabels(which=which):
(28)                                        label.set_ha(ha)
(28)                                        label.set_rotation(rotation)
(20)                                else:
(24)                                    for label in ax.get_xticklabels(which=which):
(28)                                        label.set_visible(False)
(24)                                    ax.set_xlabel('')
(8)                     if allsubplots:
(12)                        self.subplots_adjust(bottom=bottom)
(8)                     self.stale = True
(4)                 def get_children(self):
(8)                     """Get a list of artists contained in the figure."""
(8)                     return [self.patch,
(16)                            *self.artists,
(16)                            *self._localaxes,
(16)                            *self.lines,
(16)                            *self.patches,
(16)                            *self.texts,
(16)                            *self.images,
(16)                            *self.legends,
(16)                            *self.subfigs]
(4)                 def contains(self, mouseevent):
(8)                     """
(8)                     Test whether the mouse event occurred on the figure.
(8)                     Returns
(8)                     -------
(12)                        bool, {}
(8)                     """
(8)                     if self._different_canvas(mouseevent):
(12)                        return False, {}
(8)                     inside = self.bbox.contains(mouseevent.x, mouseevent.y)
(8)                     return inside, {}
(4)                 def get_window_extent(self, renderer=None):
(8)                     return self.bbox
(4)                 def _suplabels(self, t, info, **kwargs):
(8)                     """
(8)                     Add a centered %(name)s to the figure.
```

```
(8)                     Parameters
(8)                     ----------
(8)                     t : str
(12)                        The %(name)s text.
(8)                     x : float, default: %(x0)s
(12)                        The x location of the text in figure coordinates.
(8)                     y : float, default: %(y0)s
(12)                        The y location of the text in figure coordinates.
(8)                     horizontalalignment, ha : {'center', 'left', 'right'}, default: %(ha)s
(12)                        The horizontal alignment of the text relative to (*x*, *y*).
(8)                     verticalalignment, va : {'top', 'center', 'bottom', 'baseline'}, \
(0)             default: %(va)s
(12)                        The vertical alignment of the text relative to (*x*, *y*).
(8)                     fontsize, size : default: :rc:`figure.%(rc)ssize`
(12)                        The font size of the text. See `.Text.set_size` for possible
(12)                        values.
(8)                     fontweight, weight : default: :rc:`figure.%(rc)sweight`
(12)                        The font weight of the text. See `.Text.set_weight` for possible
(12)                        values.
(8)                     Returns
(8)                     -------
(8)                     text
(12)                        The `.Text` instance of the %(name)s.
(8)                     Other Parameters
(8)                     ----------------
(8)                     fontproperties : None or dict, optional
(12)                        A dict of font properties. If *fontproperties* is given the
(12)                        default values for font size and weight are taken from the
(12)                        `.FontProperties` defaults. :rc:`figure.%(rc)ssize` and
(12)                        :rc:`figure.%(rc)sweight` are ignored in this case.
(8)                     **kwargs
(12)                        Additional kwargs are `matplotlib.text.Text` properties.
(8)                     """
(8)                     x = kwargs.pop('x', None)
(8)                     y = kwargs.pop('y', None)
(8)                     if info['name'] in ['_supxlabel', '_suptitle']:
(12)                        autopos = y is None
(8)                     elif info['name'] == '_supylabel':
(12)                        autopos = x is None
(8)                     if x is None:
(12)                        x = info['x0']
(8)                     if y is None:
(12)                        y = info['y0']
(8)                     kwargs = cbook.normalize_kwargs(kwargs, Text)
(8)                     kwargs.setdefault('horizontalalignment', info['ha'])
(8)                     kwargs.setdefault('verticalalignment', info['va'])
(8)                     kwargs.setdefault('rotation', info['rotation'])
(8)                     if 'fontproperties' not in kwargs:
(12)                        kwargs.setdefault('fontsize', mpl.rcParams[info['size']])
(12)                        kwargs.setdefault('fontweight', mpl.rcParams[info['weight']])
(8)                     suplab = getattr(self, info['name'])
(8)                     if suplab is not None:
(12)                        suplab.set_text(t)
(12)                        suplab.set_position((x, y))
(12)                        suplab.set(**kwargs)
(8)                     else:
(12)                        suplab = self.text(x, y, t, **kwargs)
(12)                        setattr(self, info['name'], suplab)
(8)                     suplab._autopos = autopos
(8)                     self.stale = True
(8)                     return suplab
(4)                 @_docstring.Substitution(x0=0.5, y0=0.98, name='suptitle', ha='center',
(29)                                         va='top', rc='title')
(4)                 @_docstring.copy(_suplabels)
(4)                 def suptitle(self, t, **kwargs):
(8)                     info = {'name': '_suptitle', 'x0': 0.5, 'y0': 0.98,
(16)                            'ha': 'center', 'va': 'top', 'rotation': 0,
(16)                            'size': 'figure.titlesize', 'weight': 'figure.titleweight'}
(8)                     return self._suplabels(t, info, **kwargs)
```

```
    def get_suptitle(self):
        """Return the suptitle as string or an empty string if not set."""
        text_obj = self._suptitle
        return "" if text_obj is None else text_obj.get_text()
    @_docstring.Substitution(x0=0.5, y0=0.01, name='supxlabel', ha='center',
                             va='bottom', rc='label')
    @_docstring.copy(_suplabels)
    def supxlabel(self, t, **kwargs):
        info = {'name': '_supxlabel', 'x0': 0.5, 'y0': 0.01,
                'ha': 'center', 'va': 'bottom', 'rotation': 0,
                'size': 'figure.labelsize', 'weight': 'figure.labelweight'}
        return self._suplabels(t, info, **kwargs)
    def get_supxlabel(self):
        """Return the supxlabel as string or an empty string if not set."""
        text_obj = self._supxlabel
        return "" if text_obj is None else text_obj.get_text()
    @_docstring.Substitution(x0=0.02, y0=0.5, name='supylabel', ha='left',
                             va='center', rc='label')
    @_docstring.copy(_suplabels)
    def supylabel(self, t, **kwargs):
        info = {'name': '_supylabel', 'x0': 0.02, 'y0': 0.5,
                'ha': 'left', 'va': 'center', 'rotation': 'vertical',
                'rotation_mode': 'anchor', 'size': 'figure.labelsize',
                'weight': 'figure.labelweight'}
        return self._suplabels(t, info, **kwargs)
    def get_supylabel(self):
        """Return the supylabel as string or an empty string if not set."""
        text_obj = self._supylabel
        return "" if text_obj is None else text_obj.get_text()
    def get_edgecolor(self):
        """Get the edge color of the Figure rectangle."""
        return self.patch.get_edgecolor()
    def get_facecolor(self):
        """Get the face color of the Figure rectangle."""
        return self.patch.get_facecolor()
    def get_frameon(self):
        """
        Return the figure's background patch visibility, i.e.
        whether the figure background will be drawn. Equivalent to
        ``Figure.patch.get_visible()``.
        """
        return self.patch.get_visible()
    def set_linewidth(self, linewidth):
        """
        Set the line width of the Figure rectangle.
        Parameters
        ----------
        linewidth : number
        """
        self.patch.set_linewidth(linewidth)
    def get_linewidth(self):
        """
        Get the line width of the Figure rectangle.
        """
        return self.patch.get_linewidth()
    def set_edgecolor(self, color):
        """
        Set the edge color of the Figure rectangle.
        Parameters
        ----------
        color : :mpltype:`color`
        """
        self.patch.set_edgecolor(color)
    def set_facecolor(self, color):
        """
        Set the face color of the Figure rectangle.
        Parameters
        ----------
        color : :mpltype:`color`
```

```
(8)                 """
(8)                 self.patch.set_facecolor(color)
(4)             def set_frameon(self, b):
(8)                 """
(8)                 Set the figure's background patch visibility, i.e.
(8)                 whether the figure background will be drawn. Equivalent to
(8)                 ``Figure.patch.set_visible()``.
(8)                 Parameters
(8)                 ----------
(8)                 b : bool
(8)                 """
(8)                 self.patch.set_visible(b)
(8)                 self.stale = True
(4)             frameon = property(get_frameon, set_frameon)
(4)             def add_artist(self, artist, clip=False):
(8)                 """
(8)                 Add an `.Artist` to the figure.
(8)                 Usually artists are added to `~.axes.Axes` objects using
(8)                 `.Axes.add_artist`; this method can be used in the rare cases where
(8)                 one needs to add artists directly to the figure instead.
(8)                 Parameters
(8)                 ----------
(8)                 artist : `~matplotlib.artist.Artist`
(12)                    The artist to add to the figure. If the added artist has no
(12)                    transform previously set, its transform will be set to
(12)                    ``figure.transSubfigure``.
(8)                 clip : bool, default: False
(12)                    Whether the added artist should be clipped by the figure patch.
(8)                 Returns
(8)                 -------
(8)                 `~matplotlib.artist.Artist`
(12)                    The added artist.
(8)                 """
(8)                 artist.set_figure(self)
(8)                 self.artists.append(artist)
(8)                 artist._remove_method = self.artists.remove
(8)                 if not artist.is_transform_set():
(12)                    artist.set_transform(self.transSubfigure)
(8)                 if clip and artist.get_clip_path() is None:
(12)                    artist.set_clip_path(self.patch)
(8)                 self.stale = True
(8)                 return artist
(4)             @_docstring.dedent_interpd
(4)             def add_axes(self, *args, **kwargs):
(8)                 """
(8)                 Add an `~.axes.Axes` to the figure.
(8)                 Call signatures::
(12)                    add_axes(rect, projection=None, polar=False, **kwargs)
(12)                    add_axes(ax)
(8)                 Parameters
(8)                 ----------
(8)                 rect : tuple (left, bottom, width, height)
(12)                    The dimensions (left, bottom, width, height) of the new
(12)                    `~.axes.Axes`. All quantities are in fractions of figure width and
(12)                    height.
(8)                 projection : {None, 'aitoff', 'hammer', 'lambert', 'mollweide', \
(0)         'polar', 'rectilinear', str}, optional
(12)                    The projection type of the `~.axes.Axes`. *str* is the name of
(12)                    a custom projection, see `~matplotlib.projections`. The default
(12)                    None results in a 'rectilinear' projection.
(8)                 polar : bool, default: False
(12)                    If True, equivalent to projection='polar'.
(8)                 axes_class : subclass type of `~.axes.Axes`, optional
(12)                    The `.axes.Axes` subclass that is instantiated.  This parameter
(12)                    is incompatible with *projection* and *polar*.  See
(12)                    :ref:`axisartist_users-guide-index` for examples.
(8)                 sharex, sharey : `~matplotlib.axes.Axes`, optional
(12)                    Share the x or y `~matplotlib.axis` with sharex and/or sharey.
(12)                    The axis will have the same limits, ticks, and scale as the axis
```

```
(12)                        of the shared Axes.
(8)                     label : str
(12)                        A label for the returned Axes.
(8)                     Returns
(8)                     -------
(8)                     `~.axes.Axes`, or a subclass of `~.axes.Axes`
(12)                        The returned Axes class depends on the projection used. It is
(12)                        `~.axes.Axes` if rectilinear projection is used and
(12)                        `.projections.polar.PolarAxes` if polar projection is used.
(8)                     Other Parameters
(8)                     ----------------
(8)                     **kwargs
(12)                        This method also takes the keyword arguments for
(12)                        the returned Axes class. The keyword arguments for the
(12)                        rectilinear Axes class `~.axes.Axes` can be found in
(12)                        the following table but there might also be other keyword
(12)                        arguments if another projection is used, see the actual Axes
(12)                        class.
(12)                        %(Axes:kwdoc)s
(8)                     Notes
(8)                     -----
(8)                     In rare circumstances, `.add_axes` may be called with a single
(8)                     argument, an Axes instance already created in the present figure but
(8)                     not in the figure's list of Axes.
(8)                     See Also
(8)                     --------
(8)                     .Figure.add_subplot
(8)                     .pyplot.subplot
(8)                     .pyplot.axes
(8)                     .Figure.subplots
(8)                     .pyplot.subplots
(8)                     Examples
(8)                     --------
(8)                     Some simple examples::
(12)                        rect = l, b, w, h
(12)                        fig = plt.figure()
(12)                        fig.add_axes(rect)
(12)                        fig.add_axes(rect, frameon=False, facecolor='g')
(12)                        fig.add_axes(rect, polar=True)
(12)                        ax = fig.add_axes(rect, projection='polar')
(12)                        fig.delaxes(ax)
(12)                        fig.add_axes(ax)
(8)                     """
(8)                     if not len(args) and 'rect' not in kwargs:
(12)                        raise TypeError(
(16)                            "add_axes() missing 1 required positional argument: 'rect'")
(8)                     elif 'rect' in kwargs:
(12)                        if len(args):
(16)                            raise TypeError(
(20)                                "add_axes() got multiple values for argument 'rect'")
(12)                        args = (kwargs.pop('rect'), )
(8)                     if isinstance(args[0], Axes):
(12)                        a, *extra_args = args
(12)                        key = a._projection_init
(12)                        if a.get_figure() is not self:
(16)                            raise ValueError(
(20)                                "The Axes must have been created in the present figure")
(8)                     else:
(12)                        rect, *extra_args = args
(12)                        if not np.isfinite(rect).all():
(16)                            raise ValueError(f'all entries in rect must be finite not
{rect}')
(12)                        projection_class, pkw =
self._process_projection_requirements(**kwargs)
(12)                        a = projection_class(self, rect, **pkw)
(12)                        key = (projection_class, pkw)
(8)                     if extra_args:
(12)                        _api.warn_deprecated(
(16)                            "3.8",
```

```
476: (16)                           name="Passing more than one positional argument to
Figure.add_axes",
477: (16)                           addendum="Any additional positional arguments are currently
ignored.")
478: (8)                    return self._add_axes_internal(a, key)
479: (4)                @_docstring.dedent_interpd
480: (4)                def add_subplot(self, *args, **kwargs):
481: (8)                    """
482: (8)                    Add an `~.axes.Axes` to the figure as part of a subplot arrangement.
483: (8)                    Call signatures::
484: (11)                      add_subplot(nrows, ncols, index, **kwargs)
485: (11)                      add_subplot(pos, **kwargs)
486: (11)                      add_subplot(ax)
487: (11)                      add_subplot()
488: (8)                    Parameters
489: (8)                    ----------
490: (8)                    *args : int, (int, int, *index*), or `.SubplotSpec`, default: (1, 1,
1)
491: (12)                       The position of the subplot described by one of
492: (12)                       - Three integers (*nrows*, *ncols*, *index*). The subplot will
493: (14)                         take the *index* position on a grid with *nrows* rows and
494: (14)                         *ncols* columns. *index* starts at 1 in the upper left corner
495: (14)                         and increases to the right.  *index* can also be a two-tuple
496: (14)                         specifying the (*first*, *last*) indices (1-based, and including
497: (14)                         *last*) of the subplot, e.g., ``fig.add_subplot(3, 1, (1, 2))``
498: (14)                         makes a subplot that spans the upper 2/3 of the figure.
499: (12)                       - A 3-digit integer. The digits are interpreted as if given
500: (14)                         separately as three single-digit integers, i.e.
501: (14)                         ``fig.add_subplot(235)`` is the same as
502: (14)                         ``fig.add_subplot(2, 3, 5)``. Note that this can only be used
503: (14)                         if there are no more than 9 subplots.
504: (12)                       - A `.SubplotSpec`.
505: (12)                       In rare circumstances, `.add_subplot` may be called with a single
506: (12)                       argument, a subplot Axes instance already created in the
507: (12)                       present figure but not in the figure's list of Axes.
508: (8)                    projection : {None, 'aitoff', 'hammer', 'lambert', 'mollweide', \
509: (0)             'polar', 'rectilinear', str}, optional
510: (12)                       The projection type of the subplot (`~.axes.Axes`). *str* is the
511: (12)                       name of a custom projection, see `~matplotlib.projections`. The
512: (12)                       default None results in a 'rectilinear' projection.
513: (8)                    polar : bool, default: False
514: (12)                       If True, equivalent to projection='polar'.
515: (8)                    axes_class : subclass type of `~.axes.Axes`, optional
516: (12)                       The `.axes.Axes` subclass that is instantiated.  This parameter
517: (12)                       is incompatible with *projection* and *polar*.  See
518: (12)                       :ref:`axisartist_users-guide-index` for examples.
519: (8)                    sharex, sharey : `~matplotlib.axes.Axes`, optional
520: (12)                       Share the x or y `~matplotlib.axis` with sharex and/or sharey.
521: (12)                       The axis will have the same limits, ticks, and scale as the axis
522: (12)                       of the shared Axes.
523: (8)                    label : str
524: (12)                       A label for the returned Axes.
525: (8)                    Returns
526: (8)                    -------
527: (8)                    `~.axes.Axes`
528: (12)                       The Axes of the subplot. The returned Axes can actually be an
529: (12)                       instance of a subclass, such as `.projections.polar.PolarAxes` for
530: (12)                       polar projections.
531: (8)                    Other Parameters
532: (8)                    ----------------
533: (8)                    **kwargs
534: (12)                       This method also takes the keyword arguments for the returned Axes
535: (12)                       base class; except for the *figure* argument. The keyword
arguments
536: (12)                       for the rectilinear base class `~.axes.Axes` can be found in
537: (12)                       the following table but there might also be other keyword
538: (12)                       arguments if another projection is used.
539: (12)                       %(Axes:kwdoc)s
540: (8)                    See Also
```

```
(8)                     --------
(8)                     .Figure.add_axes
(8)                     .pyplot.subplot
(8)                     .pyplot.axes
(8)                     .Figure.subplots
(8)                     .pyplot.subplots
(8)                     Examples
(8)                     --------
(8)                     ::
(12)                        fig = plt.figure()
(12)                        fig.add_subplot(231)
(12)                        ax1 = fig.add_subplot(2, 3, 1)  # equivalent but more general
(12)                        fig.add_subplot(232, frameon=False)  # subplot with no frame
(12)                        fig.add_subplot(233, projection='polar')  # polar subplot
(12)                        fig.add_subplot(234, sharex=ax1)  # subplot sharing x-axis with
ax1
(12)                        fig.add_subplot(235, facecolor="red")  # red subplot
(12)                        ax1.remove()  # delete ax1 from the figure
(12)                        fig.add_subplot(ax1)  # add ax1 back to the figure
(8)                     """
(8)                     if 'figure' in kwargs:
(12)                        raise _api.kwarg_error("add_subplot", "figure")
(8)                     if (len(args) == 1
(16)                            and isinstance(args[0], mpl.axes._base._AxesBase)
(16)                            and args[0].get_subplotspec()):
(12)                        ax = args[0]
(12)                        key = ax._projection_init
(12)                        if ax.get_figure() is not self:
(16)                            raise ValueError("The Axes must have been created in "
(33)                                             "the present figure")
(8)                     else:
(12)                        if not args:
(16)                            args = (1, 1, 1)
(12)                        if (len(args) == 1 and isinstance(args[0], Integral)
(20)                                and 100 <= args[0] <= 999):
(16)                            args = tuple(map(int, str(args[0])))
(12)                        projection_class, pkw =
self._process_projection_requirements(**kwargs)
(12)                        ax = projection_class(self, *args, **pkw)
(12)                        key = (projection_class, pkw)
(8)                     return self._add_axes_internal(ax, key)
(4)                 def _add_axes_internal(self, ax, key):
(8)                     """Private helper for `add_axes` and `add_subplot`."""
(8)                     self._axstack.add(ax)
(8)                     if ax not in self._localaxes:
(12)                        self._localaxes.append(ax)
(8)                     self.sca(ax)
(8)                     ax._remove_method = self.delaxes
(8)                     ax._projection_init = key
(8)                     self.stale = True
(8)                     ax.stale_callback = _stale_figure_callback
(8)                     return ax
(4)                 def subplots(self, nrows=1, ncols=1, *, sharex=False, sharey=False,
(17)                         squeeze=True, width_ratios=None, height_ratios=None,
(17)                         subplot_kw=None, gridspec_kw=None):
(8)                     """
(8)                     Add a set of subplots to this figure.
(8)                     This utility wrapper makes it convenient to create common layouts of
(8)                     subplots in a single call.
(8)                     Parameters
(8)                     ----------
(8)                     nrows, ncols : int, default: 1
(12)                        Number of rows/columns of the subplot grid.
(8)                     sharex, sharey : bool or {'none', 'all', 'row', 'col'}, default: False
(12)                        Controls sharing of x-axis (*sharex*) or y-axis (*sharey*):
(12)                        - True or 'all': x- or y-axis will be shared among all subplots.
(12)                        - False or 'none': each subplot x- or y-axis will be independent.
(12)                        - 'row': each subplot row will share an x- or y-axis.
(12)                        - 'col': each subplot column will share an x- or y-axis.
```

```
(12)                        When subplots have a shared x-axis along a column, only the x tick
(12)                        labels of the bottom subplot are created. Similarly, when subplots
(12)                        have a shared y-axis along a row, only the y tick labels of the
(12)                        first column subplot are created. To later turn other subplots'
(12)                        ticklabels on, use `~matplotlib.axes.Axes.tick_params`.
(12)                        When subplots have a shared axis that has units, calling
(12)                        `.Axis.set_units` will update each axis with the new units.
(12)                        Note that it is not possible to unshare axes.
(8)                     squeeze : bool, default: True
(12)                        - If True, extra dimensions are squeezed out from the returned
(14)                          array of Axes:
(14)                          - if only one subplot is constructed (nrows=ncols=1), the
(16)                            resulting single Axes object is returned as a scalar.
(14)                          - for Nx1 or 1xM subplots, the returned object is a 1D numpy
(16)                            object array of Axes objects.
(14)                          - for NxM, subplots with N>1 and M>1 are returned as a 2D array.
(12)                        - If False, no squeezing at all is done: the returned Axes object
(14)                          is always a 2D array containing Axes instances, even if it ends
(14)                          up being 1x1.
(8)                     width_ratios : array-like of length *ncols*, optional
(12)                        Defines the relative widths of the columns. Each column gets a
(12)                        relative width of ``width_ratios[i] / sum(width_ratios)``.
(12)                        If not given, all columns will have the same width.  Equivalent
(12)                        to ``gridspec_kw={'width_ratios': [...]}``.
(8)                     height_ratios : array-like of length *nrows*, optional
(12)                        Defines the relative heights of the rows. Each row gets a
(12)                        relative height of ``height_ratios[i] / sum(height_ratios)``.
(12)                        If not given, all rows will have the same height. Equivalent
(12)                        to ``gridspec_kw={'height_ratios': [...]}``.
(8)                     subplot_kw : dict, optional
(12)                        Dict with keywords passed to the `.Figure.add_subplot` call used
to
(12)                        create each subplot.
(8)                     gridspec_kw : dict, optional
(12)                        Dict with keywords passed to the
(12)                        `~matplotlib.gridspec.GridSpec` constructor used to create
(12)                        the grid the subplots are placed on.
(8)                     Returns
(8)                     -------
(8)                     `~.axes.Axes` or array of Axes
(12)                        Either a single `~matplotlib.axes.Axes` object or an array of Axes
(12)                        objects if more than one subplot was created. The dimensions of
the
(12)                        resulting array can be controlled with the *squeeze* keyword, see
(12)                        above.
(8)                     See Also
(8)                     --------
(8)                     .pyplot.subplots
(8)                     .Figure.add_subplot
(8)                     .pyplot.subplot
(8)                     Examples
(8)                     --------
(8)                     ::
(12)                        x = np.linspace(0, 2*np.pi, 400)
(12)                        y = np.sin(x**2)
(12)                        fig = plt.figure()
(12)                        ax = fig.subplots()
(12)                        ax.plot(x, y)
(12)                        ax.set_title('Simple plot')
(12)                        ax1, ax2 = fig.subplots(1, 2, sharey=True)
(12)                        ax1.plot(x, y)
(12)                        ax1.set_title('Sharing Y axis')
(12)                        ax2.scatter(x, y)
(12)                        axes = fig.subplots(2, 2, subplot_kw=dict(projection='polar'))
(12)                        axes[0, 0].plot(x, y)
(12)                        axes[1, 1].scatter(x, y)
(12)                        fig.subplots(2, 2, sharex='col')
(12)                        fig.subplots(2, 2, sharey='row')
(12)                        fig.subplots(2, 2, sharex='all', sharey='all')
```

```
675: (12)                        fig.subplots(2, 2, sharex=True, sharey=True)
676: (8)                    """
677: (8)                    gridspec_kw = dict(gridspec_kw or {})
678: (8)                    if height_ratios is not None:
679: (12)                       if 'height_ratios' in gridspec_kw:
680: (16)                           raise ValueError("'height_ratios' must not be defined both as
"
681: (33)                                            "parameter and as key in 'gridspec_kw'")
682: (12)                       gridspec_kw['height_ratios'] = height_ratios
683: (8)                    if width_ratios is not None:
684: (12)                       if 'width_ratios' in gridspec_kw:
685: (16)                           raise ValueError("'width_ratios' must not be defined both as "
686: (33)                                            "parameter and as key in 'gridspec_kw'")
687: (12)                       gridspec_kw['width_ratios'] = width_ratios
688: (8)                    gs = self.add_gridspec(nrows, ncols, figure=self, **gridspec_kw)
689: (8)                    axs = gs.subplots(sharex=sharex, sharey=sharey, squeeze=squeeze,
690: (26)                                      subplot_kw=subplot_kw)
691: (8)                    return axs
692: (4)                def delaxes(self, ax):
693: (8)                    """
694: (8)                    Remove the `~.axes.Axes` *ax* from the figure; update the current
Axes.
695: (8)                    """
696: (8)                    self._remove_axes(ax, owners=[self._axstack, self._localaxes])
697: (4)                def _remove_axes(self, ax, owners):
698: (8)                    """
699: (8)                    Common helper for removal of standard Axes (via delaxes) and of child
Axes.
700: (8)                    Parameters
701: (8)                    ----------
702: (8)                    ax : `~.AxesBase`
703: (12)                       The Axes to remove.
704: (8)                    owners
705: (12)                       List of objects (list or _AxesStack) "owning" the Axes, from which
the Axes
706: (12)                       will be remove()d.
707: (8)                    """
708: (8)                    for owner in owners:
709: (12)                       owner.remove(ax)
710: (8)                    self._axobservers.process("_axes_change_event", self)
711: (8)                    self.stale = True
712: (8)                    self.canvas.release_mouse(ax)
713: (8)                    for name in ax._axis_names:  # Break link between any shared Axes
714: (12)                       grouper = ax._shared_axes[name]
715: (12)                       siblings = [other for other in grouper.get_siblings(ax) if other
is not ax]
716: (12)                       if not siblings:  # Axes was not shared along this axis; we're
done.
717: (16)                           continue
718: (12)                       grouper.remove(ax)
719: (12)                       remaining_axis = siblings[0]._axis_map[name]
720: (12)                       remaining_axis.get_major_formatter().set_axis(remaining_axis)
721: (12)                       remaining_axis.get_major_locator().set_axis(remaining_axis)
722: (12)                       remaining_axis.get_minor_formatter().set_axis(remaining_axis)
723: (12)                       remaining_axis.get_minor_locator().set_axis(remaining_axis)
724: (8)                    ax._twinned_axes.remove(ax)  # Break link between any twinned Axes.
725: (4)                def clear(self, keep_observers=False):
726: (8)                    """
727: (8)                    Clear the figure.
728: (8)                    Parameters
729: (8)                    ----------
730: (8)                    keep_observers : bool, default: False
731: (12)                       Set *keep_observers* to True if, for example,
732: (12)                       a gui widget is tracking the Axes in the figure.
733: (8)                    """
734: (8)                    self.suppressComposite = None
735: (8)                    for subfig in self.subfigs:
736: (12)                       subfig.clear(keep_observers=keep_observers)
737: (8)                    self.subfigs = []
```

```
(8)                     for ax in tuple(self.axes):  # Iterate over the copy.
(12)                        ax.clear()
(12)                        self.delaxes(ax)  # Remove ax from self._axstack.
(8)                     self.artists = []
(8)                     self.lines = []
(8)                     self.patches = []
(8)                     self.texts = []
(8)                     self.images = []
(8)                     self.legends = []
(8)                     if not keep_observers:
(12)                        self._axobservers = cbook.CallbackRegistry()
(8)                     self._suptitle = None
(8)                     self._supxlabel = None
(8)                     self._supylabel = None
(8)                     self.stale = True
(4)                 def clf(self, keep_observers=False):
(8)                     """
(8)                     [*Discouraged*] Alias for the `clear()` method.
(8)                     .. admonition:: Discouraged
(12)                        The use of ``clf()`` is discouraged. Use ``clear()`` instead.
(8)                     Parameters
(8)                     ----------
(8)                     keep_observers : bool, default: False
(12)                        Set *keep_observers* to True if, for example,
(12)                        a gui widget is tracking the Axes in the figure.
(8)                     """
(8)                     return self.clear(keep_observers=keep_observers)
(4)                 @_docstring.dedent_interpd
(4)                 def legend(self, *args, **kwargs):
(8)                     """
(8)                     Place a legend on the figure.
(8)                     Call signatures::
(12)                        legend()
(12)                        legend(handles, labels)
(12)                        legend(handles=handles)
(12)                        legend(labels)
(8)                     The call signatures correspond to the following different ways to use
(8)                     this method:
(8)                     **1. Automatic detection of elements to be shown in the legend**
(8)                     The elements to be added to the legend are automatically determined,
(8)                     when you do not pass in any extra arguments.
(8)                     In this case, the labels are taken from the artist. You can specify
(8)                     them either at artist creation or by calling the
(8)                     :meth:`~.Artist.set_label` method on the artist::
(12)                        ax.plot([1, 2, 3], label='Inline label')
(12)                        fig.legend()
(8)                     or::
(12)                        line, = ax.plot([1, 2, 3])
(12)                        line.set_label('Label via method')
(12)                        fig.legend()
(8)                     Specific lines can be excluded from the automatic legend element
(8)                     selection by defining a label starting with an underscore.
(8)                     This is default for all artists, so calling `.Figure.legend` without
(8)                     any arguments and without setting the labels manually will result in
(8)                     no legend being drawn.
(8)                     **2. Explicitly listing the artists and labels in the legend**
(8)                     For full control of which artists have a legend entry, it is possible
(8)                     to pass an iterable of legend artists followed by an iterable of
(8)                     legend labels respectively::
(12)                        fig.legend([line1, line2, line3], ['label1', 'label2', 'label3'])
(8)                     **3. Explicitly listing the artists in the legend**
(8)                     This is similar to 2, but the labels are taken from the artists'
(8)                     label properties. Example::
(12)                        line1, = ax1.plot([1, 2, 3], label='label1')
(12)                        line2, = ax2.plot([1, 2, 3], label='label2')
(12)                        fig.legend(handles=[line1, line2])
(8)                     **4. Labeling existing plot elements**
(8)                     .. admonition:: Discouraged
(12)                        This call signature is discouraged, because the relation between
```

```
(12)                        plot elements and labels is only implicit by their order and can
(12)                        easily be mixed up.
(8)                     To make a legend for all artists on all Axes, call this function with
(8)                     an iterable of strings, one for each legend item. For example::
(12)                        fig, (ax1, ax2) = plt.subplots(1, 2)
(12)                        ax1.plot([1, 3, 5], color='blue')
(12)                        ax2.plot([2, 4, 6], color='red')
(12)                        fig.legend(['the blues', 'the reds'])
(8)                     Parameters
(8)                     ----------
(8)                     handles : list of `.Artist`, optional
(12)                        A list of Artists (lines, patches) to be added to the legend.
(12)                        Use this together with *labels*, if you need full control on what
(12)                        is shown in the legend and the automatic mechanism described above
(12)                        is not sufficient.
(12)                        The length of handles and labels should be the same in this
(12)                        case. If they are not, they are truncated to the smaller length.
(8)                     labels : list of str, optional
(12)                        A list of labels to show next to the artists.
(12)                        Use this together with *handles*, if you need full control on what
(12)                        is shown in the legend and the automatic mechanism described above
(12)                        is not sufficient.
(8)                     Returns
(8)                     -------
(8)                     `~matplotlib.legend.Legend`
(8)                     Other Parameters
(8)                     ----------------
(8)                     %(_legend_kw_figure)s
(8)                     See Also
(8)                     --------
(8)                     .Axes.legend
(8)                     Notes
(8)                     -----
(8)                     Some artists are not supported by this function.  See
(8)                     :ref:`legend_guide` for details.
(8)                     """
(8)                     handles, labels, kwargs = mlegend._parse_legend_args(self.axes, *args,
**kwargs)
(8)                     kwargs.setdefault("bbox_transform", self.transSubfigure)
(8)                     l = mlegend.Legend(self, handles, labels, **kwargs)
(8)                     self.legends.append(l)
(8)                     l._remove_method = self.legends.remove
(8)                     self.stale = True
(8)                     return l
(4)                 @_docstring.dedent_interpd
(4)                 def text(self, x, y, s, fontdict=None, **kwargs):
(8)                     """
(8)                     Add text to figure.
(8)                     Parameters
(8)                     ----------
(8)                     x, y : float
(12)                        The position to place the text. By default, this is in figure
(12)                        coordinates, floats in [0, 1]. The coordinate system can be
changed
(12)                        using the *transform* keyword.
(8)                     s : str
(12)                        The text string.
(8)                     fontdict : dict, optional
(12)                        A dictionary to override the default text properties. If not
given,
(12)                        the defaults are determined by :rc:`font.*`. Properties passed as
(12)                        *kwargs* override the corresponding ones given in *fontdict*.
(8)                     Returns
(8)                     -------
(8)                     `~.text.Text`
(8)                     Other Parameters
(8)                     ----------------
(8)                     **kwargs : `~matplotlib.text.Text` properties
(12)                        Other miscellaneous text parameters.
```

```
873: (12)                        %(Text:kwdoc)s
874: (8)                     See Also
875: (8)                     --------
876: (8)                     .Axes.text
877: (8)                     .pyplot.text
878: (8)                     """
879: (8)                     effective_kwargs = {
880: (12)                        'transform': self.transSubfigure,
881: (12)                        **(fontdict if fontdict is not None else {}),
882: (12)                        **kwargs,
883: (8)                     }
884: (8)                     text = Text(x=x, y=y, text=s, **effective_kwargs)
885: (8)                     text.set_figure(self)
886: (8)                     text.stale_callback = _stale_figure_callback
887: (8)                     self.texts.append(text)
888: (8)                     text._remove_method = self.texts.remove
889: (8)                     self.stale = True
890: (8)                     return text
891: (4)                 @_docstring.dedent_interpd
892: (4)                 def colorbar(
893: (12)                        self, mappable, cax=None, ax=None, use_gridspec=True, **kwargs):
894: (8)                     """
895: (8)                     Add a colorbar to a plot.
896: (8)                     Parameters
897: (8)                     ----------
898: (8)                     mappable
899: (12)                        The `matplotlib.cm.ScalarMappable` (i.e., `.AxesImage`,
900: (12)                        `.ContourSet`, etc.) described by this colorbar.  This argument is
901: (12)                        mandatory for the `.Figure.colorbar` method but optional for the
902: (12)                        `.pyplot.colorbar` function, which sets the default to the current
903: (12)                        image.
904: (12)                        Note that one can create a `.ScalarMappable` "on-the-fly" to
905: (12)                        generate colorbars not attached to a previously drawn artist, e.g.
906: (12)                        ::
907: (16)                            fig.colorbar(cm.ScalarMappable(norm=norm, cmap=cmap), ax=ax)
908: (8)                     cax : `~matplotlib.axes.Axes`, optional
909: (12)                        Axes into which the colorbar will be drawn.  If `None`, then a new
910: (12)                        Axes is created and the space for it will be stolen from the
Axes(s)
911: (12)                        specified in *ax*.
912: (8)                     ax : `~matplotlib.axes.Axes` or iterable or `numpy.ndarray` of Axes,
optional
913: (12)                        The one or more parent Axes from which space for a new colorbar
Axes
914: (12)                        will be stolen. This parameter is only used if *cax* is not set.
915: (12)                        Defaults to the Axes that contains the mappable used to create the
916: (12)                        colorbar.
917: (8)                     use_gridspec : bool, optional
918: (12)                        If *cax* is ``None``, a new *cax* is created as an instance of
919: (12)                        Axes.  If *ax* is positioned with a subplotspec and *use_gridspec*
920: (12)                        is ``True``, then *cax* is also positioned with a subplotspec.
921: (8)                     Returns
922: (8)                     -------
923: (8)                     colorbar : `~matplotlib.colorbar.Colorbar`
924: (8)                     Other Parameters
925: (8)                     ----------------
926: (8)                     %(_make_axes_kw_doc)s
927: (8)                     %(_colormap_kw_doc)s
928: (8)                     Notes
929: (8)                     -----
930: (8)                     If *mappable* is a `~.contour.ContourSet`, its *extend* kwarg is
931: (8)                     included automatically.
932: (8)                     The *shrink* kwarg provides a simple way to scale the colorbar with
933: (8)                     respect to the Axes. Note that if *cax* is specified, it determines
the
934: (8)                     size of the colorbar, and *shrink* and *aspect* are ignored.
935: (8)                     For more precise control, you can manually specify the positions of
the
936: (8)                     axes objects in which the mappable and the colorbar are drawn.  In
```

```
this
937: (8)                    case, do not use any of the Axes properties kwargs.
938: (8)                    It is known that some vector graphics viewers (svg and pdf) render
939: (8)                    white gaps between segments of the colorbar.  This is due to bugs in
940: (8)                    the viewers, not Matplotlib.  As a workaround, the colorbar can be
941: (8)                    rendered with overlapping segments::
942: (12)                       cbar = colorbar()
943: (12)                       cbar.solids.set_edgecolor("face")
944: (12)                       draw()
945: (8)                    However, this has negative consequences in other circumstances, e.g.
946: (8)                    with semi-transparent images (alpha < 1) and colorbar extensions;
947: (8)                    therefore, this workaround is not used by default (see issue #1188).
948: (8)                    """
949: (8)                    if ax is None:
950: (12)                       ax = getattr(mappable, "axes", None)
951: (8)                    if cax is None:
952: (12)                       if ax is None:
953: (16)                           raise ValueError(
954: (20)                               'Unable to determine Axes to steal space for Colorbar. '
955: (20)                               'Either provide the *cax* argument to use as the Axes for
'
956: (20)                               'the Colorbar, provide the *ax* argument to steal space '
957: (20)                               'from it, or add *mappable* to an Axes.')
958: (12)                       fig = (  # Figure of first Axes; logic copied from make_axes.
959: (16)                           [*ax.flat] if isinstance(ax, np.ndarray)
960: (16)                           else [*ax] if np.iterable(ax)
961: (16)                           else [ax])[0].figure
962: (12)                       current_ax = fig.gca()
963: (12)                       if (fig.get_layout_engine() is not None and
964: (20)                               not fig.get_layout_engine().colorbar_gridspec):
965: (16)                           use_gridspec = False
966: (12)                       if (use_gridspec
967: (20)                               and isinstance(ax, mpl.axes._base._AxesBase)
968: (20)                               and ax.get_subplotspec()):
969: (16)                           cax, kwargs = cbar.make_axes_gridspec(ax, **kwargs)
970: (12)                       else:
971: (16)                           cax, kwargs = cbar.make_axes(ax, **kwargs)
972: (12)                       fig.sca(current_ax)
973: (12)                       cax.grid(visible=False, which='both', axis='both')
974: (8)                    if hasattr(mappable, "figure") and mappable.figure is not None:
975: (12)                       mappable_host_fig = mappable.figure
976: (12)                       if isinstance(mappable_host_fig, mpl.figure.SubFigure):
977: (16)                           mappable_host_fig = mappable_host_fig.figure
978: (12)                       if mappable_host_fig is not self.figure:
979: (16)                           _api.warn_external(
980: (24)                                   f'Adding colorbar to a different Figure '
981: (24)                                   f'{repr(mappable.figure)} than '
982: (24)                                   f'{repr(self.figure)} which '
983: (24)                                   f'fig.colorbar is called on.')
984: (8)                    NON_COLORBAR_KEYS = [  # remove kws that cannot be passed to Colorbar
985: (12)                       'fraction', 'pad', 'shrink', 'aspect', 'anchor', 'panchor']
986: (8)                    cb = cbar.Colorbar(cax, mappable, **{
987: (12)                       k: v for k, v in kwargs.items() if k not in NON_COLORBAR_KEYS})
988: (8)                    cax.figure.stale = True
989: (8)                    return cb
990: (4)                def subplots_adjust(self, left=None, bottom=None, right=None, top=None,
991: (24)                                   wspace=None, hspace=None):
992: (8)                    """
993: (8)                    Adjust the subplot layout parameters.
994: (8)                    Unset parameters are left unmodified; initial values are given by
995: (8)                    :rc:`figure.subplot.[name]`.
996: (8)                    Parameters
997: (8)                    ----------
998: (8)                    left : float, optional
999: (12)                       The position of the left edge of the subplots,
1000: (12)                      as a fraction of the figure width.
1001: (8)                   right : float, optional
1002: (12)                      The position of the right edge of the subplots,
1003: (12)                      as a fraction of the figure width.
```

```
(8)                 bottom : float, optional
(12)                    The position of the bottom edge of the subplots,
(12)                    as a fraction of the figure height.
(8)                 top : float, optional
(12)                    The position of the top edge of the subplots,
(12)                    as a fraction of the figure height.
(8)                 wspace : float, optional
(12)                    The width of the padding between subplots,
(12)                    as a fraction of the average Axes width.
(8)                 hspace : float, optional
(12)                    The height of the padding between subplots,
(12)                    as a fraction of the average Axes height.
(8)                 """
(8)                 if (self.get_layout_engine() is not None and
(16)                        not self.get_layout_engine().adjust_compatible):
(12)                    _api.warn_external(
(16)                        "This figure was using a layout engine that is "
(16)                        "incompatible with subplots_adjust and/or tight_layout; "
(16)                        "not calling subplots_adjust.")
(12)                    return
(8)                 self.subplotpars.update(left, bottom, right, top, wspace, hspace)
(8)                 for ax in self.axes:
(12)                    if ax.get_subplotspec() is not None:
(16)                        ax._set_position(ax.get_subplotspec().get_position(self))
(8)                 self.stale = True
(4)             def align_xlabels(self, axs=None):
(8)                 """
(8)                 Align the xlabels of subplots in the same subplot row if label
(8)                 alignment is being done automatically (i.e. the label position is
(8)                 not manually set).
(8)                 Alignment persists for draw events after this is called.
(8)                 If a label is on the bottom, it is aligned with labels on Axes that
(8)                 also have their label on the bottom and that have the same
(8)                 bottom-most subplot row.  If the label is on the top,
(8)                 it is aligned with labels on Axes with the same top-most row.
(8)                 Parameters
(8)                 ----------
(8)                 axs : list of `~matplotlib.axes.Axes`
(12)                    Optional list of (or `~numpy.ndarray`) `~matplotlib.axes.Axes`
(12)                    to align the xlabels.
(12)                    Default is to align all Axes on the figure.
(8)                 See Also
(8)                 --------
(8)                 matplotlib.figure.Figure.align_ylabels
(8)                 matplotlib.figure.Figure.align_titles
(8)                 matplotlib.figure.Figure.align_labels
(8)                 Notes
(8)                 -----
(8)                 This assumes that ``axs`` are from the same `.GridSpec`, so that
(8)                 their `.SubplotSpec` positions correspond to figure positions.
(8)                 Examples
(8)                 --------
(8)                 Example with rotated xtick labels::
(12)                    fig, axs = plt.subplots(1, 2)
(12)                    for tick in axs[0].get_xticklabels():
(16)                        tick.set_rotation(55)
(12)                    axs[0].set_xlabel('XLabel 0')
(12)                    axs[1].set_xlabel('XLabel 1')
(12)                    fig.align_xlabels()
(8)                 """
(8)                 if axs is None:
(12)                    axs = self.axes
(8)                 axs = [ax for ax in np.ravel(axs) if ax.get_subplotspec() is not None]
(8)                 for ax in axs:
(12)                    _log.debug(' Working on: %s', ax.get_xlabel())
(12)                    rowspan = ax.get_subplotspec().rowspan
(12)                    pos = ax.xaxis.get_label_position()  # top or bottom
(12)                    for axc in axs:
(16)                        if axc.xaxis.get_label_position() == pos:
```

```
(20)                                rowspanc = axc.get_subplotspec().rowspan
(20)                                if (pos == 'top' and rowspan.start == rowspanc.start or
(28)                                        pos == 'bottom' and rowspan.stop ==
rowspanc.stop):
(24)                                    self._align_label_groups['x'].join(ax, axc)
(4)                 def align_ylabels(self, axs=None):
(8)                     """
(8)                     Align the ylabels of subplots in the same subplot column if label
(8)                     alignment is being done automatically (i.e. the label position is
(8)                     not manually set).
(8)                     Alignment persists for draw events after this is called.
(8)                     If a label is on the left, it is aligned with labels on Axes that
(8)                     also have their label on the left and that have the same
(8)                     left-most subplot column.  If the label is on the right,
(8)                     it is aligned with labels on Axes with the same right-most column.
(8)                     Parameters
(8)                     ----------
(8)                     axs : list of `~matplotlib.axes.Axes`
(12)                        Optional list (or `~numpy.ndarray`) of `~matplotlib.axes.Axes`
(12)                        to align the ylabels.
(12)                        Default is to align all Axes on the figure.
(8)                     See Also
(8)                     --------
(8)                     matplotlib.figure.Figure.align_xlabels
(8)                     matplotlib.figure.Figure.align_titles
(8)                     matplotlib.figure.Figure.align_labels
(8)                     Notes
(8)                     -----
(8)                     This assumes that ``axs`` are from the same `.GridSpec`, so that
(8)                     their `.SubplotSpec` positions correspond to figure positions.
(8)                     Examples
(8)                     --------
(8)                     Example with large yticks labels::
(12)                        fig, axs = plt.subplots(2, 1)
(12)                        axs[0].plot(np.arange(0, 1000, 50))
(12)                        axs[0].set_ylabel('YLabel 0')
(12)                        axs[1].set_ylabel('YLabel 1')
(12)                        fig.align_ylabels()
(8)                     """
(8)                     if axs is None:
(12)                        axs = self.axes
(8)                     axs = [ax for ax in np.ravel(axs) if ax.get_subplotspec() is not None]
(8)                     for ax in axs:
(12)                        _log.debug(' Working on: %s', ax.get_ylabel())
(12)                        colspan = ax.get_subplotspec().colspan
(12)                        pos = ax.yaxis.get_label_position()  # left or right
(12)                        for axc in axs:
(16)                            if axc.yaxis.get_label_position() == pos:
(20)                                colspanc = axc.get_subplotspec().colspan
(20)                                if (pos == 'left' and colspan.start == colspanc.start or
(28)                                        pos == 'right' and colspan.stop == colspanc.stop):
(24)                                    self._align_label_groups['y'].join(ax, axc)
(4)                 def align_titles(self, axs=None):
(8)                     """
(8)                     Align the titles of subplots in the same subplot row if title
(8)                     alignment is being done automatically (i.e. the title position is
(8)                     not manually set).
(8)                     Alignment persists for draw events after this is called.
(8)                     Parameters
(8)                     ----------
(8)                     axs : list of `~matplotlib.axes.Axes`
(12)                        Optional list of (or ndarray) `~matplotlib.axes.Axes`
(12)                        to align the titles.
(12)                        Default is to align all Axes on the figure.
(8)                     See Also
(8)                     --------
(8)                     matplotlib.figure.Figure.align_xlabels
(8)                     matplotlib.figure.Figure.align_ylabels
(8)                     matplotlib.figure.Figure.align_labels
```

```
(8)                 Notes
(8)                 -----
(8)                 This assumes that ``axs`` are from the same `.GridSpec`, so that
(8)                 their `.SubplotSpec` positions correspond to figure positions.
(8)                 Examples
(8)                 --------
(8)                 Example with titles::
(12)                    fig, axs = plt.subplots(1, 2)
(12)                    axs[0].set_aspect('equal')
(12)                    axs[0].set_title('Title 0')
(12)                    axs[1].set_title('Title 1')
(12)                    fig.align_titles()
(8)                 """
(8)                 if axs is None:
(12)                    axs = self.axes
(8)                 axs = [ax for ax in np.ravel(axs) if ax.get_subplotspec() is not None]
(8)                 for ax in axs:
(12)                    _log.debug(' Working on: %s', ax.get_title())
(12)                    rowspan = ax.get_subplotspec().rowspan
(12)                    for axc in axs:
(16)                        rowspanc = axc.get_subplotspec().rowspan
(16)                        if (rowspan.start == rowspanc.start):
(20)                            self._align_label_groups['title'].join(ax, axc)
(4)             def align_labels(self, axs=None):
(8)                 """
(8)                 Align the xlabels and ylabels of subplots with the same subplots
(8)                 row or column (respectively) if label alignment is being
(8)                 done automatically (i.e. the label position is not manually set).
(8)                 Alignment persists for draw events after this is called.
(8)                 Parameters
(8)                 ----------
(8)                 axs : list of `~matplotlib.axes.Axes`
(12)                    Optional list (or `~numpy.ndarray`) of `~matplotlib.axes.Axes`
(12)                    to align the labels.
(12)                    Default is to align all Axes on the figure.
(8)                 See Also
(8)                 --------
(8)                 matplotlib.figure.Figure.align_xlabels
(8)                 matplotlib.figure.Figure.align_ylabels
(8)                 matplotlib.figure.Figure.align_titles
(8)                 """
(8)                 self.align_xlabels(axs=axs)
(8)                 self.align_ylabels(axs=axs)
(4)             def add_gridspec(self, nrows=1, ncols=1, **kwargs):
(8)                 """
(8)                 Low-level API for creating a `.GridSpec` that has this figure as a
parent.
(8)                 This is a low-level API, allowing you to create a gridspec and
(8)                 subsequently add subplots based on the gridspec. Most users do
(8)                 not need that freedom and should use the higher-level methods
(8)                 `~.Figure.subplots` or `~.Figure.subplot_mosaic`.
(8)                 Parameters
(8)                 ----------
(8)                 nrows : int, default: 1
(12)                    Number of rows in grid.
(8)                 ncols : int, default: 1
(12)                    Number of columns in grid.
(8)                 Returns
(8)                 -------
(8)                 `.GridSpec`
(8)                 Other Parameters
(8)                 ----------------
(8)                 **kwargs
(12)                    Keyword arguments are passed to `.GridSpec`.
(8)                 See Also
(8)                 --------
(8)                 matplotlib.pyplot.subplots
(8)                 Examples
(8)                 --------
```

```
(8)                 Adding a subplot that spans two rows::
(12)                    fig = plt.figure()
(12)                    gs = fig.add_gridspec(2, 2)
(12)                    ax1 = fig.add_subplot(gs[0, 0])
(12)                    ax2 = fig.add_subplot(gs[1, 0])
(12)                    ax3 = fig.add_subplot(gs[:, 1])
(8)                 """
(8)                 _ = kwargs.pop('figure', None)  # pop in case user has added this...
(8)                 gs = GridSpec(nrows=nrows, ncols=ncols, figure=self, **kwargs)
(8)                 return gs
(4)             def subfigures(self, nrows=1, ncols=1, squeeze=True,
(19)                           wspace=None, hspace=None,
(19)                           width_ratios=None, height_ratios=None,
(19)                           **kwargs):
(8)                 """
(8)                 Add a set of subfigures to this figure or subfigure.
(8)                 A subfigure has the same artist methods as a figure, and is logically
(8)                 the same as a figure, but cannot print itself.
(8)                 See :doc:`/gallery/subplots_axes_and_figures/subfigures`.
(8)                 .. note::
(12)                    The *subfigure* concept is new in v3.4, and the API is still
provisional.
(8)                 Parameters
(8)                 ----------
(8)                 nrows, ncols : int, default: 1
(12)                    Number of rows/columns of the subfigure grid.
(8)                 squeeze : bool, default: True
(12)                    If True, extra dimensions are squeezed out from the returned
(12)                    array of subfigures.
(8)                 wspace, hspace : float, default: None
(12)                    The amount of width/height reserved for space between subfigures,
(12)                    expressed as a fraction of the average subfigure width/height.
(12)                    If not given, the values will be inferred from rcParams if using
(12)                    constrained layout (see `~.ConstrainedLayoutEngine`), or zero if
(12)                    not using a layout engine.
(8)                 width_ratios : array-like of length *ncols*, optional
(12)                    Defines the relative widths of the columns. Each column gets a
(12)                    relative width of ``width_ratios[i] / sum(width_ratios)``.
(12)                    If not given, all columns will have the same width.
(8)                 height_ratios : array-like of length *nrows*, optional
(12)                    Defines the relative heights of the rows. Each row gets a
(12)                    relative height of ``height_ratios[i] / sum(height_ratios)``.
(12)                    If not given, all rows will have the same height.
(8)                 """
(8)                 gs = GridSpec(nrows=nrows, ncols=ncols, figure=self,
(22)                              wspace=wspace, hspace=hspace,
(22)                              width_ratios=width_ratios,
(22)                              height_ratios=height_ratios,
(22)                              left=0, right=1, bottom=0, top=1)
(8)                 sfarr = np.empty((nrows, ncols), dtype=object)
(8)                 for i in range(ncols):
(12)                    for j in range(nrows):
(16)                        sfarr[j, i] = self.add_subfigure(gs[j, i], **kwargs)
(8)                 if self.get_layout_engine() is None and (wspace is not None or
(49)                                                         hspace is not None):
(12)                    bottoms, tops, lefts, rights = gs.get_grid_positions(self)
(12)                    for sfrow, bottom, top in zip(sfarr, bottoms, tops):
(16)                        for sf, left, right in zip(sfrow, lefts, rights):
(20)                            bbox = Bbox.from_extents(left, bottom, right, top)
(20)                            sf._redo_transform_rel_fig(bbox=bbox)
(8)                 if squeeze:
(12)                    return sfarr.item() if sfarr.size == 1 else sfarr.squeeze()
(8)                 else:
(12)                    return sfarr
(4)             def add_subfigure(self, subplotspec, **kwargs):
(8)                 """
(8)                 Add a `.SubFigure` to the figure as part of a subplot arrangement.
(8)                 Parameters
(8)                 ----------
```

```
(8)                 subplotspec : `.gridspec.SubplotSpec`
(12)                    Defines the region in a parent gridspec where the subfigure will
(12)                    be placed.
(8)                 Returns
(8)                 -------
(8)                 `.SubFigure`
(8)                 Other Parameters
(8)                 ----------------
(8)                 **kwargs
(12)                    Are passed to the `.SubFigure` object.
(8)                 See Also
(8)                 --------
(8)                 .Figure.subfigures
(8)                 """
(8)                 sf = SubFigure(self, subplotspec, **kwargs)
(8)                 self.subfigs += [sf]
(8)                 sf._remove_method = self.subfigs.remove
(8)                 sf.stale_callback = _stale_figure_callback
(8)                 self.stale = True
(8)                 return sf
(4)             def sca(self, a):
(8)                 """Set the current Axes to be *a* and return *a*."""
(8)                 self._axstack.bubble(a)
(8)                 self._axobservers.process("_axes_change_event", self)
(8)                 return a
(4)             def gca(self):
(8)                 """
(8)                 Get the current Axes.
(8)                 If there is currently no Axes on this Figure, a new one is created
(8)                 using `.Figure.add_subplot`.  (To test whether there is currently an
(8)                 Axes on a Figure, check whether ``figure.axes`` is empty.  To test
(8)                 whether there is currently a Figure on the pyplot figure stack, check
(8)                 whether `.pyplot.get_fignums()` is empty.)
(8)                 """
(8)                 ax = self._axstack.current()
(8)                 return ax if ax is not None else self.add_subplot()
(4)             def _gci(self):
(8)                 """
(8)                 Get the current colorable artist.
(8)                 Specifically, returns the current `.ScalarMappable` instance (`.Image`
(8)                 created by `imshow` or `figimage`, `.Collection` created by `pcolor`
or
(8)                 `scatter`, etc.), or *None* if no such instance has been defined.
(8)                 The current image is an attribute of the current Axes, or the nearest
(8)                 earlier Axes in the current figure that contains an image.
(8)                 Notes
(8)                 -----
(8)                 Historically, the only colorable artists were images; hence the name
(8)                 ``gci`` (get current image).
(8)                 """
(8)                 ax = self._axstack.current()
(8)                 if ax is None:
(12)                    return None
(8)                 im = ax._gci()
(8)                 if im is not None:
(12)                    return im
(8)                 for ax in reversed(self.axes):
(12)                    im = ax._gci()
(12)                    if im is not None:
(16)                        return im
(8)                 return None
(4)             def _process_projection_requirements(self, *, axes_class=None,
polar=False,
(41)                                                 projection=None, **kwargs):
(8)                 """
(8)                 Handle the args/kwargs to add_axes/add_subplot/gca, returning::
(12)                    (axes_proj_class, proj_class_kwargs)
(8)                 which can be used for new Axes initialization/identification.
(8)                 """
```

```
(8)                     if axes_class is not None:
(12)                        if polar or projection is not None:
(16)                            raise ValueError(
(20)                                "Cannot combine 'axes_class' and 'projection' or 'polar'")
(12)                        projection_class = axes_class
(8)                     else:
(12)                        if polar:
(16)                            if projection is not None and projection != 'polar':
(20)                                raise ValueError(
(24)                                    f"polar={polar}, yet projection={projection!r}. "
(24)                                    "Only one of these arguments should be supplied."
(20)                                )
(16)                            projection = 'polar'
(12)                        if isinstance(projection, str) or projection is None:
(16)                            projection_class =
projections.get_projection_class(projection)
(12)                        elif hasattr(projection, '_as_mpl_axes'):
(16)                            projection_class, extra_kwargs = projection._as_mpl_axes()
(16)                            kwargs.update(**extra_kwargs)
(12)                        else:
(16)                            raise TypeError(
(20)                                f"projection must be a string, None or implement a "
(20)                                f"_as_mpl_axes method, not {projection!r}")
(8)                     return projection_class, kwargs
(4)                 def get_default_bbox_extra_artists(self):
(8)                     """
(8)                     Return a list of Artists typically used in `.Figure.get_tightbbox`.
(8)                     """
(8)                     bbox_artists = [artist for artist in self.get_children()
(24)                                    if (artist.get_visible() and artist.get_in_layout())]
(8)                     for ax in self.axes:
(12)                        if ax.get_visible():
(16)                            bbox_artists.extend(ax.get_default_bbox_extra_artists())
(8)                     return bbox_artists
(4)                 @_api.make_keyword_only("3.8", "bbox_extra_artists")
(4)                 def get_tightbbox(self, renderer=None, bbox_extra_artists=None):
(8)                     """
(8)                     Return a (tight) bounding box of the figure *in inches*.
(8)                     Note that `.FigureBase` differs from all other artists, which return
(8)                     their `.Bbox` in pixels.
(8)                     Artists that have ``artist.set_in_layout(False)`` are not included
(8)                     in the bbox.
(8)                     Parameters
(8)                     ----------
(8)                     renderer : `.RendererBase` subclass
(12)                        Renderer that will be used to draw the figures (i.e.
(12)                        ``fig.canvas.get_renderer()``)
(8)                     bbox_extra_artists : list of `.Artist` or ``None``
(12)                        List of artists to include in the tight bounding box.  If
(12)                        ``None`` (default), then all artist children of each Axes are
(12)                        included in the tight bounding box.
(8)                     Returns
(8)                     -------
(8)                     `.BboxBase`
(12)                        containing the bounding box (in figure inches).
(8)                     """
(8)                     if renderer is None:
(12)                        renderer = self.figure._get_renderer()
(8)                     bb = []
(8)                     if bbox_extra_artists is None:
(12)                        artists = [artist for artist in self.get_children()
(23)                                   if (artist not in self.axes and artist.get_visible()
(27)                                       and artist.get_in_layout())]
(8)                     else:
(12)                        artists = bbox_extra_artists
(8)                     for a in artists:
(12)                        bbox = a.get_tightbbox(renderer)
(12)                        if bbox is not None:
(16)                            bb.append(bbox)
```

```
(8)                     for ax in self.axes:
(12)                        if ax.get_visible():
(16)                            try:
(20)                                bbox = ax.get_tightbbox(
(24)                                    renderer, bbox_extra_artists=bbox_extra_artists)
(16)                            except TypeError:
(20)                                bbox = ax.get_tightbbox(renderer)
(16)                            bb.append(bbox)
(8)                     bb = [b for b in bb
(14)                          if (np.isfinite(b.width) and np.isfinite(b.height)
(18)                              and (b.width != 0 or b.height != 0))]
(8)                     isfigure = hasattr(self, 'bbox_inches')
(8)                     if len(bb) == 0:
(12)                        if isfigure:
(16)                            return self.bbox_inches
(12)                        else:
(16)                            bb = [self.bbox]
(8)                     _bbox = Bbox.union(bb)
(8)                     if isfigure:
(12)                        _bbox = TransformedBbox(_bbox, self.dpi_scale_trans.inverted())
(8)                     return _bbox
(4)                 @staticmethod
(4)                 def _norm_per_subplot_kw(per_subplot_kw):
(8)                     expanded = {}
(8)                     for k, v in per_subplot_kw.items():
(12)                        if isinstance(k, tuple):
(16)                            for sub_key in k:
(20)                                if sub_key in expanded:
(24)                                    raise ValueError(f'The key {sub_key!r} appears
multiple times.')
(20)                                expanded[sub_key] = v
(12)                        else:
(16)                            if k in expanded:
(20)                                raise ValueError(f'The key {k!r} appears multiple times.')
(16)                            expanded[k] = v
(8)                     return expanded
(4)                 @staticmethod
(4)                 def _normalize_grid_string(layout):
(8)                     if '\n' not in layout:
(12)                        return [list(ln) for ln in layout.split(';')]
(8)                     else:
(12)                        layout = inspect.cleandoc(layout)
(12)                        return [list(ln) for ln in layout.strip('\n').split('\n')]
(4)                 def subplot_mosaic(self, mosaic, *, sharex=False, sharey=False,
(23)                                   width_ratios=None, height_ratios=None,
(23)                                   empty_sentinel='.',
(23)                                   subplot_kw=None, per_subplot_kw=None,
gridspec_kw=None):
(8)                     """
(8)                     Build a layout of Axes based on ASCII art or nested lists.
(8)                     This is a helper function to build complex GridSpec layouts visually.
(8)                     See :ref:`mosaic`
(8)                     for an example and full API documentation
(8)                     Parameters
(8)                     ----------
(8)                     mosaic : list of list of {hashable or nested} or str
(12)                        A visual layout of how you want your Axes to be arranged
(12)                        labeled as strings.  For example ::
(15)                           x = [['A panel', 'A panel', 'edge'],
(20)                                ['C panel', '.',       'edge']]
(12)                        produces 4 Axes:
(12)                        - 'A panel' which is 1 row high and spans the first two columns
(12)                        - 'edge' which is 2 rows high and is on the right edge
(12)                        - 'C panel' which in 1 row and 1 column wide in the bottom left
(12)                        - a blank space 1 row and 1 column wide in the bottom center
(12)                        Any of the entries in the layout can be a list of lists
(12)                        of the same form to create nested layouts.
(12)                        If input is a str, then it can either be a multi-line string of
(12)                        the form ::
```

```
1479: (14)                        '''
1480: (14)                        AAE
1481: (14)                        C.E
1482: (14)                        '''
1483: (12)                    where each character is a column and each line is a row. Or it
1484: (12)                    can be a single-line string where rows are separated by ``;``::
1485: (14)                        'AB;CC'
1486: (12)                    The string notation allows only single character Axes labels and
1487: (12)                    does not support nesting but is very terse.
1488: (12)                    The Axes identifiers may be `str` or a non-iterable hashable
1489: (12)                    object (e.g. `tuple` s may not be used).
1490: (8)                 sharex, sharey : bool, default: False
1491: (12)                    If True, the x-axis (*sharex*) or y-axis (*sharey*) will be shared
1492: (12)                    among all subplots.  In that case, tick label visibility and axis
1493: (12)                    units behave as for `subplots`.  If False, each subplot's x- or
1494: (12)                    y-axis will be independent.
1495: (8)                 width_ratios : array-like of length *ncols*, optional
1496: (12)                    Defines the relative widths of the columns. Each column gets a
1497: (12)                    relative width of ``width_ratios[i] / sum(width_ratios)``.
1498: (12)                    If not given, all columns will have the same width.  Equivalent
1499: (12)                    to ``gridspec_kw={'width_ratios': [...]}``. In the case of nested
1500: (12)                    layouts, this argument applies only to the outer layout.
1501: (8)                 height_ratios : array-like of length *nrows*, optional
1502: (12)                    Defines the relative heights of the rows. Each row gets a
1503: (12)                    relative height of ``height_ratios[i] / sum(height_ratios)``.
1504: (12)                    If not given, all rows will have the same height. Equivalent
1505: (12)                    to ``gridspec_kw={'height_ratios': [...]}``. In the case of nested
1506: (12)                    layouts, this argument applies only to the outer layout.
1507: (8)                 subplot_kw : dict, optional
1508: (12)                    Dictionary with keywords passed to the `.Figure.add_subplot` call
1509: (12)                    used to create each subplot.  These values may be overridden by
1510: (12)                    values in *per_subplot_kw*.
1511: (8)                 per_subplot_kw : dict, optional
1512: (12)                    A dictionary mapping the Axes identifiers or tuples of identifiers
1513: (12)                    to a dictionary of keyword arguments to be passed to the
1514: (12)                    `.Figure.add_subplot` call used to create each subplot.  The
values
1515: (12)                    in these dictionaries have precedence over the values in
1516: (12)                    *subplot_kw*.
1517: (12)                    If *mosaic* is a string, and thus all keys are single characters,
1518: (12)                    it is possible to use a single string instead of a tuple as keys;
1519: (12)                    i.e. ``"AB"`` is equivalent to ``("A", "B")``.
1520: (12)                    .. versionadded:: 3.7
1521: (8)                 gridspec_kw : dict, optional
1522: (12)                    Dictionary with keywords passed to the `.GridSpec` constructor
used
1523: (12)                    to create the grid the subplots are placed on. In the case of
1524: (12)                    nested layouts, this argument applies only to the outer layout.
1525: (12)                    For more complex layouts, users should use `.Figure.subfigures`
1526: (12)                    to create the nesting.
1527: (8)                 empty_sentinel : object, optional
1528: (12)                    Entry in the layout to mean "leave this space empty".  Defaults
1529: (12)                    to ``'.'``. Note, if *layout* is a string, it is processed via
1530: (12)                    `inspect.cleandoc` to remove leading white space, which may
1531: (12)                    interfere with using white-space as the empty sentinel.
1532: (8)                 Returns
1533: (8)                 -------
1534: (8)                 dict[label, Axes]
1535: (11)                   A dictionary mapping the labels to the Axes objects.  The order of
1536: (11)                   the Axes is left-to-right and top-to-bottom of their position in
the
1537: (11)                   total layout.
1538: (8)                 """
1539: (8)                 subplot_kw = subplot_kw or {}
1540: (8)                 gridspec_kw = dict(gridspec_kw or {})
1541: (8)                 per_subplot_kw = per_subplot_kw or {}
1542: (8)                 if height_ratios is not None:
1543: (12)                    if 'height_ratios' in gridspec_kw:
1544: (16)                        raise ValueError("'height_ratios' must not be defined both as
```

```
"
(33)                                "parameter and as key in 'gridspec_kw'")
(12)                    gridspec_kw['height_ratios'] = height_ratios
(8)                 if width_ratios is not None:
(12)                    if 'width_ratios' in gridspec_kw:
(16)                        raise ValueError("'width_ratios' must not be defined both as "
(33)                                "parameter and as key in 'gridspec_kw'")
(12)                    gridspec_kw['width_ratios'] = width_ratios
(8)                 if isinstance(mosaic, str):
(12)                    mosaic = self._normalize_grid_string(mosaic)
(12)                    per_subplot_kw = {
(16)                        tuple(k): v for k, v in per_subplot_kw.items()
(12)                    }
(8)                 per_subplot_kw = self._norm_per_subplot_kw(per_subplot_kw)
(8)                 _api.check_isinstance(bool, sharex=sharex, sharey=sharey)
(8)                 def _make_array(inp):
(12)                    """
(12)                    Convert input into 2D array
(12)                    We need to have this internal function rather than
(12)                    ``np.asarray(..., dtype=object)`` so that a list of lists
(12)                    of lists does not get converted to an array of dimension > 2.
(12)                    Returns
(12)                    -------
(12)                    2D object array
(12)                    """
(12)                    r0, *rest = inp
(12)                    if isinstance(r0, str):
(16)                        raise ValueError('List mosaic specification must be 2D')
(12)                    for j, r in enumerate(rest, start=1):
(16)                        if isinstance(r, str):
(20)                            raise ValueError('List mosaic specification must be 2D')
(16)                        if len(r0) != len(r):
(20)                            raise ValueError(
(24)                                "All of the rows must be the same length, however "
(24)                                f"the first row ({r0!r}) has length {len(r0)} "
(24)                                f"and row {j} ({r!r}) has length {len(r)}."
(20)                            )
(12)                    out = np.zeros((len(inp), len(r0)), dtype=object)
(12)                    for j, r in enumerate(inp):
(16)                        for k, v in enumerate(r):
(20)                            out[j, k] = v
(12)                    return out
(8)                 def _identify_keys_and_nested(mosaic):
(12)                    """
(12)                    Given a 2D object array, identify unique IDs and nested mosaics
(12)                    Parameters
(12)                    ----------
(12)                    mosaic : 2D object array
(12)                    Returns
(12)                    -------
(12)                    unique_ids : tuple
(16)                        The unique non-sub mosaic entries in this mosaic
(12)                    nested : dict[tuple[int, int], 2D object array]
(12)                    """
(12)                    unique_ids = cbook._OrderedSet()
(12)                    nested = {}
(12)                    for j, row in enumerate(mosaic):
(16)                        for k, v in enumerate(row):
(20)                            if v == empty_sentinel:
(24)                                continue
(20)                            elif not cbook.is_scalar_or_string(v):
(24)                                nested[(j, k)] = _make_array(v)
(20)                            else:
(24)                                unique_ids.add(v)
(12)                    return tuple(unique_ids), nested
(8)                 def _do_layout(gs, mosaic, unique_ids, nested):
(12)                    """
(12)                    Recursively do the mosaic.
(12)                    Parameters
```

```
(12)                        ----------
(12)                        gs : GridSpec
(12)                        mosaic : 2D object array
(16)                            The input converted to a 2D array for this level.
(12)                        unique_ids : tuple
(16)                            The identified scalar labels at this level of nesting.
(12)                        nested : dict[tuple[int, int]], 2D object array
(16)                            The identified nested mosaics, if any.
(12)                        Returns
(12)                        -------
(12)                        dict[label, Axes]
(16)                            A flat dict of all of the Axes created.
(12)                        """
(12)                        output = dict()
(12)                        this_level = dict()
(12)                        for name in unique_ids:
(16)                            indx = np.argwhere(mosaic == name)
(16)                            start_row, start_col = np.min(indx, axis=0)
(16)                            end_row, end_col = np.max(indx, axis=0) + 1
(16)                            slc = (slice(start_row, end_row), slice(start_col, end_col))
(16)                            if (mosaic[slc] != name).any():
(20)                                raise ValueError(
(24)                                    f"While trying to layout\n{mosaic!r}\n"
(24)                                    f"we found that the label {name!r} specifies a "
(24)                                    "non-rectangular or non-contiguous area.")
(16)                            this_level[(start_row, start_col)] = (name, slc, 'axes')
(12)                        for (j, k), nested_mosaic in nested.items():
(16)                            this_level[(j, k)] = (None, nested_mosaic, 'nested')
(12)                        for key in sorted(this_level):
(16)                            name, arg, method = this_level[key]
(16)                            if method == 'axes':
(20)                                slc = arg
(20)                                if name in output:
(24)                                    raise ValueError(f"There are duplicate keys {name} "
(41)                                                     f"in the layout\n{mosaic!r}")
(20)                                ax = self.add_subplot(
(24)                                    gs[slc], **{
(28)                                        'label': str(name),
(28)                                        **subplot_kw,
(28)                                        **per_subplot_kw.get(name, {})
(24)                                    }
(20)                                )
(20)                                output[name] = ax
(16)                            elif method == 'nested':
(20)                                nested_mosaic = arg
(20)                                j, k = key
(20)                                rows, cols = nested_mosaic.shape
(20)                                nested_output = _do_layout(
(24)                                    gs[j, k].subgridspec(rows, cols),
(24)                                    nested_mosaic,
(24)                                    *_identify_keys_and_nested(nested_mosaic)
(20)                                )
(20)                                overlap = set(output) & set(nested_output)
(20)                                if overlap:
(24)                                    raise ValueError(
(28)                                        f"There are duplicate keys {overlap} "
(28)                                        f"between the outer layout\n{mosaic!r}\n"
(28)                                        f"and the nested layout\n{nested_mosaic}"
(24)                                    )
(20)                                output.update(nested_output)
(16)                            else:
(20)                                raise RuntimeError("This should never happen")
(12)                        return output
(8)                     mosaic = _make_array(mosaic)
(8)                     rows, cols = mosaic.shape
(8)                     gs = self.add_gridspec(rows, cols, **gridspec_kw)
(8)                     ret = _do_layout(gs, mosaic, *_identify_keys_and_nested(mosaic))
(8)                     ax0 = next(iter(ret.values()))
(8)                     for ax in ret.values():
```

```
(12)                        if sharex:
(16)                            ax.sharex(ax0)
(16)                            ax._label_outer_xaxis(skip_non_rectangular_axes=True)
(12)                        if sharey:
(16)                            ax.sharey(ax0)
(16)                            ax._label_outer_yaxis(skip_non_rectangular_axes=True)
(8)                     if extra := set(per_subplot_kw) - set(ret):
(12)                        raise ValueError(
(16)                            f"The keys {extra} are in *per_subplot_kw* "
(16)                            "but not in the mosaic."
(12)                        )
(8)                     return ret
(4)                 def _set_artist_props(self, a):
(8)                     if a != self:
(12)                        a.set_figure(self)
(8)                     a.stale_callback = _stale_figure_callback
(8)                     a.set_transform(self.transSubfigure)
(0)             @_docstring.interpd
(0)             class SubFigure(FigureBase):
(4)                 """
(4)                 Logical figure that can be placed inside a figure.
(4)                 See :ref:`figure-api-subfigure` for an index of methods on this class.
(4)                 Typically instantiated using `.Figure.add_subfigure` or
(4)                 `.SubFigure.add_subfigure`, or `.SubFigure.subfigures`.  A subfigure has
(4)                 the same methods as a figure except for those particularly tied to the
size
(4)                 or dpi of the figure, and is confined to a prescribed region of the
figure.
(4)                 For example the following puts two subfigures side-by-side::
(8)                     fig = plt.figure()
(8)                     sfigs = fig.subfigures(1, 2)
(8)                     axsL = sfigs[0].subplots(1, 2)
(8)                     axsR = sfigs[1].subplots(2, 1)
(4)                 See :doc:`/gallery/subplots_axes_and_figures/subfigures`
(4)                 .. note::
(8)                     The *subfigure* concept is new in v3.4, and the API is still
provisional.
(4)                 """
(4)                 def __init__(self, parent, subplotspec, *,
(17)                             facecolor=None,
(17)                             edgecolor=None,
(17)                             linewidth=0.0,
(17)                             frameon=None,
(17)                             **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     parent : `.Figure` or `.SubFigure`
(12)                        Figure or subfigure that contains the SubFigure.  SubFigures
(12)                        can be nested.
(8)                     subplotspec : `.gridspec.SubplotSpec`
(12)                        Defines the region in a parent gridspec where the subfigure will
(12)                        be placed.
(8)                     facecolor : default: ``"none"``
(12)                        The figure patch face color; transparent by default.
(8)                     edgecolor : default: :rc:`figure.edgecolor`
(12)                        The figure patch edge color.
(8)                     linewidth : float
(12)                        The linewidth of the frame (i.e. the edge linewidth of the figure
(12)                        patch).
(8)                     frameon : bool, default: :rc:`figure.frameon`
(12)                        If ``False``, suppress drawing the figure background patch.
(8)                     Other Parameters
(8)                     ----------------
(8)                     **kwargs : `.SubFigure` properties, optional
(12)                        %(SubFigure:kwdoc)s
(8)                     """
(8)                     super().__init__(**kwargs)
(8)                     if facecolor is None:
```

```
(12)                        facecolor = "none"
(8)                     if edgecolor is None:
(12)                        edgecolor = mpl.rcParams['figure.edgecolor']
(8)                     if frameon is None:
(12)                        frameon = mpl.rcParams['figure.frameon']
(8)                     self._subplotspec = subplotspec
(8)                     self._parent = parent
(8)                     self.figure = parent.figure
(8)                     self._axstack = parent._axstack
(8)                     self.subplotpars = parent.subplotpars
(8)                     self.dpi_scale_trans = parent.dpi_scale_trans
(8)                     self._axobservers = parent._axobservers
(8)                     self.transFigure = parent.transFigure
(8)                     self.bbox_relative = Bbox.null()
(8)                     self._redo_transform_rel_fig()
(8)                     self.figbbox = self._parent.figbbox
(8)                     self.bbox = TransformedBbox(self.bbox_relative,
(36)                                                self._parent.transSubfigure)
(8)                     self.transSubfigure = BboxTransformTo(self.bbox)
(8)                     self.patch = Rectangle(
(12)                        xy=(0, 0), width=1, height=1, visible=frameon,
(12)                        facecolor=facecolor, edgecolor=edgecolor, linewidth=linewidth,
(12)                        in_layout=False, transform=self.transSubfigure)
(8)                     self._set_artist_props(self.patch)
(8)                     self.patch.set_antialiased(False)
(4)                 @property
(4)                 def canvas(self):
(8)                     return self._parent.canvas
(4)                 @property
(4)                 def dpi(self):
(8)                     return self._parent.dpi
(4)                 @dpi.setter
(4)                 def dpi(self, value):
(8)                     self._parent.dpi = value
(4)                 def get_dpi(self):
(8)                     """
(8)                     Return the resolution of the parent figure in dots-per-inch as a
float.
(8)                     """
(8)                     return self._parent.dpi
(4)                 def set_dpi(self, val):
(8)                     """
(8)                     Set the resolution of parent figure in dots-per-inch.
(8)                     Parameters
(8)                     ----------
(8)                     val : float
(8)                     """
(8)                     self._parent.dpi = val
(8)                     self.stale = True
(4)                 def _get_renderer(self):
(8)                     return self._parent._get_renderer()
(4)                 def _redo_transform_rel_fig(self, bbox=None):
(8)                     """
(8)                     Make the transSubfigure bbox relative to Figure transform.
(8)                     Parameters
(8)                     ----------
(8)                     bbox : bbox or None
(12)                        If not None, then the bbox is used for relative bounding box.
(12)                        Otherwise, it is calculated from the subplotspec.
(8)                     """
(8)                     if bbox is not None:
(12)                        self.bbox_relative.p0 = bbox.p0
(12)                        self.bbox_relative.p1 = bbox.p1
(12)                        return
(8)                     gs = self._subplotspec.get_gridspec()
(8)                     wr = np.asarray(gs.get_width_ratios())
(8)                     hr = np.asarray(gs.get_height_ratios())
(8)                     dx = wr[self._subplotspec.colspan].sum() / wr.sum()
(8)                     dy = hr[self._subplotspec.rowspan].sum() / hr.sum()
```

```
(8)                 x0 = wr[:self._subplotspec.colspan.start].sum() / wr.sum()
(8)                 y0 = 1 - hr[:self._subplotspec.rowspan.stop].sum() / hr.sum()
(8)                 self.bbox_relative.p0 = (x0, y0)
(8)                 self.bbox_relative.p1 = (x0 + dx, y0 + dy)
(4)             def get_constrained_layout(self):
(8)                 """
(8)                 Return whether constrained layout is being used.
(8)                 See :ref:`constrainedlayout_guide`.
(8)                 """
(8)                 return self._parent.get_constrained_layout()
(4)             def get_constrained_layout_pads(self, relative=False):
(8)                 """
(8)                 Get padding for ``constrained_layout``.
(8)                 Returns a list of ``w_pad, h_pad`` in inches and
(8)                 ``wspace`` and ``hspace`` as fractions of the subplot.
(8)                 See :ref:`constrainedlayout_guide`.
(8)                 Parameters
(8)                 ----------
(8)                 relative : bool
(12)                    If `True`, then convert from inches to figure relative.
(8)                 """
(8)                 return self._parent.get_constrained_layout_pads(relative=relative)
(4)             def get_layout_engine(self):
(8)                 return self._parent.get_layout_engine()
(4)             @property
(4)             def axes(self):
(8)                 """
(8)                 List of Axes in the SubFigure.  You can access and modify the Axes
(8)                 in the SubFigure through this list.
(8)                 Modifying this list has no effect. Instead, use
`~.SubFigure.add_axes`,
(8)                 `~.SubFigure.add_subplot` or `~.SubFigure.delaxes` to add or remove an
(8)                 Axes.
(8)                 Note: The `.SubFigure.axes` property and `~.SubFigure.get_axes` method
(8)                 are equivalent.
(8)                 """
(8)                 return self._localaxes[:]
(4)             get_axes = axes.fget
(4)             def draw(self, renderer):
(8)                 if not self.get_visible():
(12)                    return
(8)                 artists = self._get_draw_artists(renderer)
(8)                 try:
(12)                    renderer.open_group('subfigure', gid=self.get_gid())
(12)                    self.patch.draw(renderer)
(12)                    mimage._draw_list_compositing_images(
(16)                        renderer, self, artists, self.figure.suppressComposite)
(12)                    renderer.close_group('subfigure')
(8)                 finally:
(12)                    self.stale = False
(0)         @_docstring.interpd
(0)         class Figure(FigureBase):
(4)             """
(4)             The top level container for all the plot elements.
(4)             See `matplotlib.figure` for an index of class methods.
(4)             Attributes
(4)             ----------
(4)             patch
(8)                 The `.Rectangle` instance representing the figure background patch.
(4)             suppressComposite
(8)                 For multiple images, the figure will make composite images
(8)                 depending on the renderer option_image_nocomposite function.  If
(8)                 *suppressComposite* is a boolean, this will override the renderer.
(4)             """
(4)             _render_lock = threading.RLock()
(4)             def __str__(self):
(8)                 return "Figure(%gx%g)" % tuple(self.bbox.size)
(4)             def __repr__(self):
(8)                 return "<{clsname} size {h:g}x{w:g} with {naxes} Axes>".format(
```

```
(12)                        clsname=self.__class__.__name__,
(12)                        h=self.bbox.size[0], w=self.bbox.size[1],
(12)                        naxes=len(self.axes),
(8)                     )
(4)                 def __init__(self,
(17)                            figsize=None,
(17)                            dpi=None,
(17)                            *,
(17)                            facecolor=None,
(17)                            edgecolor=None,
(17)                            linewidth=0.0,
(17)                            frameon=None,
(17)                            subplotpars=None,  # rc figure.subplot.*
(17)                            tight_layout=None,  # rc figure.autolayout
(17)                            constrained_layout=None,  # rc figure.constrained_layout.use
(17)                            layout=None,
(17)                            **kwargs
(17)                            ):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     figsize : 2-tuple of floats, default: :rc:`figure.figsize`
(12)                        Figure dimension ``(width, height)`` in inches.
(8)                     dpi : float, default: :rc:`figure.dpi`
(12)                        Dots per inch.
(8)                     facecolor : default: :rc:`figure.facecolor`
(12)                        The figure patch facecolor.
(8)                     edgecolor : default: :rc:`figure.edgecolor`
(12)                        The figure patch edge color.
(8)                     linewidth : float
(12)                        The linewidth of the frame (i.e. the edge linewidth of the figure
(12)                        patch).
(8)                     frameon : bool, default: :rc:`figure.frameon`
(12)                        If ``False``, suppress drawing the figure background patch.
(8)                     subplotpars : `~matplotlib.gridspec.SubplotParams`
(12)                        Subplot parameters. If not given, the default subplot
(12)                        parameters :rc:`figure.subplot.*` are used.
(8)                     tight_layout : bool or dict, default: :rc:`figure.autolayout`
(12)                        Whether to use the tight layout mechanism. See
`.set_tight_layout`.
(12)                        .. admonition:: Discouraged
(16)                            The use of this parameter is discouraged. Please use
(16)                            ``layout='tight'`` instead for the common case of
(16)                            ``tight_layout=True`` and use `.set_tight_layout` otherwise.
(8)                     constrained_layout : bool, default:
:rc:`figure.constrained_layout.use`
(12)                        This is equal to ``layout='constrained'``.
(12)                        .. admonition:: Discouraged
(16)                            The use of this parameter is discouraged. Please use
(16)                            ``layout='constrained'`` instead.
(8)                     layout : {'constrained', 'compressed', 'tight', 'none',
`.LayoutEngine`, \
(0)             None}, default: None
(12)                        The layout mechanism for positioning of plot elements to avoid
(12)                        overlapping Axes decorations (labels, ticks, etc). Note that
(12)                        layout managers can have significant performance penalties.
(12)                        - 'constrained': The constrained layout solver adjusts Axes sizes
(14)                          to avoid overlapping Axes decorations.  Can handle complex plot
(14)                          layouts and colorbars, and is thus recommended.
(14)                          See :ref:`constrainedlayout_guide` for examples.
(12)                        - 'compressed': uses the same algorithm as 'constrained', but
(14)                          removes extra space between fixed-aspect-ratio Axes.  Best for
(14)                          simple grids of Axes.
(12)                        - 'tight': Use the tight layout mechanism. This is a relatively
(14)                          simple algorithm that adjusts the subplot parameters so that
(14)                          decorations do not overlap.
(14)                          See :ref:`tight_layout_guide` for examples.
(12)                        - 'none': Do not use a layout engine.
(12)                        - A `.LayoutEngine` instance. Builtin layout classes are
```

```
(14)                        `.ConstrainedLayoutEngine` and `.TightLayoutEngine`, more easily
(14)                        accessible by 'constrained' and 'tight'.  Passing an instance
(14)                        allows third parties to provide their own layout engine.
(12)                    If not given, fall back to using the parameters *tight_layout* and
(12)                    *constrained_layout*, including their config defaults
(12)                    :rc:`figure.autolayout` and :rc:`figure.constrained_layout.use`.
(8)                 Other Parameters
(8)                 ----------------
(8)                 **kwargs : `.Figure` properties, optional
(12)                    %(Figure:kwdoc)s
(8)                 """
(8)                 super().__init__(**kwargs)
(8)                 self.figure = self
(8)                 self._layout_engine = None
(8)                 if layout is not None:
(12)                    if (tight_layout is not None):
(16)                        _api.warn_external(
(20)                            "The Figure parameters 'layout' and 'tight_layout' cannot
"
(20)                            "be used together. Please use 'layout' only.")
(12)                    if (constrained_layout is not None):
(16)                        _api.warn_external(
(20)                            "The Figure parameters 'layout' and 'constrained_layout' "
(20)                            "cannot be used together. Please use 'layout' only.")
(12)                    self.set_layout_engine(layout=layout)
(8)                 elif tight_layout is not None:
(12)                    if constrained_layout is not None:
(16)                        _api.warn_external(
(20)                            "The Figure parameters 'tight_layout' and "
(20)                            "'constrained_layout' cannot be used together. Please use
"
(20)                            "'layout' parameter")
(12)                    self.set_layout_engine(layout='tight')
(12)                    if isinstance(tight_layout, dict):
(16)                        self.get_layout_engine().set(**tight_layout)
(8)                 elif constrained_layout is not None:
(12)                    if isinstance(constrained_layout, dict):
(16)                        self.set_layout_engine(layout='constrained')
(16)                        self.get_layout_engine().set(**constrained_layout)
(12)                    elif constrained_layout:
(16)                        self.set_layout_engine(layout='constrained')
(8)                 else:
(12)                    self.set_layout_engine(layout=layout)
(8)                 self._canvas_callbacks = cbook.CallbackRegistry(
(12)                    signals=FigureCanvasBase.events)
(8)                 connect = self._canvas_callbacks._connect_picklable
(8)                 self._mouse_key_ids = [
(12)                    connect('key_press_event', backend_bases._key_handler),
(12)                    connect('key_release_event', backend_bases._key_handler),
(12)                    connect('key_release_event', backend_bases._key_handler),
(12)                    connect('button_press_event', backend_bases._mouse_handler),
(12)                    connect('button_release_event', backend_bases._mouse_handler),
(12)                    connect('scroll_event', backend_bases._mouse_handler),
(12)                    connect('motion_notify_event', backend_bases._mouse_handler),
(8)                 ]
(8)                 self._button_pick_id = connect('button_press_event', self.pick)
(8)                 self._scroll_pick_id = connect('scroll_event', self.pick)
(8)                 if figsize is None:
(12)                    figsize = mpl.rcParams['figure.figsize']
(8)                 if dpi is None:
(12)                    dpi = mpl.rcParams['figure.dpi']
(8)                 if facecolor is None:
(12)                    facecolor = mpl.rcParams['figure.facecolor']
(8)                 if edgecolor is None:
(12)                    edgecolor = mpl.rcParams['figure.edgecolor']
(8)                 if frameon is None:
(12)                    frameon = mpl.rcParams['figure.frameon']
(8)                 if not np.isfinite(figsize).all() or (np.array(figsize) < 0).any():
(12)                    raise ValueError('figure size must be positive finite not '
```

```
(29)                                    f'{figsize}')
(8)                     self.bbox_inches = Bbox.from_bounds(0, 0, *figsize)
(8)                     self.dpi_scale_trans = Affine2D().scale(dpi)
(8)                     self._dpi = dpi
(8)                     self.bbox = TransformedBbox(self.bbox_inches, self.dpi_scale_trans)
(8)                     self.figbbox = self.bbox
(8)                     self.transFigure = BboxTransformTo(self.bbox)
(8)                     self.transSubfigure = self.transFigure
(8)                     self.patch = Rectangle(
(12)                        xy=(0, 0), width=1, height=1, visible=frameon,
(12)                        facecolor=facecolor, edgecolor=edgecolor, linewidth=linewidth,
(12)                        in_layout=False)
(8)                     self._set_artist_props(self.patch)
(8)                     self.patch.set_antialiased(False)
(8)                     FigureCanvasBase(self)  # Set self.canvas.
(8)                     if subplotpars is None:
(12)                        subplotpars = SubplotParams()
(8)                     self.subplotpars = subplotpars
(8)                     self._axstack = _AxesStack()  # track all figure Axes and current Axes
(8)                     self.clear()
(4)                 def pick(self, mouseevent):
(8)                     if not self.canvas.widgetlock.locked():
(12)                        super().pick(mouseevent)
(4)                 def _check_layout_engines_compat(self, old, new):
(8)                     """
(8)                     Helper for set_layout engine
(8)                     If the figure has used the old engine and added a colorbar then the
(8)                     value of colorbar_gridspec must be the same on the new engine.
(8)                     """
(8)                     if old is None or new is None:
(12)                        return True
(8)                     if old.colorbar_gridspec == new.colorbar_gridspec:
(12)                        return True
(8)                     for ax in self.axes:
(12)                        if hasattr(ax, '_colorbar'):
(16)                            return False
(8)                     return True
(4)                 def set_layout_engine(self, layout=None, **kwargs):
(8)                     """
(8)                     Set the layout engine for this figure.
(8)                     Parameters
(8)                     ----------
(8)                     layout : {'constrained', 'compressed', 'tight', 'none',
`.LayoutEngine`, None}
(12)                        - 'constrained' will use `~.ConstrainedLayoutEngine`
(12)                        - 'compressed' will also use `~.ConstrainedLayoutEngine`, but with
(14)                          a correction that attempts to make a good layout for fixed-
aspect
(14)                          ratio Axes.
(12)                        - 'tight' uses `~.TightLayoutEngine`
(12)                        - 'none' removes layout engine.
(12)                        If a `.LayoutEngine` instance, that instance will be used.
(12)                        If `None`, the behavior is controlled by :rc:`figure.autolayout`
(12)                        (which if `True` behaves as if 'tight' was passed) and
(12)                        :rc:`figure.constrained_layout.use` (which if `True` behaves as if
(12)                        'constrained' was passed).  If both are `True`,
(12)                        :rc:`figure.autolayout` takes priority.
(12)                        Users and libraries can define their own layout engines and pass
(12)                        the instance directly as well.
(8)                     **kwargs
(12)                        The keyword arguments are passed to the layout engine to set
things
(12)                        like padding and margin sizes.  Only used if *layout* is a string.
(8)                     """
(8)                     if layout is None:
(12)                        if mpl.rcParams['figure.autolayout']:
(16)                            layout = 'tight'
(12)                        elif mpl.rcParams['figure.constrained_layout.use']:
(16)                            layout = 'constrained'
```

```
2083: (12)                      else:
2084: (16)                          self._layout_engine = None
2085: (16)                          return
2086: (8)                   if layout == 'tight':
2087: (12)                      new_layout_engine = TightLayoutEngine(**kwargs)
2088: (8)                   elif layout == 'constrained':
2089: (12)                      new_layout_engine = ConstrainedLayoutEngine(**kwargs)
2090: (8)                   elif layout == 'compressed':
2091: (12)                      new_layout_engine = ConstrainedLayoutEngine(compress=True,
2092: (56)                                                                  **kwargs)
2093: (8)                   elif layout == 'none':
2094: (12)                      if self._layout_engine is not None:
2095: (16)                          new_layout_engine = PlaceHolderLayoutEngine(
2096: (20)                              self._layout_engine.adjust_compatible,
2097: (20)                              self._layout_engine.colorbar_gridspec
2098: (16)                          )
2099: (12)                      else:
2100: (16)                          new_layout_engine = None
2101: (8)                   elif isinstance(layout, LayoutEngine):
2102: (12)                      new_layout_engine = layout
2103: (8)                   else:
2104: (12)                      raise ValueError(f"Invalid value for 'layout': {layout!r}")
2105: (8)                   if self._check_layout_engines_compat(self._layout_engine,
2106: (45)                                                       new_layout_engine):
2107: (12)                      self._layout_engine = new_layout_engine
2108: (8)                   else:
2109: (12)                      raise RuntimeError('Colorbar layout of new layout engine not '
2110: (31)                                         'compatible with old engine, and a colorbar '
2111: (31)                                         'has been created.  Engine not changed.')
2112: (4)               def get_layout_engine(self):
2113: (8)                   return self._layout_engine
2114: (4)               def _repr_html_(self):
2115: (8)                   if 'WebAgg' in type(self.canvas).__name__:
2116: (12)                      from matplotlib.backends import backend_webagg
2117: (12)                      return backend_webagg.ipython_inline_display(self)
2118: (4)               def show(self, warn=True):
2119: (8)                   """
2120: (8)                   If using a GUI backend with pyplot, display the figure window.
2121: (8)                   If the figure was not created using `~.pyplot.figure`, it will lack
2122: (8)                   a `~.backend_bases.FigureManagerBase`, and this method will raise an
2123: (8)                   AttributeError.
2124: (8)                   .. warning::
2125: (12)                      This does not manage an GUI event loop. Consequently, the figure
2126: (12)                      may only be shown briefly or not shown at all if you or your
2127: (12)                      environment are not managing an event loop.
2128: (12)                      Use cases for `.Figure.show` include running this from a GUI
2129: (12)                      application (where there is persistently an event loop running) or
2130: (12)                      from a shell, like IPython, that install an input hook to allow
the
2131: (12)                      interactive shell to accept input while the figure is also being
2132: (12)                      shown and interactive.  Some, but not all, GUI toolkits will
2133: (12)                      register an input hook on import.  See :ref:`cp_integration` for
2134: (12)                      more details.
2135: (12)                      If you're in a shell without input hook integration or executing a
2136: (12)                      python script, you should use `matplotlib.pyplot.show` with
2137: (12)                      ``block=True`` instead, which takes care of starting and running
2138: (12)                      the event loop for you.
2139: (8)                   Parameters
2140: (8)                   ----------
2141: (8)                   warn : bool, default: True
2142: (12)                      If ``True`` and we are not running headless (i.e. on Linux with an
2143: (12)                      unset DISPLAY), issue warning when called on a non-GUI backend.
2144: (8)                   """
2145: (8)                   if self.canvas.manager is None:
2146: (12)                      raise AttributeError(
2147: (16)                          "Figure.show works only for figures managed by pyplot, "
2148: (16)                          "normally created by pyplot.figure()")
2149: (8)                   try:
2150: (12)                      self.canvas.manager.show()
```

```
(8)                 except NonGuiException as exc:
(12)                    if warn:
(16)                        _api.warn_external(str(exc))
(4)             @property
(4)             def axes(self):
(8)                 """
(8)                 List of Axes in the Figure. You can access and modify the Axes in the
(8)                 Figure through this list.
(8)                 Do not modify the list itself. Instead, use `~Figure.add_axes`,
(8)                 `~.Figure.add_subplot` or `~.Figure.delaxes` to add or remove an Axes.
(8)                 Note: The `.Figure.axes` property and `~.Figure.get_axes` method are
(8)                 equivalent.
(8)                 """
(8)                 return self._axstack.as_list()
(4)             get_axes = axes.fget
(4)             def _get_renderer(self):
(8)                 if hasattr(self.canvas, 'get_renderer'):
(12)                    return self.canvas.get_renderer()
(8)                 else:
(12)                    return _get_renderer(self)
(4)             def _get_dpi(self):
(8)                 return self._dpi
(4)             def _set_dpi(self, dpi, forward=True):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 dpi : float
(8)                 forward : bool
(12)                    Passed on to `~.Figure.set_size_inches`
(8)                 """
(8)                 if dpi == self._dpi:
(12)                    return
(8)                 self._dpi = dpi
(8)                 self.dpi_scale_trans.clear().scale(dpi)
(8)                 w, h = self.get_size_inches()
(8)                 self.set_size_inches(w, h, forward=forward)
(4)             dpi = property(_get_dpi, _set_dpi, doc="The resolution in dots per inch.")
(4)             def get_tight_layout(self):
(8)                 """Return whether `.Figure.tight_layout` is called when drawing."""
(8)                 return isinstance(self.get_layout_engine(), TightLayoutEngine)
(4)             @_api.deprecated("3.6", alternative="set_layout_engine",
(21)                             pending=True)
(4)             def set_tight_layout(self, tight):
(8)                 """
(8)                 Set whether and how `.Figure.tight_layout` is called when drawing.
(8)                 Parameters
(8)                 ----------
(8)                 tight : bool or dict with keys "pad", "w_pad", "h_pad", "rect" or None
(12)                    If a bool, sets whether to call `.Figure.tight_layout` upon
drawing.
(12)                    If ``None``, use :rc:`figure.autolayout` instead.
(12)                    If a dict, pass it as kwargs to `.Figure.tight_layout`, overriding
the
(12)                    default paddings.
(8)                 """
(8)                 if tight is None:
(12)                    tight = mpl.rcParams['figure.autolayout']
(8)                 _tight = 'tight' if bool(tight) else 'none'
(8)                 _tight_parameters = tight if isinstance(tight, dict) else {}
(8)                 self.set_layout_engine(_tight, **_tight_parameters)
(8)                 self.stale = True
(4)             def get_constrained_layout(self):
(8)                 """
(8)                 Return whether constrained layout is being used.
(8)                 See :ref:`constrainedlayout_guide`.
(8)                 """
(8)                 return isinstance(self.get_layout_engine(), ConstrainedLayoutEngine)
(4)             @_api.deprecated("3.6", alternative="set_layout_engine('constrained')",
(21)                             pending=True)
```

```
(4)    def set_constrained_layout(self, constrained):
(8)        """
(8)        Set whether ``constrained_layout`` is used upon drawing.
(8)        If None, :rc:`figure.constrained_layout.use` value will be used.
(8)        When providing a dict containing the keys ``w_pad``, ``h_pad``
(8)        the default ``constrained_layout`` paddings will be
(8)        overridden.  These pads are in inches and default to 3.0/72.0.
(8)        ``w_pad`` is the width padding and ``h_pad`` is the height padding.
(8)        Parameters
(8)        ----------
(8)        constrained : bool or dict or None
(8)        """
(8)        if constrained is None:
(12)           constrained = mpl.rcParams['figure.constrained_layout.use']
(8)        _constrained = 'constrained' if bool(constrained) else 'none'
(8)        _parameters = constrained if isinstance(constrained, dict) else {}
(8)        self.set_layout_engine(_constrained, **_parameters)
(8)        self.stale = True
(4)    @_api.deprecated(
(9)         "3.6", alternative="figure.get_layout_engine().set()",
(9)         pending=True)
(4)    def set_constrained_layout_pads(self, **kwargs):
(8)        """
(8)        Set padding for ``constrained_layout``.
(8)        Tip: The parameters can be passed from a dictionary by using
(8)        ``fig.set_constrained_layout(**pad_dict)``.
(8)        See :ref:`constrainedlayout_guide`.
(8)        Parameters
(8)        ----------
(8)        w_pad : float, default: :rc:`figure.constrained_layout.w_pad`
(12)           Width padding in inches.  This is the pad around Axes
(12)           and is meant to make sure there is enough room for fonts to
(12)           look good.  Defaults to 3 pts = 0.04167 inches
(8)        h_pad : float, default: :rc:`figure.constrained_layout.h_pad`
(12)           Height padding in inches. Defaults to 3 pts.
(8)        wspace : float, default: :rc:`figure.constrained_layout.wspace`
(12)           Width padding between subplots, expressed as a fraction of the
(12)           subplot width.  The total padding ends up being w_pad + wspace.
(8)        hspace : float, default: :rc:`figure.constrained_layout.hspace`
(12)           Height padding between subplots, expressed as a fraction of the
(12)           subplot width. The total padding ends up being h_pad + hspace.
(8)        """
(8)        if isinstance(self.get_layout_engine(), ConstrainedLayoutEngine):
(12)           self.get_layout_engine().set(**kwargs)
(4)    @_api.deprecated("3.6", alternative="fig.get_layout_engine().get()",
(21)                    pending=True)
(4)    def get_constrained_layout_pads(self, relative=False):
(8)        """
(8)        Get padding for ``constrained_layout``.
(8)        Returns a list of ``w_pad, h_pad`` in inches and
(8)        ``wspace`` and ``hspace`` as fractions of the subplot.
(8)        All values are None if ``constrained_layout`` is not used.
(8)        See :ref:`constrainedlayout_guide`.
(8)        Parameters
(8)        ----------
(8)        relative : bool
(12)           If `True`, then convert from inches to figure relative.
(8)        """
(8)        if not isinstance(self.get_layout_engine(), ConstrainedLayoutEngine):
(12)           return None, None, None, None
(8)        info = self.get_layout_engine().get()
(8)        w_pad = info['w_pad']
(8)        h_pad = info['h_pad']
(8)        wspace = info['wspace']
(8)        hspace = info['hspace']
(8)        if relative and (w_pad is not None or h_pad is not None):
(12)           renderer = self._get_renderer()
(12)           dpi = renderer.dpi
(12)           w_pad = w_pad * dpi / renderer.width
```

```
(12)                        h_pad = h_pad * dpi / renderer.height
(8)                     return w_pad, h_pad, wspace, hspace
(4)                 def set_canvas(self, canvas):
(8)                     """
(8)                     Set the canvas that contains the figure
(8)                     Parameters
(8)                     ----------
(8)                     canvas : FigureCanvas
(8)                     """
(8)                     self.canvas = canvas
(4)                 @_docstring.interpd
(4)                 def figimage(self, X, xo=0, yo=0, alpha=None, norm=None, cmap=None,
(17)                             vmin=None, vmax=None, origin=None, resize=False, **kwargs):
(8)                     """
(8)                     Add a non-resampled image to the figure.
(8)                     The image is attached to the lower or upper left corner depending on
(8)                     *origin*.
(8)                     Parameters
(8)                     ----------
(8)                     X
(12)                        The image data. This is an array of one of the following shapes:
(12)                        - (M, N): an image with scalar data.  Color-mapping is controlled
(14)                          by *cmap*, *norm*, *vmin*, and *vmax*.
(12)                        - (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
(12)                        - (M, N, 4): an image with RGBA values (0-1 float or 0-255 int),
(14)                          i.e. including transparency.
(8)                     xo, yo : int
(12)                        The *x*/*y* image offset in pixels.
(8)                     alpha : None or float
(12)                        The alpha blending value.
(8)                     %(cmap_doc)s
(12)                        This parameter is ignored if *X* is RGB(A).
(8)                     %(norm_doc)s
(12)                        This parameter is ignored if *X* is RGB(A).
(8)                     %(vmin_vmax_doc)s
(12)                        This parameter is ignored if *X* is RGB(A).
(8)                     origin : {'upper', 'lower'}, default: :rc:`image.origin`
(12)                        Indicates where the [0, 0] index of the array is in the upper left
(12)                        or lower left corner of the Axes.
(8)                     resize : bool
(12)                        If *True*, resize the figure to match the given image size.
(8)                     Returns
(8)                     -------
(8)                     `matplotlib.image.FigureImage`
(8)                     Other Parameters
(8)                     ----------------
(8)                     **kwargs
(12)                        Additional kwargs are `.Artist` kwargs passed on to
`.FigureImage`.
(8)                     Notes
(8)                     -----
(8)                     figimage complements the Axes image (`~matplotlib.axes.Axes.imshow`)
(8)                     which will be resampled to fit the current Axes.  If you want
(8)                     a resampled image to fill the entire figure, you can define an
(8)                     `~matplotlib.axes.Axes` with extent [0, 0, 1, 1].
(8)                     Examples
(8)                     --------
(8)                     ::
(12)                        f = plt.figure()
(12)                        nx = int(f.get_figwidth() * f.dpi)
(12)                        ny = int(f.get_figheight() * f.dpi)
(12)                        data = np.random.random((ny, nx))
(12)                        f.figimage(data)
(12)                        plt.show()
(8)                     """
(8)                     if resize:
(12)                        dpi = self.get_dpi()
(12)                        figsize = [x / dpi for x in (X.shape[1], X.shape[0])]
(12)                        self.set_size_inches(figsize, forward=True)
```

```
(8)                 im = mimage.FigureImage(self, cmap=cmap, norm=norm,
(32)                                        offsetx=xo, offsety=yo,
(32)                                        origin=origin, **kwargs)
(8)                 im.stale_callback = _stale_figure_callback
(8)                 im.set_array(X)
(8)                 im.set_alpha(alpha)
(8)                 if norm is None:
(12)                    im.set_clim(vmin, vmax)
(8)                 self.images.append(im)
(8)                 im._remove_method = self.images.remove
(8)                 self.stale = True
(8)                 return im
(4)             def set_size_inches(self, w, h=None, forward=True):
(8)                 """
(8)                 Set the figure size in inches.
(8)                 Call signatures::
(13)                     fig.set_size_inches(w, h)  # OR
(13)                     fig.set_size_inches((w, h))
(8)                 Parameters
(8)                 ----------
(8)                 w : (float, float) or float
(12)                    Width and height in inches (if height not specified as a separate
(12)                    argument) or width.
(8)                 h : float
(12)                    Height in inches.
(8)                 forward : bool, default: True
(12)                    If ``True``, the canvas size is automatically updated, e.g.,
(12)                    you can resize the figure window from the shell.
(8)                 See Also
(8)                 --------
(8)                 matplotlib.figure.Figure.get_size_inches
(8)                 matplotlib.figure.Figure.set_figwidth
(8)                 matplotlib.figure.Figure.set_figheight
(8)                 Notes
(8)                 -----
(8)                 To transform from pixels to inches divide by `Figure.dpi`.
(8)                 """
(8)                 if h is None:  # Got called with a single pair as argument.
(12)                    w, h = w
(8)                 size = np.array([w, h])
(8)                 if not np.isfinite(size).all() or (size < 0).any():
(12)                    raise ValueError(f'figure size must be positive finite not
{size}')
(8)                 self.bbox_inches.p1 = size
(8)                 if forward:
(12)                    manager = self.canvas.manager
(12)                    if manager is not None:
(16)                        manager.resize(*(size * self.dpi).astype(int))
(8)                 self.stale = True
(4)             def get_size_inches(self):
(8)                 """
(8)                 Return the current size of the figure in inches.
(8)                 Returns
(8)                 -------
(8)                 ndarray
(11)                   The size (width, height) of the figure in inches.
(8)                 See Also
(8)                 --------
(8)                 matplotlib.figure.Figure.set_size_inches
(8)                 matplotlib.figure.Figure.get_figwidth
(8)                 matplotlib.figure.Figure.get_figheight
(8)                 Notes
(8)                 -----
(8)                 The size in pixels can be obtained by multiplying with `Figure.dpi`.
(8)                 """
(8)                 return np.array(self.bbox_inches.p1)
(4)             def get_figwidth(self):
(8)                 """Return the figure width in inches."""
(8)                 return self.bbox_inches.width
```

```
(4)    def get_figheight(self):
(8)        """Return the figure height in inches."""
(8)        return self.bbox_inches.height
(4)    def get_dpi(self):
(8)        """Return the resolution in dots per inch as a float."""
(8)        return self.dpi
(4)    def set_dpi(self, val):
(8)        """
(8)        Set the resolution of the figure in dots-per-inch.
(8)        Parameters
(8)        ----------
(8)        val : float
(8)        """
(8)        self.dpi = val
(8)        self.stale = True
(4)    def set_figwidth(self, val, forward=True):
(8)        """
(8)        Set the width of the figure in inches.
(8)        Parameters
(8)        ----------
(8)        val : float
(8)        forward : bool
(12)           See `set_size_inches`.
(8)        See Also
(8)        --------
(8)        matplotlib.figure.Figure.set_figheight
(8)        matplotlib.figure.Figure.set_size_inches
(8)        """
(8)        self.set_size_inches(val, self.get_figheight(), forward=forward)
(4)    def set_figheight(self, val, forward=True):
(8)        """
(8)        Set the height of the figure in inches.
(8)        Parameters
(8)        ----------
(8)        val : float
(8)        forward : bool
(12)           See `set_size_inches`.
(8)        See Also
(8)        --------
(8)        matplotlib.figure.Figure.set_figwidth
(8)        matplotlib.figure.Figure.set_size_inches
(8)        """
(8)        self.set_size_inches(self.get_figwidth(), val, forward=forward)
(4)    def clear(self, keep_observers=False):
(8)        super().clear(keep_observers=keep_observers)
(8)        toolbar = self.canvas.toolbar
(8)        if toolbar is not None:
(12)           toolbar.update()
(4)    @_finalize_rasterization
(4)    @allow_rasterization
(4)    def draw(self, renderer):
(8)        if not self.get_visible():
(12)           return
(8)        with self._render_lock:
(12)           artists = self._get_draw_artists(renderer)
(12)           try:
(16)               renderer.open_group('figure', gid=self.get_gid())
(16)               if self.axes and self.get_layout_engine() is not None:
(20)                   try:
(24)                       self.get_layout_engine().execute(self)
(20)                   except ValueError:
(24)                       pass
(16)               self.patch.draw(renderer)
(16)               mimage._draw_list_compositing_images(
(20)                   renderer, self, artists, self.suppressComposite)
(16)               renderer.close_group('figure')
(12)           finally:
(16)               self.stale = False
(12)           DrawEvent("draw_event", self.canvas, renderer)._process()
```

```
(4)     def draw_without_rendering(self):
(8)         """
(8)         Draw the figure with no output.  Useful to get the final size of
(8)         artists that require a draw before their size is known (e.g. text).
(8)         """
(8)         renderer = _get_renderer(self)
(8)         with renderer._draw_disabled():
(12)            self.draw(renderer)
(4)     def draw_artist(self, a):
(8)         """
(8)         Draw `.Artist` *a* only.
(8)         """
(8)         a.draw(self.canvas.get_renderer())
(4)     def __getstate__(self):
(8)         state = super().__getstate__()
(8)         state.pop("canvas")
(8)         state["_dpi"] = state.get('_original_dpi', state['_dpi'])
(8)         state['__mpl_version__'] = mpl.__version__
(8)         from matplotlib import _pylab_helpers
(8)         if self.canvas.manager in _pylab_helpers.Gcf.figs.values():
(12)            state['_restore_to_pylab'] = True
(8)         return state
(4)     def __setstate__(self, state):
(8)         version = state.pop('__mpl_version__')
(8)         restore_to_pylab = state.pop('_restore_to_pylab', False)
(8)         if version != mpl.__version__:
(12)            _api.warn_external(
(16)                f"This figure was saved with matplotlib version {version} and
"
(16)                f"loaded with {mpl.__version__} so may not function
correctly."
(12)            )
(8)         self.__dict__ = state
(8)         FigureCanvasBase(self)  # Set self.canvas.
(8)         if restore_to_pylab:
(12)            import matplotlib.pyplot as plt
(12)            import matplotlib._pylab_helpers as pylab_helpers
(12)            allnums = plt.get_fignums()
(12)            num = max(allnums) + 1 if allnums else 1
(12)            backend = plt._get_backend_mod()
(12)            mgr = backend.new_figure_manager_given_figure(num, self)
(12)            pylab_helpers.Gcf._set_new_active_manager(mgr)
(12)            plt.draw_if_interactive()
(8)         self.stale = True
(4)     def add_axobserver(self, func):
(8)         """Whenever the Axes state change, ``func(self)`` will be called."""
(8)         self._axobservers.connect("_axes_change_event", lambda arg: func(arg))
(4)     def savefig(self, fname, *, transparent=None, **kwargs):
(8)         """
(8)         Save the current figure as an image or vector graphic to a file.
(8)         Call signature::
(10)          savefig(fname, *, transparent=None, dpi='figure', format=None,
(18)                  metadata=None, bbox_inches=None, pad_inches=0.1,
(18)                  facecolor='auto', edgecolor='auto', backend=None,
(18)                  **kwargs
(17)                 )
(8)         The available output formats depend on the backend being used.
(8)         Parameters
(8)         ----------
(8)         fname : str or path-like or binary file-like
(12)            A path, or a Python file-like object, or
(12)            possibly some backend-dependent object such as
(12)            `matplotlib.backends.backend_pdf.PdfPages`.
(12)            If *format* is set, it determines the output format, and the file
(12)            is saved as *fname*.  Note that *fname* is used verbatim, and
there
(12)            is no attempt to make the extension, if any, of *fname* match
(12)            *format*, and no extension is appended.
(12)            If *format* is not set, then the format is inferred from the
```

```
(12)                        extension of *fname*, if there is one.  If *format* is not
(12)                        set and *fname* has no extension, then the file is saved with
(12)                        :rc:`savefig.format` and the appropriate extension is appended to
(12)                        *fname*.
(8)                     Other Parameters
(8)                     ----------------
(8)                     transparent : bool, default: :rc:`savefig.transparent`
(12)                        If *True*, the Axes patches will all be transparent; the
(12)                        Figure patch will also be transparent unless *facecolor*
(12)                        and/or *edgecolor* are specified via kwargs.
(12)                        If *False* has no effect and the color of the Axes and
(12)                        Figure patches are unchanged (unless the Figure patch
(12)                        is specified via the *facecolor* and/or *edgecolor* keyword
(12)                        arguments in which case those colors are used).
(12)                        The transparency of these patches will be restored to their
(12)                        original values upon exit of this function.
(12)                        This is useful, for example, for displaying
(12)                        a plot on top of a colored background on a web page.
(8)                     dpi : float or 'figure', default: :rc:`savefig.dpi`
(12)                        The resolution in dots per inch.  If 'figure', use the figure's
(12)                        dpi value.
(8)                     format : str
(12)                        The file format, e.g. 'png', 'pdf', 'svg', ... The behavior when
(12)                        this is unset is documented under *fname*.
(8)                     metadata : dict, optional
(12)                        Key/value pairs to store in the image metadata. The supported keys
(12)                        and defaults depend on the image format and backend:
(12)                        - 'png' with Agg backend: See the parameter ``metadata`` of
(14)                          `~.FigureCanvasAgg.print_png`.
(12)                        - 'pdf' with pdf backend: See the parameter ``metadata`` of
(14)                          `~.backend_pdf.PdfPages`.
(12)                        - 'svg' with svg backend: See the parameter ``metadata`` of
(14)                          `~.FigureCanvasSVG.print_svg`.
(12)                        - 'eps' and 'ps' with PS backend: Only 'Creator' is supported.
(12)                        Not supported for 'pgf', 'raw', and 'rgba' as those formats do not
support
(12)                        embedding metadata.
(12)                        Does not currently support 'jpg', 'tiff', or 'webp', but may
include
(12)                        embedding EXIF metadata in the future.
(8)                     bbox_inches : str or `.Bbox`, default: :rc:`savefig.bbox`
(12)                        Bounding box in inches: only the given portion of the figure is
(12)                        saved.  If 'tight', try to figure out the tight bbox of the
figure.
(8)                     pad_inches : float or 'layout', default: :rc:`savefig.pad_inches`
(12)                        Amount of padding in inches around the figure when bbox_inches is
(12)                        'tight'. If 'layout' use the padding from the constrained or
(12)                        compressed layout engine; ignored if one of those engines is not
in
(12)                        use.
(8)                     facecolor : :mpltype:`color` or 'auto', default:
:rc:`savefig.facecolor`
(12)                        The facecolor of the figure.  If 'auto', use the current figure
(12)                        facecolor.
(8)                     edgecolor : :mpltype:`color` or 'auto', default:
:rc:`savefig.edgecolor`
(12)                        The edgecolor of the figure.  If 'auto', use the current figure
(12)                        edgecolor.
(8)                     backend : str, optional
(12)                        Use a non-default backend to render the file, e.g. to render a
(12)                        png file with the "cairo" backend rather than the default "agg",
(12)                        or a pdf file with the "pgf" backend rather than the default
(12)                        "pdf".  Note that the default backend is normally sufficient.  See
(12)                        :ref:`the-builtin-backends` for a list of valid backends for each
(12)                        file format.  Custom backends can be referenced as "module://...".
(8)                     orientation : {'landscape', 'portrait'}
(12)                        Currently only supported by the postscript backend.
(8)                     papertype : str
(12)                        One of 'letter', 'legal', 'executive', 'ledger', 'a0' through
```

```
(12)                        'a10', 'b0' through 'b10'. Only supported for postscript
(12)                        output.
(8)                     bbox_extra_artists : list of `~matplotlib.artist.Artist`, optional
(12)                        A list of extra artists that will be considered when the
(12)                        tight bbox is calculated.
(8)                     pil_kwargs : dict, optional
(12)                        Additional keyword arguments that are passed to
(12)                        `PIL.Image.Image.save` when saving the figure.
(8)                     """
(8)                     kwargs.setdefault('dpi', mpl.rcParams['savefig.dpi'])
(8)                     if transparent is None:
(12)                        transparent = mpl.rcParams['savefig.transparent']
(8)                     with ExitStack() as stack:
(12)                        if transparent:
(16)                            def _recursively_make_subfig_transparent(exit_stack, subfig):
(20)                                exit_stack.enter_context(
(24)                                    subfig.patch._cm_set(
(28)                                        facecolor="none", edgecolor="none"))
(20)                                for ax in subfig.axes:
(24)                                    exit_stack.enter_context(
(28)                                        ax.patch._cm_set(
(32)                                            facecolor="none", edgecolor="none"))
(20)                                for sub_subfig in subfig.subfigs:
(24)                                    _recursively_make_subfig_transparent(
(28)                                        exit_stack, sub_subfig)
(16)                            def _recursively_make_axes_transparent(exit_stack, ax):
(20)                                exit_stack.enter_context(
(24)                                    ax.patch._cm_set(facecolor="none", edgecolor="none"))
(20)                                for child_ax in ax.child_axes:
(24)                                    exit_stack.enter_context(
(28)                                        child_ax.patch._cm_set(
(32)                                            facecolor="none", edgecolor="none"))
(20)                                for child_childax in ax.child_axes:
(24)                                    _recursively_make_axes_transparent(
(28)                                        exit_stack, child_childax)
(16)                            kwargs.setdefault('facecolor', 'none')
(16)                            kwargs.setdefault('edgecolor', 'none')
(16)                            for subfig in self.subfigs:
(20)                                _recursively_make_subfig_transparent(stack, subfig)
(16)                            for ax in self.axes:
(20)                                _recursively_make_axes_transparent(stack, ax)
(12)                        self.canvas.print_figure(fname, **kwargs)
(4)                 def ginput(self, n=1, timeout=30, show_clicks=True,
(15)                           mouse_add=MouseButton.LEFT,
(15)                           mouse_pop=MouseButton.RIGHT,
(15)                           mouse_stop=MouseButton.MIDDLE):
(8)                     """
(8)                     Blocking call to interact with a figure.
(8)                     Wait until the user clicks *n* times on the figure, and return the
(8)                     coordinates of each click in a list.
(8)                     There are three possible interactions:
(8)                     - Add a point.
(8)                     - Remove the most recently added point.
(8)                     - Stop the interaction and return the points added so far.
(8)                     The actions are assigned to mouse buttons via the arguments
(8)                     *mouse_add*, *mouse_pop* and *mouse_stop*.
(8)                     Parameters
(8)                     ----------
(8)                     n : int, default: 1
(12)                        Number of mouse clicks to accumulate. If negative, accumulate
(12)                        clicks until the input is terminated manually.
(8)                     timeout : float, default: 30 seconds
(12)                        Number of seconds to wait before timing out. If zero or negative
(12)                        will never time out.
(8)                     show_clicks : bool, default: True
(12)                        If True, show a red cross at the location of each click.
(8)                     mouse_add : `.MouseButton` or None, default: `.MouseButton.LEFT`
(12)                        Mouse button used to add points.
(8)                     mouse_pop : `.MouseButton` or None, default: `.MouseButton.RIGHT`
```

```
            Mouse button used to remove the most recently added point.
        mouse_stop : `.MouseButton` or None, default: `.MouseButton.MIDDLE`
            Mouse button used to stop input.
        Returns
        -------
        list of tuples
            A list of the clicked (x, y) coordinates.
        Notes
        -----
        The keyboard can also be used to select points in case your mouse
        does not have one or more of the buttons.  The delete and backspace
        keys act like right-clicking (i.e., remove last point), the enter key
        terminates input and any other key (not already used by the window
        manager) selects a point.
        """
        clicks = []
        marks = []
        def handler(event):
            is_button = event.name == "button_press_event"
            is_key = event.name == "key_press_event"
            if (is_button and event.button == mouse_stop
                    or is_key and event.key in ["escape", "enter"]):
                self.canvas.stop_event_loop()
            elif (is_button and event.button == mouse_pop
                  or is_key and event.key in ["backspace", "delete"]):
                if clicks:
                    clicks.pop()
                    if show_clicks:
                        marks.pop().remove()
                        self.canvas.draw()
            elif (is_button and event.button == mouse_add
                  or is_key and event.key is not None):
                if event.inaxes:
                    clicks.append((event.xdata, event.ydata))
                    _log.info("input %i: %f, %f",
                              len(clicks), event.xdata, event.ydata)
                    if show_clicks:
                        line = mpl.lines.Line2D([event.xdata], [event.ydata],
                                                marker="+", color="r")
                        event.inaxes.add_line(line)
                        marks.append(line)
                        self.canvas.draw()
            if len(clicks) == n and n > 0:
                self.canvas.stop_event_loop()
        _blocking_input.blocking_input_loop(
            self, ["button_press_event", "key_press_event"], timeout, handler)
        for mark in marks:
            mark.remove()
        self.canvas.draw()
        return clicks
    def waitforbuttonpress(self, timeout=-1):
        """
        Blocking call to interact with the figure.
        Wait for user input and return True if a key was pressed, False if a
        mouse button was pressed and None if no input was given within
        *timeout* seconds.  Negative values deactivate *timeout*.
        """
        event = None
        def handler(ev):
            nonlocal event
            event = ev
            self.canvas.stop_event_loop()
        _blocking_input.blocking_input_loop(
            self, ["button_press_event", "key_press_event"], timeout, handler)
        return None if event is None else event.name == "key_press_event"
    def tight_layout(self, *, pad=1.08, h_pad=None, w_pad=None, rect=None):
        """
        Adjust the padding between and around subplots.
        To exclude an artist on the Axes from the bounding box calculation
```

```
(8)                 that determines the subplot parameters (i.e. legend, or annotation),
(8)                 set ``a.set_in_layout(False)`` for that artist.
(8)                 Parameters
(8)                 ----------
(8)                 pad : float, default: 1.08
(12)                    Padding between the figure edge and the edges of subplots,
(12)                    as a fraction of the font size.
(8)                 h_pad, w_pad : float, default: *pad*
(12)                    Padding (height/width) between edges of adjacent subplots,
(12)                    as a fraction of the font size.
(8)                 rect : tuple (left, bottom, right, top), default: (0, 0, 1, 1)
(12)                    A rectangle in normalized figure coordinates into which the whole
(12)                    subplots area (including labels) will fit.
(8)                 See Also
(8)                 --------
(8)                 .Figure.set_layout_engine
(8)                 .pyplot.tight_layout
(8)                 """
(8)                 engine = TightLayoutEngine(pad=pad, h_pad=h_pad, w_pad=w_pad,
rect=rect)
(8)                 try:
(12)                    previous_engine = self.get_layout_engine()
(12)                    self.set_layout_engine(engine)
(12)                    engine.execute(self)
(12)                    if previous_engine is not None and not isinstance(
(16)                        previous_engine, (TightLayoutEngine, PlaceHolderLayoutEngine)
(12)                    ):
(16)                        _api.warn_external('The figure layout has changed to tight')
(8)                 finally:
(12)                    self.set_layout_engine('none')
(0)         def figaspect(arg):
(4)             """
(4)             Calculate the width and height for a figure with a specified aspect ratio.
(4)             While the height is taken from :rc:`figure.figsize`, the width is
(4)             adjusted to match the desired aspect ratio. Additionally, it is ensured
(4)             that the width is in the range [4., 16.] and the height is in the range
(4)             [2., 16.]. If necessary, the default height is adjusted to ensure this.
(4)             Parameters
(4)             ----------
(4)             arg : float or 2D array
(8)                 If a float, this defines the aspect ratio (i.e. the ratio height /
(8)                 width).
(8)                 In case of an array the aspect ratio is number of rows / number of
(8)                 columns, so that the array could be fitted in the figure undistorted.
(4)             Returns
(4)             -------
(4)             width, height : float
(8)                 The figure size in inches.
(4)             Notes
(4)             -----
(4)             If you want to create an Axes within the figure, that still preserves the
(4)             aspect ratio, be sure to create it with equal width and height. See
(4)             examples below.
(4)             Thanks to Fernando Perez for this function.
(4)             Examples
(4)             --------
(4)             Make a figure twice as tall as it is wide::
(8)                 w, h = figaspect(2.)
(8)                 fig = Figure(figsize=(w, h))
(8)                 ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
(8)                 ax.imshow(A, **kwargs)
(4)             Make a figure with the proper aspect for an array::
(8)                 A = rand(5, 3)
(8)                 w, h = figaspect(A)
(8)                 fig = Figure(figsize=(w, h))
(8)                 ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
(8)                 ax.imshow(A, **kwargs)
(4)             """
(4)             isarray = hasattr(arg, 'shape') and not np.isscalar(arg)
```

```
2827: (4)               figsize_min = np.array((4.0, 2.0))  # min length for width/height
2828: (4)               figsize_max = np.array((16.0, 16.0))  # max length for width/height
2829: (4)               if isarray:
2830: (8)                   nr, nc = arg.shape[:2]
2831: (8)                   arr_ratio = nr / nc
2832: (4)               else:
2833: (8)                   arr_ratio = arg
2834: (4)               fig_height = mpl.rcParams['figure.figsize'][1]
2835: (4)               newsize = np.array((fig_height / arr_ratio, fig_height))
2836: (4)               newsize /= min(1.0, *(newsize / figsize_min))
2837: (4)               newsize /= max(1.0, *(newsize / figsize_max))
2838: (4)               newsize = np.clip(newsize, figsize_min, figsize_max)
2839: (4)               return newsize
```

File 39 - font_manager.py:

```
1: (0)              """
2: (0)              A module for finding, managing, and using fonts across platforms.
3: (0)              This module provides a single `FontManager` instance, ``fontManager``, that
can
4: (0)              be shared across backends and platforms.  The `findfont`
5: (0)              function returns the best TrueType (TTF) font file in the local or
6: (0)              system font path that matches the specified `FontProperties`
7: (0)              instance.  The `FontManager` also handles Adobe Font Metrics
8: (0)              (AFM) font files for use by the PostScript backend.
9: (0)              The `FontManager.addfont` function adds a custom font from a file without
10: (0)             installing it into your operating system.
11: (0)             The design is based on the `W3C Cascading Style Sheet, Level 1 (CSS1)
12: (0)             font specification <http://www.w3.org/TR/1998/REC-CSS2-19980512/>`_.
13: (0)             Future versions may implement the Level 2 or 2.1 specifications.
14: (0)             """
15: (0)             from __future__ import annotations
16: (0)             from base64 import b64encode
17: (0)             from collections import namedtuple
18: (0)             import copy
19: (0)             import dataclasses
20: (0)             from functools import lru_cache
21: (0)             from io import BytesIO
22: (0)             import json
23: (0)             import logging
24: (0)             from numbers import Number
25: (0)             import os
26: (0)             from pathlib import Path
27: (0)             import plistlib
28: (0)             import re
29: (0)             import subprocess
30: (0)             import sys
31: (0)             import threading
32: (0)             import matplotlib as mpl
33: (0)             from matplotlib import _api, _afm, cbook, ft2font
34: (0)             from matplotlib._fontconfig_pattern import (
35: (4)                 parse_fontconfig_pattern, generate_fontconfig_pattern)
36: (0)             from matplotlib.rcsetup import _validators
37: (0)             _log = logging.getLogger(__name__)
38: (0)             font_scalings = {
39: (4)                 'xx-small': 0.579,
40: (4)                 'x-small':  0.694,
41: (4)                 'small':    0.833,
42: (4)                 'medium':   1.0,
43: (4)                 'large':    1.200,
44: (4)                 'x-large':  1.440,
45: (4)                 'xx-large': 1.728,
46: (4)                 'larger':   1.2,
47: (4)                 'smaller':  0.833,
48: (4)                 None:       1.0,
49: (0)             }
50: (0)             stretch_dict = {
```

```
(4)                 'ultra-condensed': 100,
(4)                 'extra-condensed': 200,
(4)                 'condensed':       300,
(4)                 'semi-condensed':  400,
(4)                 'normal':          500,
(4)                 'semi-expanded':   600,
(4)                 'semi-extended':   600,
(4)                 'expanded':        700,
(4)                 'extended':        700,
(4)                 'extra-expanded':  800,
(4)                 'extra-extended':  800,
(4)                 'ultra-expanded':  900,
(4)                 'ultra-extended':  900,
(0)             }
(0)             weight_dict = {
(4)                 'ultralight': 100,
(4)                 'light':      200,
(4)                 'normal':     400,
(4)                 'regular':    400,
(4)                 'book':       400,
(4)                 'medium':     500,
(4)                 'roman':      500,
(4)                 'semibold':   600,
(4)                 'demibold':   600,
(4)                 'demi':       600,
(4)                 'bold':       700,
(4)                 'heavy':      800,
(4)                 'extra bold': 800,
(4)                 'black':      900,
(0)             }
(0)             _weight_regexes = [
(4)                 ("thin", 100),
(4)                 ("extralight", 200),
(4)                 ("ultralight", 200),
(4)                 ("demilight", 350),
(4)                 ("semilight", 350),
(4)                 ("light", 300),  # Needs to come *after* demi/semilight!
(4)                 ("book", 380),
(4)                 ("regular", 400),
(4)                 ("normal", 400),
(4)                 ("medium", 500),
(4)                 ("demibold", 600),
(4)                 ("demi", 600),
(4)                 ("semibold", 600),
(4)                 ("extrabold", 800),
(4)                 ("superbold", 800),
(4)                 ("ultrabold", 800),
(4)                 ("bold", 700),  # Needs to come *after* extra/super/ultrabold!
(4)                 ("ultrablack", 1000),
(4)                 ("superblack", 1000),
(4)                 ("extrablack", 1000),
(4)                 (r"\bultra", 1000),
(4)                 ("black", 900),  # Needs to come *after* ultra/super/extrablack!
(4)                 ("heavy", 900),
(0)             ]
(0)             font_family_aliases = {
(4)                 'serif',
(4)                 'sans-serif',
(4)                 'sans serif',
(4)                 'cursive',
(4)                 'fantasy',
(4)                 'monospace',
(4)                 'sans',
(0)             }
(0)             _ExceptionProxy = namedtuple('_ExceptionProxy', ['klass', 'message'])
(0)             try:
(4)                 _HOME = Path.home()
(0)             except Exception:  # Exceptions thrown by home() are not specified...
(4)                 _HOME = Path(os.devnull)  # Just an arbitrary path with no children.
```

```
120: (0)            MSFolders = \
121: (4)                r'Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders'
122: (0)            MSFontDirectories = [
123: (4)                r'SOFTWARE\Microsoft\Windows NT\CurrentVersion\Fonts',
124: (4)                r'SOFTWARE\Microsoft\Windows\CurrentVersion\Fonts']
125: (0)            MSUserFontDirectories = [
126: (4)                str(_HOME / 'AppData/Local/Microsoft/Windows/Fonts'),
127: (4)                str(_HOME / 'AppData/Roaming/Microsoft/Windows/Fonts'),
128: (0)            ]
129: (0)            X11FontDirectories = [
130: (4)                "/usr/X11R6/lib/X11/fonts/TTF/",
131: (4)                "/usr/X11/lib/X11/fonts",
132: (4)                "/usr/share/fonts/",
133: (4)                "/usr/local/share/fonts/",
134: (4)                "/usr/lib/openoffice/share/fonts/truetype/",
135: (4)                str((Path(os.environ.get('XDG_DATA_HOME') or _HOME / ".local/share"))
136: (8)                    / "fonts"),
137: (4)                str(_HOME / ".fonts"),
138: (0)            ]
139: (0)            OSXFontDirectories = [
140: (4)                "/Library/Fonts/",
141: (4)                "/Network/Library/Fonts/",
142: (4)                "/System/Library/Fonts/",
143: (4)                "/opt/local/share/fonts",
144: (4)                str(_HOME / "Library/Fonts"),
145: (0)            ]
146: (0)            def get_fontext_synonyms(fontext):
147: (4)                """
148: (4)                Return a list of file extensions that are synonyms for
149: (4)                the given file extension *fileext*.
150: (4)                """
151: (4)                return {
152: (8)                    'afm': ['afm'],
153: (8)                    'otf': ['otf', 'ttc', 'ttf'],
154: (8)                    'ttc': ['otf', 'ttc', 'ttf'],
155: (8)                    'ttf': ['otf', 'ttc', 'ttf'],
156: (4)                }[fontext]
157: (0)            def list_fonts(directory, extensions):
158: (4)                """
159: (4)                Return a list of all fonts matching any of the extensions, found
160: (4)                recursively under the directory.
161: (4)                """
162: (4)                extensions = ["." + ext for ext in extensions]
163: (4)                return [os.path.join(dirpath, filename)
164: (12)                        for dirpath, _, filenames in os.walk(directory)
165: (12)                        for filename in filenames
166: (12)                        if Path(filename).suffix.lower() in extensions]
167: (0)            def win32FontDirectory():
168: (4)                r"""
169: (4)                Return the user-specified font directory for Win32.  This is
170: (4)                looked up from the registry key ::
171: (6)
\\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders\Fonts
172: (4)                If the key is not found, ``%WINDIR%\Fonts`` will be returned.
173: (4)                """  # noqa: E501
174: (4)                import winreg
175: (4)                try:
176: (8)                    with winreg.OpenKey(winreg.HKEY_CURRENT_USER, MSFolders) as user:
177: (12)                        return winreg.QueryValueEx(user, 'Fonts')[0]
178: (4)                except OSError:
179: (8)                    return os.path.join(os.environ['WINDIR'], 'Fonts')
180: (0)            def _get_win32_installed_fonts():
181: (4)                """List the font paths known to the Windows registry."""
182: (4)                import winreg
183: (4)                items = set()
184: (4)                for domain, base_dirs in [
185: (12)                        (winreg.HKEY_LOCAL_MACHINE, [win32FontDirectory()]),  # System.
186: (12)                        (winreg.HKEY_CURRENT_USER, MSUserFontDirectories),  # User.
187: (4)                ]:
```

```
188: (8)                    for base_dir in base_dirs:
189: (12)                       for reg_path in MSFontDirectories:
190: (16)                           try:
191: (20)                               with winreg.OpenKey(domain, reg_path) as local:
192: (24)                                   for j in range(winreg.QueryInfoKey(local)[1]):
193: (28)                                       key, value, tp = winreg.EnumValue(local, j)
194: (28)                                       if not isinstance(value, str):
195: (32)                                           continue
196: (28)                                       try:
197: (32)                                           path = Path(base_dir, value).resolve()
198: (28)                                       except RuntimeError:
199: (32)                                           continue
200: (28)                                       items.add(path)
201: (16)                           except (OSError, MemoryError):
202: (20)                               continue
203: (4)                return items
204: (0)            @lru_cache
205: (0)            def _get_fontconfig_fonts():
206: (4)                """Cache and list the font paths known to ``fc-list``."""
207: (4)                try:
208: (8)                    if b'--format' not in subprocess.check_output(['fc-list', '--help']):
209: (12)                       _log.warning(  # fontconfig 2.7 implemented --format.
210: (16)                           'Matplotlib needs fontconfig>=2.7 to query system fonts.')
211: (12)                       return []
212: (8)                    out = subprocess.check_output(['fc-list', '--format=%{file}\\n'])
213: (4)                except (OSError, subprocess.CalledProcessError):
214: (8)                    return []
215: (4)                return [Path(os.fsdecode(fname)) for fname in out.split(b'\n')]
216: (0)            @lru_cache
217: (0)            def _get_macos_fonts():
218: (4)                """Cache and list the font paths known to ``system_profiler
SPFontsDataType``."""
219: (4)                try:
220: (8)                    d, = plistlib.loads(
221: (12)                       subprocess.check_output(["system_profiler", "-xml",
"SPFontsDataType"]))
222: (4)                except (OSError, subprocess.CalledProcessError,
plistlib.InvalidFileException):
223: (8)                    return []
224: (4)                return [Path(entry["path"]) for entry in d["_items"]]
225: (0)            def findSystemFonts(fontpaths=None, fontext='ttf'):
226: (4)                """
227: (4)                Search for fonts in the specified font paths.  If no paths are
228: (4)                given, will use a standard set of system paths, as well as the
229: (4)                list of fonts tracked by fontconfig if fontconfig is installed and
230: (4)                available.  A list of TrueType fonts are returned by default with
231: (4)                AFM fonts as an option.
232: (4)                """
233: (4)                fontfiles = set()
234: (4)                fontexts = get_fontext_synonyms(fontext)
235: (4)                if fontpaths is None:
236: (8)                    if sys.platform == 'win32':
237: (12)                       installed_fonts = _get_win32_installed_fonts()
238: (12)                       fontpaths = []
239: (8)                    else:
240: (12)                       installed_fonts = _get_fontconfig_fonts()
241: (12)                       if sys.platform == 'darwin':
242: (16)                           installed_fonts += _get_macos_fonts()
243: (16)                           fontpaths = [*X11FontDirectories, *OSXFontDirectories]
244: (12)                       else:
245: (16)                           fontpaths = X11FontDirectories
246: (8)                    fontfiles.update(str(path) for path in installed_fonts
247: (25)                                     if path.suffix.lower()[1:] in fontexts)
248: (4)                elif isinstance(fontpaths, str):
249: (8)                    fontpaths = [fontpaths]
250: (4)                for path in fontpaths:
251: (8)                    fontfiles.update(map(os.path.abspath, list_fonts(path, fontexts)))
252: (4)                return [fname for fname in fontfiles if os.path.exists(fname)]
253: (0)            @dataclasses.dataclass(frozen=True)
```

```
(0)            class FontEntry:
(4)                """
(4)                A class for storing Font properties.
(4)                It is used when populating the font lookup dictionary.
(4)                """
(4)                fname: str = ''
(4)                name: str = ''
(4)                style: str = 'normal'
(4)                variant: str = 'normal'
(4)                weight: str | int = 'normal'
(4)                stretch: str = 'normal'
(4)                size: str = 'medium'
(4)                def _repr_html_(self) -> str:
(8)                    png_stream = self._repr_png_()
(8)                    png_b64 = b64encode(png_stream).decode()
(8)                    return f"<img src=\" {png_b64}\" />"
(4)                def _repr_png_(self) -> bytes:
(8)                    from matplotlib.figure import Figure  # Circular import.
(8)                    fig = Figure()
(8)                    font_path = Path(self.fname) if self.fname != '' else None
(8)                    fig.text(0, 0, self.name, font=font_path)
(8)                    with BytesIO() as buf:
(12)                       fig.savefig(buf, bbox_inches='tight', transparent=True)
(12)                       return buf.getvalue()
(0)            def ttfFontProperty(font):
(4)                """
(4)                Extract information from a TrueType font file.
(4)                Parameters
(4)                ----------
(4)                font : `.FT2Font`
(8)                    The TrueType font file from which information will be extracted.
(4)                Returns
(4)                -------
(4)                `FontEntry`
(8)                    The extracted font properties.
(4)                """
(4)                name = font.family_name
(4)                sfnt = font.get_sfnt()
(4)                mac_key = (1,  # platform: macintosh
(15)                          0,  # id: roman
(15)                          0)  # langid: english
(4)                ms_key = (3,  # platform: microsoft
(14)                         1,  # id: unicode_cs
(14)                         0x0409)  # langid: english_united_states
(4)                sfnt2 = (sfnt.get((*mac_key, 2), b'').decode('latin-1').lower() or
(13)                        sfnt.get((*ms_key, 2), b'').decode('utf_16_be').lower())
(4)                sfnt4 = (sfnt.get((*mac_key, 4), b'').decode('latin-1').lower() or
(13)                        sfnt.get((*ms_key, 4), b'').decode('utf_16_be').lower())
(4)                if sfnt4.find('oblique') >= 0:
(8)                    style = 'oblique'
(4)                elif sfnt4.find('italic') >= 0:
(8)                    style = 'italic'
(4)                elif sfnt2.find('regular') >= 0:
(8)                    style = 'normal'
(4)                elif font.style_flags & ft2font.ITALIC:
(8)                    style = 'italic'
(4)                else:
(8)                    style = 'normal'
(4)                if name.lower() in ['capitals', 'small-caps']:
(8)                    variant = 'small-caps'
(4)                else:
(8)                    variant = 'normal'
(4)                wws_subfamily = 22
(4)                typographic_subfamily = 16
(4)                font_subfamily = 2
(4)                styles = [
(8)                    sfnt.get((*mac_key, wws_subfamily), b'').decode('latin-1'),
(8)                    sfnt.get((*mac_key, typographic_subfamily), b'').decode('latin-1'),
(8)                    sfnt.get((*mac_key, font_subfamily), b'').decode('latin-1'),
```

```
(8)                     sfnt.get((*ms_key, wws_subfamily), b'').decode('utf-16-be'),
(8)                     sfnt.get((*ms_key, typographic_subfamily), b'').decode('utf-16-be'),
(8)                     sfnt.get((*ms_key, font_subfamily), b'').decode('utf-16-be'),
(4)                 ]
(4)                 styles = [*filter(None, styles)] or [font.style_name]
(4)                 def get_weight():  # From fontconfig's FcFreeTypeQueryFaceInternal.
(8)                     os2 = font.get_sfnt_table("OS/2")
(8)                     if os2 and os2["version"] != 0xffff:
(12)                        return os2["usWeightClass"]
(8)                     try:
(12)                        ps_font_info_weight = (
(16)                            font.get_ps_font_info()["weight"].replace(" ", "") or "")
(8)                     except ValueError:
(12)                        pass
(8)                     else:
(12)                        for regex, weight in _weight_regexes:
(16)                            if re.fullmatch(regex, ps_font_info_weight, re.I):
(20)                                return weight
(8)                     for style in styles:
(12)                        style = style.replace(" ", "")
(12)                        for regex, weight in _weight_regexes:
(16)                            if re.search(regex, style, re.I):
(20)                                return weight
(8)                     if font.style_flags & ft2font.BOLD:
(12)                        return 700  # "bold"
(8)                     return 500  # "medium", not "regular"!
(4)                 weight = int(get_weight())
(4)                 if any(word in sfnt4 for word in ['narrow', 'condensed', 'cond']):
(8)                     stretch = 'condensed'
(4)                 elif 'demi cond' in sfnt4:
(8)                     stretch = 'semi-condensed'
(4)                 elif any(word in sfnt4 for word in ['wide', 'expanded', 'extended']):
(8)                     stretch = 'expanded'
(4)                 else:
(8)                     stretch = 'normal'
(4)                 if not font.scalable:
(8)                     raise NotImplementedError("Non-scalable fonts are not supported")
(4)                 size = 'scalable'
(4)                 return FontEntry(font.fname, name, style, variant, weight, stretch, size)
(0)             def afmFontProperty(fontpath, font):
(4)                 """
(4)                 Extract information from an AFM font file.
(4)                 Parameters
(4)                 ----------
(4)                 fontpath : str
(8)                     The filename corresponding to *font*.
(4)                 font : AFM
(8)                     The AFM font file from which information will be extracted.
(4)                 Returns
(4)                 -------
(4)                 `FontEntry`
(8)                     The extracted font properties.
(4)                 """
(4)                 name = font.get_familyname()
(4)                 fontname = font.get_fontname().lower()
(4)                 if font.get_angle() != 0 or 'italic' in name.lower():
(8)                     style = 'italic'
(4)                 elif 'oblique' in name.lower():
(8)                     style = 'oblique'
(4)                 else:
(8)                     style = 'normal'
(4)                 if name.lower() in ['capitals', 'small-caps']:
(8)                     variant = 'small-caps'
(4)                 else:
(8)                     variant = 'normal'
(4)                 weight = font.get_weight().lower()
(4)                 if weight not in weight_dict:
(8)                     weight = 'normal'
(4)                 if 'demi cond' in fontname:
```

```
(8)                 stretch = 'semi-condensed'
(4)             elif any(word in fontname for word in ['narrow', 'cond']):
(8)                 stretch = 'condensed'
(4)             elif any(word in fontname for word in ['wide', 'expanded', 'extended']):
(8)                 stretch = 'expanded'
(4)             else:
(8)                 stretch = 'normal'
(4)             size = 'scalable'
(4)             return FontEntry(fontpath, name, style, variant, weight, stretch, size)
(0)         class FontProperties:
(4)             """
(4)             A class for storing and manipulating font properties.
(4)             The font properties are the six properties described in the
(4)             `W3C Cascading Style Sheet, Level 1
(4)             <http://www.w3.org/TR/1998/REC-CSS2-19980512/>`_ font
(4)             specification and *math_fontfamily* for math fonts:
(4)             - family: A list of font names in decreasing order of priority.
(6)               The items may include a generic font family name, either 'sans-serif',
(6)               'serif', 'cursive', 'fantasy', or 'monospace'.  In that case, the actual
(6)               font to be used will be looked up from the associated rcParam during the
(6)               search process in `.findfont`. Default: :rc:`font.family`
(4)             - style: Either 'normal', 'italic' or 'oblique'.
(6)               Default: :rc:`font.style`
(4)             - variant: Either 'normal' or 'small-caps'.
(6)               Default: :rc:`font.variant`
(4)             - stretch: A numeric value in the range 0-1000 or one of
(6)               'ultra-condensed', 'extra-condensed', 'condensed',
(6)               'semi-condensed', 'normal', 'semi-expanded', 'expanded',
(6)               'extra-expanded' or 'ultra-expanded'. Default: :rc:`font.stretch`
(4)             - weight: A numeric value in the range 0-1000 or one of
(6)               'ultralight', 'light', 'normal', 'regular', 'book', 'medium',
(6)               'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy',
(6)               'extra bold', 'black'. Default: :rc:`font.weight`
(4)             - size: Either a relative value of 'xx-small', 'x-small',
(6)               'small', 'medium', 'large', 'x-large', 'xx-large' or an
(6)               absolute font size, e.g., 10. Default: :rc:`font.size`
(4)             - math_fontfamily: The family of fonts used to render math text.
(6)               Supported values are: 'dejavusans', 'dejavuserif', 'cm',
(6)               'stix', 'stixsans' and 'custom'. Default: :rc:`mathtext.fontset`
(4)             Alternatively, a font may be specified using the absolute path to a font
(4)             file, by using the *fname* kwarg.  However, in this case, it is typically
(4)             simpler to just pass the path (as a `pathlib.Path`, not a `str`) to the
(4)             *font* kwarg of the `.Text` object.
(4)             The preferred usage of font sizes is to use the relative values,
(4)             e.g.,  'large', instead of absolute font sizes, e.g., 12.  This
(4)             approach allows all text sizes to be made larger or smaller based
(4)             on the font manager's default font size.
(4)             This class will also accept a fontconfig_ pattern_, if it is the only
(4)             argument provided.  This support does not depend on fontconfig; we are
(4)             merely borrowing its pattern syntax for use here.
(4)             .. _fontconfig: https://www.freedesktop.org/wiki/Software/fontconfig/
(4)             .. _pattern:
(7)                https://www.freedesktop.org/software/fontconfig/fontconfig-user.html
(4)             Note that Matplotlib's internal font manager and fontconfig use a
(4)             different algorithm to lookup fonts, so the results of the same pattern
(4)             may be different in Matplotlib than in other applications that use
(4)             fontconfig.
(4)             """
(4)             def __init__(self, family=None, style=None, variant=None, weight=None,
(17)                         stretch=None, size=None,
(17)                         fname=None,  # if set, it's a hardcoded filename to use
(17)                         math_fontfamily=None):
(8)                 self.set_family(family)
(8)                 self.set_style(style)
(8)                 self.set_variant(variant)
(8)                 self.set_weight(weight)
(8)                 self.set_stretch(stretch)
(8)                 self.set_file(fname)
(8)                 self.set_size(size)
```

```
461: (8)                    self.set_math_fontfamily(math_fontfamily)
462: (8)                    if (isinstance(family, str)
463: (16)                           and style is None and variant is None and weight is None
464: (16)                           and stretch is None and size is None and fname is None):
465: (12)                       self.set_fontconfig_pattern(family)
466: (4)                @classmethod
467: (4)                def _from_any(cls, arg):
468: (8)                    """
469: (8)                    Generic constructor which can build a `.FontProperties` from any of
the
470: (8)                    following:
471: (8)                    - a `.FontProperties`: it is passed through as is;
472: (8)                    - `None`: a `.FontProperties` using rc values is used;
473: (8)                    - an `os.PathLike`: it is used as path to the font file;
474: (8)                    - a `str`: it is parsed as a fontconfig pattern;
475: (8)                    - a `dict`: it is passed as ``**kwargs`` to `.FontProperties`.
476: (8)                    """
477: (8)                    if arg is None:
478: (12)                       return cls()
479: (8)                    elif isinstance(arg, cls):
480: (12)                       return arg
481: (8)                    elif isinstance(arg, os.PathLike):
482: (12)                       return cls(fname=arg)
483: (8)                    elif isinstance(arg, str):
484: (12)                       return cls(arg)
485: (8)                    else:
486: (12)                       return cls(**arg)
487: (4)                def __hash__(self):
488: (8)                    l = (tuple(self.get_family()),
489: (13)                         self.get_slant(),
490: (13)                         self.get_variant(),
491: (13)                         self.get_weight(),
492: (13)                         self.get_stretch(),
493: (13)                         self.get_size(),
494: (13)                         self.get_file(),
495: (13)                         self.get_math_fontfamily())
496: (8)                    return hash(l)
497: (4)                def __eq__(self, other):
498: (8)                    return hash(self) == hash(other)
499: (4)                def __str__(self):
500: (8)                    return self.get_fontconfig_pattern()
501: (4)                def get_family(self):
502: (8)                    """
503: (8)                    Return a list of individual font family names or generic family names.
504: (8)                    The font families or generic font families (which will be resolved
505: (8)                    from their respective rcParams when searching for a matching font) in
506: (8)                    the order of preference.
507: (8)                    """
508: (8)                    return self._family
509: (4)                def get_name(self):
510: (8)                    """
511: (8)                    Return the name of the font that best matches the font properties.
512: (8)                    """
513: (8)                    return get_font(findfont(self)).family_name
514: (4)                def get_style(self):
515: (8)                    """
516: (8)                    Return the font style.  Values are: 'normal', 'italic' or 'oblique'.
517: (8)                    """
518: (8)                    return self._slant
519: (4)                def get_variant(self):
520: (8)                    """
521: (8)                    Return the font variant.  Values are: 'normal' or 'small-caps'.
522: (8)                    """
523: (8)                    return self._variant
524: (4)                def get_weight(self):
525: (8)                    """
526: (8)                    Set the font weight.  Options are: A numeric value in the
527: (8)                    range 0-1000 or one of 'light', 'normal', 'regular', 'book',
528: (8)                    'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold',
```

```
529: (8)                    'heavy', 'extra bold', 'black'
530: (8)                    """
531: (8)                    return self._weight
532: (4)                def get_stretch(self):
533: (8)                    """
534: (8)                    Return the font stretch or width.  Options are: 'ultra-condensed',
535: (8)                    'extra-condensed', 'condensed', 'semi-condensed', 'normal',
536: (8)                    'semi-expanded', 'expanded', 'extra-expanded', 'ultra-expanded'.
537: (8)                    """
538: (8)                    return self._stretch
539: (4)                def get_size(self):
540: (8)                    """
541: (8)                    Return the font size.
542: (8)                    """
543: (8)                    return self._size
544: (4)                def get_file(self):
545: (8)                    """
546: (8)                    Return the filename of the associated font.
547: (8)                    """
548: (8)                    return self._file
549: (4)                def get_fontconfig_pattern(self):
550: (8)                    """
551: (8)                    Get a fontconfig_ pattern_ suitable for looking up the font as
552: (8)                    specified with fontconfig's ``fc-match`` utility.
553: (8)                    This support does not depend on fontconfig; we are merely borrowing
its
554: (8)                    pattern syntax for use here.
555: (8)                    """
556: (8)                    return generate_fontconfig_pattern(self)
557: (4)                def set_family(self, family):
558: (8)                    """
559: (8)                    Change the font family.  Can be either an alias (generic name
560: (8)                    is CSS parlance), such as: 'serif', 'sans-serif', 'cursive',
561: (8)                    'fantasy', or 'monospace', a real font name or a list of real
562: (8)                    font names.  Real font names are not supported when
563: (8)                    :rc:`text.usetex` is `True`. Default: :rc:`font.family`
564: (8)                    """
565: (8)                    if family is None:
566: (12)                       family = mpl.rcParams['font.family']
567: (8)                    if isinstance(family, str):
568: (12)                       family = [family]
569: (8)                    self._family = family
570: (4)                def set_style(self, style):
571: (8)                    """
572: (8)                    Set the font style.
573: (8)                    Parameters
574: (8)                    ----------
575: (8)                    style : {'normal', 'italic', 'oblique'}, default: :rc:`font.style`
576: (8)                    """
577: (8)                    if style is None:
578: (12)                       style = mpl.rcParams['font.style']
579: (8)                    _api.check_in_list(['normal', 'italic', 'oblique'], style=style)
580: (8)                    self._slant = style
581: (4)                def set_variant(self, variant):
582: (8)                    """
583: (8)                    Set the font variant.
584: (8)                    Parameters
585: (8)                    ----------
586: (8)                    variant : {'normal', 'small-caps'}, default: :rc:`font.variant`
587: (8)                    """
588: (8)                    if variant is None:
589: (12)                       variant = mpl.rcParams['font.variant']
590: (8)                    _api.check_in_list(['normal', 'small-caps'], variant=variant)
591: (8)                    self._variant = variant
592: (4)                def set_weight(self, weight):
593: (8)                    """
594: (8)                    Set the font weight.
595: (8)                    Parameters
596: (8)                    ----------
```

```
(8)                 weight : int or {'ultralight', 'light', 'normal', 'regular', 'book', \
(0)         'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', \
(0)         'extra bold', 'black'}, default: :rc:`font.weight`
(12)                    If int, must be in the range  0-1000.
(8)                 """
(8)                 if weight is None:
(12)                    weight = mpl.rcParams['font.weight']
(8)                 if weight in weight_dict:
(12)                    self._weight = weight
(12)                    return
(8)                 try:
(12)                    weight = int(weight)
(8)                 except ValueError:
(12)                    pass
(8)                 else:
(12)                    if 0 <= weight <= 1000:
(16)                        self._weight = weight
(16)                        return
(8)                 raise ValueError(f"{weight=} is invalid")
(4)             def set_stretch(self, stretch):
(8)                 """
(8)                 Set the font stretch or width.
(8)                 Parameters
(8)                 ----------
(8)                 stretch : int or {'ultra-condensed', 'extra-condensed', 'condensed', \
(0)         'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded', \
(0)         'ultra-expanded'}, default: :rc:`font.stretch`
(12)                    If int, must be in the range  0-1000.
(8)                 """
(8)                 if stretch is None:
(12)                    stretch = mpl.rcParams['font.stretch']
(8)                 if stretch in stretch_dict:
(12)                    self._stretch = stretch
(12)                    return
(8)                 try:
(12)                    stretch = int(stretch)
(8)                 except ValueError:
(12)                    pass
(8)                 else:
(12)                    if 0 <= stretch <= 1000:
(16)                        self._stretch = stretch
(16)                        return
(8)                 raise ValueError(f"{stretch=} is invalid")
(4)             def set_size(self, size):
(8)                 """
(8)                 Set the font size.
(8)                 Parameters
(8)                 ----------
(8)                 size : float or {'xx-small', 'x-small', 'small', 'medium', \
(0)         'large', 'x-large', 'xx-large'}, default: :rc:`font.size`
(12)                    If a float, the font size in points. The string values denote
(12)                    sizes relative to the default font size.
(8)                 """
(8)                 if size is None:
(12)                    size = mpl.rcParams['font.size']
(8)                 try:
(12)                    size = float(size)
(8)                 except ValueError:
(12)                    try:
(16)                        scale = font_scalings[size]
(12)                    except KeyError as err:
(16)                        raise ValueError(
(20)                            "Size is invalid. Valid font size are "
(20)                            + ", ".join(map(str, font_scalings))) from err
(12)                    else:
(16)                        size = scale * FontManager.get_default_size()
(8)                 if size < 1.0:
(12)                    _log.info('Fontsize %1.2f < 1.0 pt not allowed by FreeType. '
(22)                              'Setting fontsize = 1 pt', size)
```

```
666: (12)                        size = 1.0
667: (8)                    self._size = size
668: (4)                def set_file(self, file):
669: (8)                    """
670: (8)                    Set the filename of the fontfile to use.  In this case, all
671: (8)                    other properties will be ignored.
672: (8)                    """
673: (8)                    self._file = os.fspath(file) if file is not None else None
674: (4)                def set_fontconfig_pattern(self, pattern):
675: (8)                    """
676: (8)                    Set the properties by parsing a fontconfig_ *pattern*.
677: (8)                    This support does not depend on fontconfig; we are merely borrowing
its
678: (8)                    pattern syntax for use here.
679: (8)                    """
680: (8)                    for key, val in parse_fontconfig_pattern(pattern).items():
681: (12)                        if type(val) is list:
682: (16)                            getattr(self, "set_" + key)(val[0])
683: (12)                        else:
684: (16)                            getattr(self, "set_" + key)(val)
685: (4)                def get_math_fontfamily(self):
686: (8)                    """
687: (8)                    Return the name of the font family used for math text.
688: (8)                    The default font is :rc:`mathtext.fontset`.
689: (8)                    """
690: (8)                    return self._math_fontfamily
691: (4)                def set_math_fontfamily(self, fontfamily):
692: (8)                    """
693: (8)                    Set the font family for text in math mode.
694: (8)                    If not set explicitly, :rc:`mathtext.fontset` will be used.
695: (8)                    Parameters
696: (8)                    ----------
697: (8)                    fontfamily : str
698: (12)                        The name of the font family.
699: (12)                        Available font families are defined in the
700: (12)                        :ref:`default matplotlibrc file <customizing-with-matplotlibrc-
files>`.
701: (8)                    See Also
702: (8)                    --------
703: (8)                    .text.Text.get_math_fontfamily
704: (8)                    """
705: (8)                    if fontfamily is None:
706: (12)                        fontfamily = mpl.rcParams['mathtext.fontset']
707: (8)                    else:
708: (12)                        valid_fonts = _validators['mathtext.fontset'].valid.values()
709: (12)                        _api.check_in_list(valid_fonts, math_fontfamily=fontfamily)
710: (8)                    self._math_fontfamily = fontfamily
711: (4)                def copy(self):
712: (8)                    """Return a copy of self."""
713: (8)                    return copy.copy(self)
714: (4)                set_name = set_family
715: (4)                get_slant = get_style
716: (4)                set_slant = set_style
717: (4)                get_size_in_points = get_size
718: (0)            class _JSONEncoder(json.JSONEncoder):
719: (4)                def default(self, o):
720: (8)                    if isinstance(o, FontManager):
721: (12)                        return dict(o.__dict__, __class__='FontManager')
722: (8)                    elif isinstance(o, FontEntry):
723: (12)                        d = dict(o.__dict__, __class__='FontEntry')
724: (12)                        try:
725: (16)                            d["fname"] =
str(Path(d["fname"]).relative_to(mpl.get_data_path()))
726: (12)                        except ValueError:
727: (16)                            pass
728: (12)                        return d
729: (8)                    else:
730: (12)                        return super().default(o)
731: (0)            def _json_decode(o):
```

```
732: (4)                cls = o.pop('__class__', None)
733: (4)                if cls is None:
734: (8)                    return o
735: (4)                elif cls == 'FontManager':
736: (8)                    r = FontManager.__new__(FontManager)
737: (8)                    r.__dict__.update(o)
738: (8)                    return r
739: (4)                elif cls == 'FontEntry':
740: (8)                    if not os.path.isabs(o['fname']):
741: (12)                       o['fname'] = os.path.join(mpl.get_data_path(), o['fname'])
742: (8)                    r = FontEntry(**o)
743: (8)                    return r
744: (4)                else:
745: (8)                    raise ValueError("Don't know how to deserialize __class__=%s" % cls)
746: (0)            def json_dump(data, filename):
747: (4)                """
748: (4)                Dump `FontManager` *data* as JSON to the file named *filename*.
749: (4)                See Also
750: (4)                --------
751: (4)                json_load
752: (4)                Notes
753: (4)                -----
754: (4)                File paths that are children of the Matplotlib data path (typically, fonts
755: (4)                shipped with Matplotlib) are stored relative to that data path (to remain
756: (4)                valid across virtualenvs).
757: (4)                This function temporarily locks the output file to prevent multiple
758: (4)                processes from overwriting one another's output.
759: (4)                """
760: (4)                try:
761: (8)                    with cbook._lock_path(filename), open(filename, 'w') as fh:
762: (12)                       json.dump(data, fh, cls=_JSONEncoder, indent=2)
763: (4)                except OSError as e:
764: (8)                    _log.warning('Could not save font_manager cache %s', e)
765: (0)            def json_load(filename):
766: (4)                """
767: (4)                Load a `FontManager` from the JSON file named *filename*.
768: (4)                See Also
769: (4)                --------
770: (4)                json_dump
771: (4)                """
772: (4)                with open(filename) as fh:
773: (8)                    return json.load(fh, object_hook=_json_decode)
774: (0)            class FontManager:
775: (4)                """
776: (4)                On import, the `FontManager` singleton instance creates a list of ttf and
777: (4)                afm fonts and caches their `FontProperties`.  The `FontManager.findfont`
778: (4)                method does a nearest neighbor search to find the font that most closely
779: (4)                matches the specification.  If no good enough match is found, the default
780: (4)                font is returned.
781: (4)                Fonts added with the `FontManager.addfont` method will not persist in the
782: (4)                cache; therefore, `addfont` will need to be called every time Matplotlib
is
783: (4)                imported. This method should only be used if and when a font cannot be
784: (4)                installed on your operating system by other means.
785: (4)                Notes
786: (4)                -----
787: (4)                The `FontManager.addfont` method must be called on the global
`FontManager`
788: (4)                instance.
789: (4)                Example usage::
790: (8)                    import matplotlib.pyplot as plt
791: (8)                    from matplotlib import font_manager
792: (8)                    font_dirs = ["/resources/fonts"]  # The path to the custom font file.
793: (8)                    font_files = font_manager.findSystemFonts(fontpaths=font_dirs)
794: (8)                    for font_file in font_files:
795: (12)                       font_manager.fontManager.addfont(font_file)
796: (4)                """
797: (4)                __version__ = 390
798: (4)                def __init__(self, size=None, weight='normal'):
```

```
799: (8)                    self._version = self.__version__
800: (8)                    self.__default_weight = weight
801: (8)                    self.default_size = size
802: (8)                    paths = [cbook._get_data_path('fonts', subdir)
803: (17)                            for subdir in ['ttf', 'afm', 'pdfcorefonts']]
804: (8)                    _log.debug('font search path %s', paths)
805: (8)                    self.defaultFamily = {
806: (12)                       'ttf': 'DejaVu Sans',
807: (12)                       'afm': 'Helvetica'}
808: (8)                    self.afmlist = []
809: (8)                    self.ttflist = []
810: (8)                    timer = threading.Timer(5, lambda: _log.warning(
811: (12)                       'Matplotlib is building the font cache; this may take a moment.'))
812: (8)                    timer.start()
813: (8)                    try:
814: (12)                       for fontext in ["afm", "ttf"]:
815: (16)                           for path in [*findSystemFonts(paths, fontext=fontext),
816: (29)                                        *findSystemFonts(fontext=fontext)]:
817: (20)                               try:
818: (24)                                   self.addfont(path)
819: (20)                               except OSError as exc:
820: (24)                                   _log.info("Failed to open font file %s: %s", path,
exc)
821: (20)                               except Exception as exc:
822: (24)                                   _log.info("Failed to extract font properties from %s:
"
823: (34)                                           "%s", path, exc)
824: (8)                    finally:
825: (12)                       timer.cancel()
826: (4)                def addfont(self, path):
827: (8)                    """
828: (8)                    Cache the properties of the font at *path* to make it available to the
829: (8)                    `FontManager`.  The type of font is inferred from the path suffix.
830: (8)                    Parameters
831: (8)                    ----------
832: (8)                    path : str or path-like
833: (8)                    Notes
834: (8)                    -----
835: (8)                    This method is useful for adding a custom font without installing it
in
836: (8)                    your operating system. See the `FontManager` singleton instance for
837: (8)                    usage and caveats about this function.
838: (8)                    """
839: (8)                    path = os.fsdecode(path)
840: (8)                    if Path(path).suffix.lower() == ".afm":
841: (12)                       with open(path, "rb") as fh:
842: (16)                           font = _afm.AFM(fh)
843: (12)                       prop = afmFontProperty(path, font)
844: (12)                       self.afmlist.append(prop)
845: (8)                    else:
846: (12)                       font = ft2font.FT2Font(path)
847: (12)                       prop = ttfFontProperty(font)
848: (12)                       self.ttflist.append(prop)
849: (8)                    self._findfont_cached.cache_clear()
850: (4)                @property
851: (4)                def defaultFont(self):
852: (8)                    return {ext: self.findfont(family, fontext=ext)
853: (16)                           for ext, family in self.defaultFamily.items()}
854: (4)                def get_default_weight(self):
855: (8)                    """
856: (8)                    Return the default font weight.
857: (8)                    """
858: (8)                    return self.__default_weight
859: (4)                @staticmethod
860: (4)                def get_default_size():
861: (8)                    """
862: (8)                    Return the default font size.
863: (8)                    """
864: (8)                    return mpl.rcParams['font.size']
```

```
(4)     def set_default_weight(self, weight):
(8)         """
(8)         Set the default font weight.  The initial value is 'normal'.
(8)         """
(8)         self.__default_weight = weight
(4)     @staticmethod
(4)     def _expand_aliases(family):
(8)         if family in ('sans', 'sans serif'):
(12)            family = 'sans-serif'
(8)         return mpl.rcParams['font.' + family]
(4)     def score_family(self, families, family2):
(8)         """
(8)         Return a match score between the list of font families in
(8)         *families* and the font family name *family2*.
(8)         An exact match at the head of the list returns 0.0.
(8)         A match further down the list will return between 0 and 1.
(8)         No match will return 1.0.
(8)         """
(8)         if not isinstance(families, (list, tuple)):
(12)            families = [families]
(8)         elif len(families) == 0:
(12)            return 1.0
(8)         family2 = family2.lower()
(8)         step = 1 / len(families)
(8)         for i, family1 in enumerate(families):
(12)            family1 = family1.lower()
(12)            if family1 in font_family_aliases:
(16)                options = [*map(str.lower, self._expand_aliases(family1))]
(16)                if family2 in options:
(20)                    idx = options.index(family2)
(20)                    return (i + (idx / len(options))) * step
(12)            elif family1 == family2:
(16)                return i * step
(8)         return 1.0
(4)     def score_style(self, style1, style2):
(8)         """
(8)         Return a match score between *style1* and *style2*.
(8)         An exact match returns 0.0.
(8)         A match between 'italic' and 'oblique' returns 0.1.
(8)         No match returns 1.0.
(8)         """
(8)         if style1 == style2:
(12)            return 0.0
(8)         elif (style1 in ('italic', 'oblique')
(14)               and style2 in ('italic', 'oblique')):
(12)            return 0.1
(8)         return 1.0
(4)     def score_variant(self, variant1, variant2):
(8)         """
(8)         Return a match score between *variant1* and *variant2*.
(8)         An exact match returns 0.0, otherwise 1.0.
(8)         """
(8)         if variant1 == variant2:
(12)            return 0.0
(8)         else:
(12)            return 1.0
(4)     def score_stretch(self, stretch1, stretch2):
(8)         """
(8)         Return a match score between *stretch1* and *stretch2*.
(8)         The result is the absolute value of the difference between the
(8)         CSS numeric values of *stretch1* and *stretch2*, normalized
(8)         between 0.0 and 1.0.
(8)         """
(8)         try:
(12)            stretchval1 = int(stretch1)
(8)         except ValueError:
(12)            stretchval1 = stretch_dict.get(stretch1, 500)
(8)         try:
(12)            stretchval2 = int(stretch2)
```

```
(8)                 except ValueError:
(12)                    stretchval2 = stretch_dict.get(stretch2, 500)
(8)                 return abs(stretchval1 - stretchval2) / 1000.0
(4)             def score_weight(self, weight1, weight2):
(8)                 """
(8)                 Return a match score between *weight1* and *weight2*.
(8)                 The result is 0.0 if both weight1 and weight 2 are given as strings
(8)                 and have the same value.
(8)                 Otherwise, the result is the absolute value of the difference between
(8)                 the CSS numeric values of *weight1* and *weight2*, normalized between
(8)                 0.05 and 1.0.
(8)                 """
(8)                 if cbook._str_equal(weight1, weight2):
(12)                    return 0.0
(8)                 w1 = weight1 if isinstance(weight1, Number) else weight_dict[weight1]
(8)                 w2 = weight2 if isinstance(weight2, Number) else weight_dict[weight2]
(8)                 return 0.95 * (abs(w1 - w2) / 1000) + 0.05
(4)             def score_size(self, size1, size2):
(8)                 """
(8)                 Return a match score between *size1* and *size2*.
(8)                 If *size2* (the size specified in the font file) is 'scalable', this
(8)                 function always returns 0.0, since any font size can be generated.
(8)                 Otherwise, the result is the absolute distance between *size1* and
(8)                 *size2*, normalized so that the usual range of font sizes (6pt -
(8)                 72pt) will lie between 0.0 and 1.0.
(8)                 """
(8)                 if size2 == 'scalable':
(12)                    return 0.0
(8)                 try:
(12)                    sizeval1 = float(size1)
(8)                 except ValueError:
(12)                    sizeval1 = self.default_size * font_scalings[size1]
(8)                 try:
(12)                    sizeval2 = float(size2)
(8)                 except ValueError:
(12)                    return 1.0
(8)                 return abs(sizeval1 - sizeval2) / 72
(4)             def findfont(self, prop, fontext='ttf', directory=None,
(17)                         fallback_to_default=True, rebuild_if_missing=True):
(8)                 """
(8)                 Find the path to the font file most closely matching the given font
properties.
(8)                 Parameters
(8)                 ----------
(8)                 prop : str or `~matplotlib.font_manager.FontProperties`
(12)                    The font properties to search for. This can be either a
(12)                    `.FontProperties` object or a string defining a
(12)                    `fontconfig patterns`_.
(8)                 fontext : {'ttf', 'afm'}, default: 'ttf'
(12)                    The extension of the font file:
(12)                    - 'ttf': TrueType and OpenType fonts (.ttf, .ttc, .otf)
(12)                    - 'afm': Adobe Font Metrics (.afm)
(8)                 directory : str, optional
(12)                    If given, only search this directory and its subdirectories.
(8)                 fallback_to_default : bool
(12)                    If True, will fall back to the default font family (usually
(12)                    "DejaVu Sans" or "Helvetica") if the first lookup hard-fails.
(8)                 rebuild_if_missing : bool
(12)                    Whether to rebuild the font cache and search again if the first
(12)                    match appears to point to a nonexisting font (i.e., the font cache
(12)                    contains outdated entries).
(8)                 Returns
(8)                 -------
(8)                 str
(12)                    The filename of the best matching font.
(8)                 Notes
(8)                 -----
(8)                 This performs a nearest neighbor search.  Each font is given a
(8)                 similarity score to the target font properties.  The first font with
```

```
(8)                     the highest score is returned.  If no matches below a certain
(8)                     threshold are found, the default font (usually DejaVu Sans) is
(8)                     returned.
(8)                     The result is cached, so subsequent lookups don't have to
(8)                     perform the O(n) nearest neighbor search.
(8)                     See the `W3C Cascading Style Sheet, Level 1
(8)                     <http://www.w3.org/TR/1998/REC-CSS2-19980512/>`_ documentation
(8)                     for a description of the font finding algorithm.
(8)                     .. _fontconfig patterns:
(11)                       https://www.freedesktop.org/software/fontconfig/fontconfig-
user.html
(8)                     """
(8)                     rc_params = tuple(tuple(mpl.rcParams[key]) for key in [
(12)                        "font.serif", "font.sans-serif", "font.cursive", "font.fantasy",
(12)                        "font.monospace"])
(8)                     ret = self._findfont_cached(
(12)                        prop, fontext, directory, fallback_to_default, rebuild_if_missing,
(12)                        rc_params)
(8)                     if isinstance(ret, _ExceptionProxy):
(12)                        raise ret.klass(ret.message)
(8)                     return ret
(4)                 def get_font_names(self):
(8)                     """Return the list of available fonts."""
(8)                     return list({font.name for font in self.ttflist})
(4)                 def _find_fonts_by_props(self, prop, fontext='ttf', directory=None,
(29)                                         fallback_to_default=True,
rebuild_if_missing=True):
(8)                     """
(8)                     Find the paths to the font files most closely matching the given
properties.
(8)                     Parameters
(8)                     ----------
(8)                     prop : str or `~matplotlib.font_manager.FontProperties`
(12)                        The font properties to search for. This can be either a
(12)                        `.FontProperties` object or a string defining a
(12)                        `fontconfig patterns`_.
(8)                     fontext : {'ttf', 'afm'}, default: 'ttf'
(12)                        The extension of the font file:
(12)                        - 'ttf': TrueType and OpenType fonts (.ttf, .ttc, .otf)
(12)                        - 'afm': Adobe Font Metrics (.afm)
(8)                     directory : str, optional
(12)                        If given, only search this directory and its subdirectories.
(8)                     fallback_to_default : bool
(12)                        If True, will fall back to the default font family (usually
(12)                        "DejaVu Sans" or "Helvetica") if none of the families were found.
(8)                     rebuild_if_missing : bool
(12)                        Whether to rebuild the font cache and search again if the first
(12)                        match appears to point to a nonexisting font (i.e., the font cache
(12)                        contains outdated entries).
(8)                     Returns
(8)                     -------
(8)                     list[str]
(12)                        The paths of the fonts found.
(8)                     Notes
(8)                     -----
(8)                     This is an extension/wrapper of the original findfont API, which only
(8)                     returns a single font for given font properties. Instead, this API
(8)                     returns a list of filepaths of multiple fonts which closely match the
(8)                     given font properties.  Since this internally uses the original API,
(8)                     there's no change to the logic of performing the nearest neighbor
(8)                     search.  See `findfont` for more details.
(8)                     """
(8)                     prop = FontProperties._from_any(prop)
(8)                     fpaths = []
(8)                     for family in prop.get_family():
(12)                        cprop = prop.copy()
(12)                        cprop.set_family(family)  # set current prop's family
(12)                        try:
(16)                            fpaths.append(
```

```
(20)                                self.findfont(
(24)                                    cprop, fontext, directory,
(24)                                    fallback_to_default=False,  # don't fallback to default
(24)                                    rebuild_if_missing=rebuild_if_missing,
(20)                                )
(16)                            )
(12)                        except ValueError:
(16)                            if family in font_family_aliases:
(20)                                _log.warning(
(24)                                    "findfont: Generic family %r not found because "
(24)                                    "none of the following families were found: %s",
(24)                                    family, ", ".join(self._expand_aliases(family))
(20)                                )
(16)                            else:
(20)                                _log.warning("findfont: Font family %r not found.", family)
(8)                     if not fpaths:
(12)                        if fallback_to_default:
(16)                            dfamily = self.defaultFamily[fontext]
(16)                            cprop = prop.copy()
(16)                            cprop.set_family(dfamily)
(16)                            fpaths.append(
(20)                                self.findfont(
(24)                                    cprop, fontext, directory,
(24)                                    fallback_to_default=True,
(24)                                    rebuild_if_missing=rebuild_if_missing,
(20)                                )
(16)                            )
(12)                        else:
(16)                            raise ValueError("Failed to find any font, and fallback "
(33)                                             "to the default font was disabled")
(8)                     return fpaths
(4)                 @lru_cache(1024)
(4)                 def _findfont_cached(self, prop, fontext, directory, fallback_to_default,
(25)                                     rebuild_if_missing, rc_params):
(8)                     prop = FontProperties._from_any(prop)
(8)                     fname = prop.get_file()
(8)                     if fname is not None:
(12)                        return fname
(8)                     if fontext == 'afm':
(12)                        fontlist = self.afmlist
(8)                     else:
(12)                        fontlist = self.ttflist
(8)                     best_score = 1e64
(8)                     best_font = None
(8)                     _log.debug('findfont: Matching %s.', prop)
(8)                     for font in fontlist:
(12)                        if (directory is not None and
(20)                                Path(directory) not in Path(font.fname).parents):
(16)                            continue
(12)                        score = (self.score_family(prop.get_family(), font.name) * 10
(21)                                 + self.score_style(prop.get_style(), font.style)
(21)                                 + self.score_variant(prop.get_variant(), font.variant)
(21)                                 + self.score_weight(prop.get_weight(), font.weight)
(21)                                 + self.score_stretch(prop.get_stretch(), font.stretch)
(21)                                 + self.score_size(prop.get_size(), font.size))
(12)                        _log.debug('findfont: score(%s) = %s', font, score)
(12)                        if score < best_score:
(16)                            best_score = score
(16)                            best_font = font
(12)                        if score == 0:
(16)                            break
(8)                     if best_font is None or best_score >= 10.0:
(12)                        if fallback_to_default:
(16)                            _log.warning(
(20)                                'findfont: Font family %s not found. Falling back to %s.',
(20)                                prop.get_family(), self.defaultFamily[fontext])
(16)                            for family in map(str.lower, prop.get_family()):
```

```
(20)                                if family in font_family_aliases:
(24)                                    _log.warning(
(28)                                        "findfont: Generic family %r not found because "
(28)                                        "none of the following families were found: %s",
(28)                                        family, ", ".join(self._expand_aliases(family)))
(16)                            default_prop = prop.copy()
(16)                            default_prop.set_family(self.defaultFamily[fontext])
(16)                            return self.findfont(default_prop, fontext, directory,
(37)                                                 fallback_to_default=False)
(12)                        else:
(16)                            return _ExceptionProxy(
(20)                                ValueError,
(20)                                f"Failed to find font {prop}, and fallback to the default font was "
(20)                                f"disabled"
(16)                            )
(8)                     else:
(12)                        _log.debug('findfont: Matching %s to %s (%r) with score of %f.',
(23)                                   prop, best_font.name, best_font.fname, best_score)
(12)                        result = best_font.fname
(8)                     if not os.path.isfile(result):
(12)                        if rebuild_if_missing:
(16)                            _log.info(
(20)                                'findfont: Found a missing font file.  Rebuilding cache.')
(16)                            new_fm = _load_fontmanager(try_read_cache=False)
(16)                            vars(self).update(vars(new_fm))
(16)                            return self.findfont(
(20)                                prop, fontext, directory, rebuild_if_missing=False)
(12)                        else:
(16)                            return _ExceptionProxy(ValueError, "No valid font could be found")
(8)                     return _cached_realpath(result)
(0)             @lru_cache
(0)             def is_opentype_cff_font(filename):
(4)                 """
(4)                 Return whether the given font is a Postscript Compact Font Format Font
(4)                 embedded in an OpenType wrapper.  Used by the PostScript and PDF backends
(4)                 that cannot subset these fonts.
(4)                 """
(4)                 if os.path.splitext(filename)[1].lower() == '.otf':
(8)                     with open(filename, 'rb') as fd:
(12)                        return fd.read(4) == b"OTTO"
(4)                 else:
(8)                     return False
(0)             @lru_cache(64)
(0)             def _get_font(font_filepaths, hinting_factor, *, _kerning_factor, thread_id):
(4)                 first_fontpath, *rest = font_filepaths
(4)                 return ft2font.FT2Font(
(8)                     first_fontpath, hinting_factor,
(8)                     _fallback_list=[
(12)                        ft2font.FT2Font(
(16)                            fpath, hinting_factor,
(16)                            _kerning_factor=_kerning_factor
(12)                        )
(12)                        for fpath in rest
(8)                     ],
(8)                     _kerning_factor=_kerning_factor
(4)                 )
(0)             if hasattr(os, "register_at_fork"):
(4)                 os.register_at_fork(after_in_child=_get_font.cache_clear)
(0)             @lru_cache(64)
(0)             def _cached_realpath(path):
(4)                 return os.path.realpath(path)
(0)             def get_font(font_filepaths, hinting_factor=None):
(4)                 """
(4)                 Get an `.ft2font.FT2Font` object given a list of file paths.
(4)                 Parameters
(4)                 ----------
(4)                 font_filepaths : Iterable[str, Path, bytes], str, Path, bytes
```

```
1202: (8)                   Relative or absolute paths to the font files to be used.
1203: (8)                   If a single string, bytes, or `pathlib.Path`, then it will be treated
1204: (8)                   as a list with that entry only.
1205: (8)                   If more than one filepath is passed, then the returned FT2Font object
1206: (8)                   will fall back through the fonts, in the order given, to find a needed
1207: (8)                   glyph.
1208: (4)               Returns
1209: (4)               -------
1210: (4)               `.ft2font.FT2Font`
1211: (4)               """
1212: (4)               if isinstance(font_filepaths, (str, Path, bytes)):
1213: (8)                   paths = (_cached_realpath(font_filepaths),)
1214: (4)               else:
1215: (8)                   paths = tuple(_cached_realpath(fname) for fname in font_filepaths)
1216: (4)               if hinting_factor is None:
1217: (8)                   hinting_factor = mpl.rcParams['text.hinting_factor']
1218: (4)               return _get_font(
1219: (8)                   paths,
1220: (8)                   hinting_factor,
1221: (8)                   _kerning_factor=mpl.rcParams['text.kerning_factor'],
1222: (8)                   thread_id=threading.get_ident()
1223: (4)               )
1224: (0)           def _load_fontmanager(*, try_read_cache=True):
1225: (4)               fm_path = Path(
1226: (8)                   mpl.get_cachedir(), f"fontlist-v{FontManager.__version__}.json")
1227: (4)               if try_read_cache:
1228: (8)                   try:
1229: (12)                      fm = json_load(fm_path)
1230: (8)                   except Exception:
1231: (12)                      pass
1232: (8)                   else:
1233: (12)                      if getattr(fm, "_version", object()) == FontManager.__version__:
1234: (16)                          _log.debug("Using fontManager instance from %s", fm_path)
1235: (16)                          return fm
1236: (4)               fm = FontManager()
1237: (4)               json_dump(fm, fm_path)
1238: (4)               _log.info("generated new fontManager")
1239: (4)               return fm
1240: (0)           fontManager = _load_fontmanager()
1241: (0)           findfont = fontManager.findfont
1242: (0)           get_font_names = fontManager.get_font_names
```

File 40 - gridspec.py:

```
1: (0)              r"""
2: (0)              :mod:`~matplotlib.gridspec` contains classes that help to layout multiple
3: (0)              `~.axes.Axes` in a grid-like pattern within a figure.
4: (0)              The `GridSpec` specifies the overall grid structure. Individual cells within
5: (0)              the grid are referenced by `SubplotSpec`\s.
6: (0)              Often, users need not access this module directly, and can use higher-level
7: (0)              methods like `~.pyplot.subplots`, `~.pyplot.subplot_mosaic` and
8: (0)              `~.Figure.subfigures`. See the tutorial :ref:`arranging_axes` for a guide.
9: (0)              """
10: (0)             import copy
11: (0)             import logging
12: (0)             from numbers import Integral
13: (0)             import numpy as np
14: (0)             import matplotlib as mpl
15: (0)             from matplotlib import _api, _pylab_helpers, _tight_layout
16: (0)             from matplotlib.transforms import Bbox
17: (0)             _log = logging.getLogger(__name__)
18: (0)             class GridSpecBase:
19: (4)                 """
20: (4)                 A base class of GridSpec that specifies the geometry of the grid
21: (4)                 that a subplot will be placed.
22: (4)                 """
23: (4)                 def __init__(self, nrows, ncols, height_ratios=None, width_ratios=None):
```

```
 24: (8)                    """
 25: (8)                    Parameters
 26: (8)                    ----------
 27: (8)                    nrows, ncols : int
 28: (12)                       The number of rows and columns of the grid.
 29: (8)                    width_ratios : array-like of length *ncols*, optional
 30: (12)                       Defines the relative widths of the columns. Each column gets a
 31: (12)                       relative width of ``width_ratios[i] / sum(width_ratios)``.
 32: (12)                       If not given, all columns will have the same width.
 33: (8)                    height_ratios : array-like of length *nrows*, optional
 34: (12)                       Defines the relative heights of the rows. Each row gets a
 35: (12)                       relative height of ``height_ratios[i] / sum(height_ratios)``.
 36: (12)                       If not given, all rows will have the same height.
 37: (8)                    """
 38: (8)                    if not isinstance(nrows, Integral) or nrows <= 0:
 39: (12)                       raise ValueError(
 40: (16)                           f"Number of rows must be a positive integer, not {nrows!r}")
 41: (8)                    if not isinstance(ncols, Integral) or ncols <= 0:
 42: (12)                       raise ValueError(
 43: (16)                           f"Number of columns must be a positive integer, not
{ncols!r}")
 44: (8)                    self._nrows, self._ncols = nrows, ncols
 45: (8)                    self.set_height_ratios(height_ratios)
 46: (8)                    self.set_width_ratios(width_ratios)
 47: (4)                def __repr__(self):
 48: (8)                    height_arg = (f', height_ratios={self._row_height_ratios!r}'
 49: (22)                                  if len(set(self._row_height_ratios)) != 1 else '')
 50: (8)                    width_arg = (f', width_ratios={self._col_width_ratios!r}'
 51: (21)                                 if len(set(self._col_width_ratios)) != 1 else '')
 52: (8)                    return '{clsname}({nrows}, {ncols}{optionals})'.format(
 53: (12)                       clsname=self.__class__.__name__,
 54: (12)                       nrows=self._nrows,
 55: (12)                       ncols=self._ncols,
 56: (12)                       optionals=height_arg + width_arg,
 57: (12)                       )
 58: (4)                nrows = property(lambda self: self._nrows,
 59: (21)                                doc="The number of rows in the grid.")
 60: (4)                ncols = property(lambda self: self._ncols,
 61: (21)                                doc="The number of columns in the grid.")
 62: (4)                def get_geometry(self):
 63: (8)                    """
 64: (8)                    Return a tuple containing the number of rows and columns in the grid.
 65: (8)                    """
 66: (8)                    return self._nrows, self._ncols
 67: (4)                def get_subplot_params(self, figure=None):
 68: (8)                    pass
 69: (4)                def new_subplotspec(self, loc, rowspan=1, colspan=1):
 70: (8)                    """
 71: (8)                    Create and return a `.SubplotSpec` instance.
 72: (8)                    Parameters
 73: (8)                    ----------
 74: (8)                    loc : (int, int)
 75: (12)                       The position of the subplot in the grid as
 76: (12)                       ``(row_index, column_index)``.
 77: (8)                    rowspan, colspan : int, default: 1
 78: (12)                       The number of rows and columns the subplot should span in the
grid.
 79: (8)                    """
 80: (8)                    loc1, loc2 = loc
 81: (8)                    subplotspec = self[loc1:loc1+rowspan, loc2:loc2+colspan]
 82: (8)                    return subplotspec
 83: (4)                def set_width_ratios(self, width_ratios):
 84: (8)                    """
 85: (8)                    Set the relative widths of the columns.
 86: (8)                    *width_ratios* must be of length *ncols*. Each column gets a relative
 87: (8)                    width of ``width_ratios[i] / sum(width_ratios)``.
 88: (8)                    """
 89: (8)                    if width_ratios is None:
 90: (12)                       width_ratios = [1] * self._ncols
```

```
(8)                     elif len(width_ratios) != self._ncols:
(12)                        raise ValueError('Expected the given number of width ratios to '
(29)                                         'match the number of columns of the grid')
(8)                     self._col_width_ratios = width_ratios
(4)                 def get_width_ratios(self):
(8)                     """
(8)                     Return the width ratios.
(8)                     This is *None* if no width ratios have been set explicitly.
(8)                     """
(8)                     return self._col_width_ratios
(4)                 def set_height_ratios(self, height_ratios):
(8)                     """
(8)                     Set the relative heights of the rows.
(8)                     *height_ratios* must be of length *nrows*. Each row gets a relative
(8)                     height of ``height_ratios[i] / sum(height_ratios)``.
(8)                     """
(8)                     if height_ratios is None:
(12)                        height_ratios = [1] * self._nrows
(8)                     elif len(height_ratios) != self._nrows:
(12)                        raise ValueError('Expected the given number of height ratios to '
(29)                                         'match the number of rows of the grid')
(8)                     self._row_height_ratios = height_ratios
(4)                 def get_height_ratios(self):
(8)                     """
(8)                     Return the height ratios.
(8)                     This is *None* if no height ratios have been set explicitly.
(8)                     """
(8)                     return self._row_height_ratios
(4)                 def get_grid_positions(self, fig):
(8)                     """
(8)                     Return the positions of the grid cells in figure coordinates.
(8)                     Parameters
(8)                     ----------
(8)                     fig : `~matplotlib.figure.Figure`
(12)                        The figure the grid should be applied to. The subplot parameters
(12)                        (margins and spacing between subplots) are taken from *fig*.
(8)                     Returns
(8)                     -------
(8)                     bottoms, tops, lefts, rights : array
(12)                        The bottom, top, left, right positions of the grid cells in
(12)                        figure coordinates.
(8)                     """
(8)                     nrows, ncols = self.get_geometry()
(8)                     subplot_params = self.get_subplot_params(fig)
(8)                     left = subplot_params.left
(8)                     right = subplot_params.right
(8)                     bottom = subplot_params.bottom
(8)                     top = subplot_params.top
(8)                     wspace = subplot_params.wspace
(8)                     hspace = subplot_params.hspace
(8)                     tot_width = right - left
(8)                     tot_height = top - bottom
(8)                     cell_h = tot_height / (nrows + hspace*(nrows-1))
(8)                     sep_h = hspace * cell_h
(8)                     norm = cell_h * nrows / sum(self._row_height_ratios)
(8)                     cell_heights = [r * norm for r in self._row_height_ratios]
(8)                     sep_heights = [0] + ([sep_h] * (nrows-1))
(8)                     cell_hs = np.cumsum(np.column_stack([sep_heights, cell_heights]).flat)
(8)                     cell_w = tot_width / (ncols + wspace*(ncols-1))
(8)                     sep_w = wspace * cell_w
(8)                     norm = cell_w * ncols / sum(self._col_width_ratios)
(8)                     cell_widths = [r * norm for r in self._col_width_ratios]
(8)                     sep_widths = [0] + ([sep_w] * (ncols-1))
(8)                     cell_ws = np.cumsum(np.column_stack([sep_widths, cell_widths]).flat)
(8)                     fig_tops, fig_bottoms = (top - cell_hs).reshape((-1, 2)).T
(8)                     fig_lefts, fig_rights = (left + cell_ws).reshape((-1, 2)).T
(8)                     return fig_bottoms, fig_tops, fig_lefts, fig_rights
(4)                 @staticmethod
(4)                 def _check_gridspec_exists(figure, nrows, ncols):
```

```
(8)                 """
(8)                 Check if the figure already has a gridspec with these dimensions,
(8)                 or create a new one
(8)                 """
(8)                 for ax in figure.get_axes():
(12)                    gs = ax.get_gridspec()
(12)                    if gs is not None:
(16)                        if hasattr(gs, 'get_topmost_subplotspec'):
(20)                            gs = gs.get_topmost_subplotspec().get_gridspec()
(16)                        if gs.get_geometry() == (nrows, ncols):
(20)                            return gs
(8)                 return GridSpec(nrows, ncols, figure=figure)
(4)             def __getitem__(self, key):
(8)                 """Create and return a `.SubplotSpec` instance."""
(8)                 nrows, ncols = self.get_geometry()
(8)                 def _normalize(key, size, axis):  # Includes last index.
(12)                    orig_key = key
(12)                    if isinstance(key, slice):
(16)                        start, stop, _ = key.indices(size)
(16)                        if stop > start:
(20)                            return start, stop - 1
(16)                        raise IndexError("GridSpec slice would result in no space "
(33)                                         "allocated for subplot")
(12)                    else:
(16)                        if key < 0:
(20)                            key = key + size
(16)                        if 0 <= key < size:
(20)                            return key, key
(16)                        elif axis is not None:
(20)                            raise IndexError(f"index {orig_key} is out of bounds for "
(37)                                             f"axis {axis} with size {size}")
(16)                        else:  # flat index
(20)                            raise IndexError(f"index {orig_key} is out of bounds for "
(37)                                             f"GridSpec with size {size}")
(8)                 if isinstance(key, tuple):
(12)                    try:
(16)                        k1, k2 = key
(12)                    except ValueError as err:
(16)                        raise ValueError("Unrecognized subplot spec") from err
(12)                    num1, num2 = np.ravel_multi_index(
(16)                        [_normalize(k1, nrows, 0), _normalize(k2, ncols, 1)],
(16)                        (nrows, ncols))
(8)                 else:  # Single key
(12)                    num1, num2 = _normalize(key, nrows * ncols, None)
(8)                 return SubplotSpec(self, num1, num2)
(4)             def subplots(self, *, sharex=False, sharey=False, squeeze=True,
(17)                         subplot_kw=None):
(8)                 """
(8)                 Add all subplots specified by this `GridSpec` to its parent figure.
(8)                 See `.Figure.subplots` for detailed documentation.
(8)                 """
(8)                 figure = self.figure
(8)                 if figure is None:
(12)                    raise ValueError("GridSpec.subplots() only works for GridSpecs "
(29)                                     "created with a parent figure")
(8)                 if not isinstance(sharex, str):
(12)                    sharex = "all" if sharex else "none"
(8)                 if not isinstance(sharey, str):
(12)                    sharey = "all" if sharey else "none"
(8)                 _api.check_in_list(["all", "row", "col", "none", False, True],
(27)                                   sharex=sharex, sharey=sharey)
(8)                 if subplot_kw is None:
(12)                    subplot_kw = {}
(8)                 subplot_kw = subplot_kw.copy()
(8)                 axarr = np.empty((self._nrows, self._ncols), dtype=object)
(8)                 for row in range(self._nrows):
(12)                    for col in range(self._ncols):
(16)                        shared_with = {"none": None, "all": axarr[0, 0],
(31)                                       "row": axarr[row, 0], "col": axarr[0, col]}
```

```
(16)                            subplot_kw["sharex"] = shared_with[sharex]
(16)                            subplot_kw["sharey"] = shared_with[sharey]
(16)                            axarr[row, col] = figure.add_subplot(
(20)                                self[row, col], **subplot_kw)
(8)                     if sharex in ["col", "all"]:
(12)                        for ax in axarr.flat:
(16)                            ax._label_outer_xaxis(skip_non_rectangular_axes=True)
(8)                     if sharey in ["row", "all"]:
(12)                        for ax in axarr.flat:
(16)                            ax._label_outer_yaxis(skip_non_rectangular_axes=True)
(8)                     if squeeze:
(12)                        return axarr.item() if axarr.size == 1 else axarr.squeeze()
(8)                     else:
(12)                        return axarr
(0)             class GridSpec(GridSpecBase):
(4)                 """
(4)                 A grid layout to place subplots within a figure.
(4)                 The location of the grid cells is determined in a similar way to
(4)                 `.SubplotParams` using *left*, *right*, *top*, *bottom*, *wspace*
(4)                 and *hspace*.
(4)                 Indexing a GridSpec instance returns a `.SubplotSpec`.
(4)                 """
(4)                 def __init__(self, nrows, ncols, figure=None,
(17)                             left=None, bottom=None, right=None, top=None,
(17)                             wspace=None, hspace=None,
(17)                             width_ratios=None, height_ratios=None):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     nrows, ncols : int
(12)                        The number of rows and columns of the grid.
(8)                     figure : `.Figure`, optional
(12)                        Only used for constrained layout to create a proper layoutgrid.
(8)                     left, right, top, bottom : float, optional
(12)                        Extent of the subplots as a fraction of figure width or height.
(12)                        Left cannot be larger than right, and bottom cannot be larger than
(12)                        top. If not given, the values will be inferred from a figure or
(12)                        rcParams at draw time. See also `GridSpec.get_subplot_params`.
(8)                     wspace : float, optional
(12)                        The amount of width reserved for space between subplots,
(12)                        expressed as a fraction of the average axis width.
(12)                        If not given, the values will be inferred from a figure or
(12)                        rcParams when necessary. See also `GridSpec.get_subplot_params`.
(8)                     hspace : float, optional
(12)                        The amount of height reserved for space between subplots,
(12)                        expressed as a fraction of the average axis height.
(12)                        If not given, the values will be inferred from a figure or
(12)                        rcParams when necessary. See also `GridSpec.get_subplot_params`.
(8)                     width_ratios : array-like of length *ncols*, optional
(12)                        Defines the relative widths of the columns. Each column gets a
(12)                        relative width of ``width_ratios[i] / sum(width_ratios)``.
(12)                        If not given, all columns will have the same width.
(8)                     height_ratios : array-like of length *nrows*, optional
(12)                        Defines the relative heights of the rows. Each row gets a
(12)                        relative height of ``height_ratios[i] / sum(height_ratios)``.
(12)                        If not given, all rows will have the same height.
(8)                     """
(8)                     self.left = left
(8)                     self.bottom = bottom
(8)                     self.right = right
(8)                     self.top = top
(8)                     self.wspace = wspace
(8)                     self.hspace = hspace
(8)                     self.figure = figure
(8)                     super().__init__(nrows, ncols,
(25)                                     width_ratios=width_ratios,
(25)                                     height_ratios=height_ratios)
(4)                 _AllowedKeys = ["left", "bottom", "right", "top", "wspace", "hspace"]
(4)                 def update(self, **kwargs):
```

```
(8)                 """
(8)                 Update the subplot parameters of the grid.
(8)                 Parameters that are not explicitly given are not changed. Setting a
(8)                 parameter to *None* resets it to :rc:`figure.subplot.*`.
(8)                 Parameters
(8)                 ----------
(8)                 left, right, top, bottom : float or None, optional
(12)                    Extent of the subplots as a fraction of figure width or height.
(8)                 wspace, hspace : float, optional
(12)                    Spacing between the subplots as a fraction of the average subplot
(12)                    width / height.
(8)                 """
(8)                 for k, v in kwargs.items():
(12)                    if k in self._AllowedKeys:
(16)                        setattr(self, k, v)
(12)                    else:
(16)                        raise AttributeError(f"{k} is an unknown keyword")
(8)                 for figmanager in _pylab_helpers.Gcf.figs.values():
(12)                    for ax in figmanager.canvas.figure.axes:
(16)                        if ax.get_subplotspec() is not None:
(20)                            ss = ax.get_subplotspec().get_topmost_subplotspec()
(20)                            if ss.get_gridspec() == self:
(24)                                ax._set_position(
(28)                                    ax.get_subplotspec().get_position(ax.figure))
(4)             def get_subplot_params(self, figure=None):
(8)                 """
(8)                 Return the `.SubplotParams` for the GridSpec.
(8)                 In order of precedence the values are taken from
(8)                 - non-*None* attributes of the GridSpec
(8)                 - the provided *figure*
(8)                 - :rc:`figure.subplot.*`
(8)                 Note that the ``figure`` attribute of the GridSpec is always ignored.
(8)                 """
(8)                 if figure is None:
(12)                    kw = {k: mpl.rcParams["figure.subplot."+k]
(18)                          for k in self._AllowedKeys}
(12)                    subplotpars = SubplotParams(**kw)
(8)                 else:
(12)                    subplotpars = copy.copy(figure.subplotpars)
(8)                 subplotpars.update(**{k: getattr(self, k) for k in self._AllowedKeys})
(8)                 return subplotpars
(4)             def locally_modified_subplot_params(self):
(8)                 """
(8)                 Return a list of the names of the subplot parameters explicitly set
(8)                 in the GridSpec.
(8)                 This is a subset of the attributes of `.SubplotParams`.
(8)                 """
(8)                 return [k for k in self._AllowedKeys if getattr(self, k)]
(4)             def tight_layout(self, figure, renderer=None,
(21)                             pad=1.08, h_pad=None, w_pad=None, rect=None):
(8)                 """
(8)                 Adjust subplot parameters to give specified padding.
(8)                 Parameters
(8)                 ----------
(8)                 figure : `.Figure`
(12)                    The figure.
(8)                 renderer :  `.RendererBase` subclass, optional
(12)                    The renderer to be used.
(8)                 pad : float
(12)                    Padding between the figure edge and the edges of subplots, as a
(12)                    fraction of the font-size.
(8)                 h_pad, w_pad : float, optional
(12)                    Padding (height/width) between edges of adjacent subplots.
(12)                    Defaults to *pad*.
(8)                 rect : tuple (left, bottom, right, top), default: None
(12)                    (left, bottom, right, top) rectangle in normalized figure
(12)                    coordinates that the whole subplots area (including labels) will
(12)                    fit into. Default (None) is the whole figure.
(8)                 """
```

```
367: (8)                    if renderer is None:
368: (12)                       renderer = figure._get_renderer()
369: (8)                    kwargs = _tight_layout.get_tight_layout_figure(
370: (12)                       figure, figure.axes,
371: (12)                       _tight_layout.get_subplotspec_list(figure.axes, grid_spec=self),
372: (12)                       renderer, pad=pad, h_pad=h_pad, w_pad=w_pad, rect=rect)
373: (8)                    if kwargs:
374: (12)                       self.update(**kwargs)
375: (0)            class GridSpecFromSubplotSpec(GridSpecBase):
376: (4)                """
377: (4)                GridSpec whose subplot layout parameters are inherited from the
378: (4)                location specified by a given SubplotSpec.
379: (4)                """
380: (4)                def __init__(self, nrows, ncols,
381: (17)                            subplot_spec,
382: (17)                            wspace=None, hspace=None,
383: (17)                            height_ratios=None, width_ratios=None):
384: (8)                    """
385: (8)                    Parameters
386: (8)                    ----------
387: (8)                    nrows, ncols : int
388: (12)                       Number of rows and number of columns of the grid.
389: (8)                    subplot_spec : SubplotSpec
390: (12)                       Spec from which the layout parameters are inherited.
391: (8)                    wspace, hspace : float, optional
392: (12)                       See `GridSpec` for more details. If not specified default values
393: (12)                       (from the figure or rcParams) are used.
394: (8)                    height_ratios : array-like of length *nrows*, optional
395: (12)                       See `GridSpecBase` for details.
396: (8)                    width_ratios : array-like of length *ncols*, optional
397: (12)                       See `GridSpecBase` for details.
398: (8)                    """
399: (8)                    self._wspace = wspace
400: (8)                    self._hspace = hspace
401: (8)                    if isinstance(subplot_spec, SubplotSpec):
402: (12)                       self._subplot_spec = subplot_spec
403: (8)                    else:
404: (12)                       raise TypeError(
405: (28)                                       "subplot_spec must be type SubplotSpec, "
406: (28)                                       "usually from GridSpec, or axes.get_subplotspec.")
407: (8)                    self.figure = self._subplot_spec.get_gridspec().figure
408: (8)                    super().__init__(nrows, ncols,
409: (25)                                    width_ratios=width_ratios,
410: (25)                                    height_ratios=height_ratios)
411: (4)                def get_subplot_params(self, figure=None):
412: (8)                    """Return a dictionary of subplot layout parameters."""
413: (8)                    hspace = (self._hspace if self._hspace is not None
414: (18)                             else figure.subplotpars.hspace if figure is not None
415: (18)                             else mpl.rcParams["figure.subplot.hspace"])
416: (8)                    wspace = (self._wspace if self._wspace is not None
417: (18)                             else figure.subplotpars.wspace if figure is not None
418: (18)                             else mpl.rcParams["figure.subplot.wspace"])
419: (8)                    figbox = self._subplot_spec.get_position(figure)
420: (8)                    left, bottom, right, top = figbox.extents
421: (8)                    return SubplotParams(left=left, right=right,
422: (29)                                        bottom=bottom, top=top,
423: (29)                                        wspace=wspace, hspace=hspace)
424: (4)                def get_topmost_subplotspec(self):
425: (8)                    """
426: (8)                    Return the topmost `.SubplotSpec` instance associated with the
subplot.
427: (8)                    """
428: (8)                    return self._subplot_spec.get_topmost_subplotspec()
429: (0)            class SubplotSpec:
430: (4)                """
431: (4)                The location of a subplot in a `GridSpec`.
432: (4)                .. note::
433: (8)                    Likely, you will never instantiate a `SubplotSpec` yourself. Instead,
434: (8)                    you will typically obtain one from a `GridSpec` using item-access.
```

```
    Parameters
    ----------
    gridspec : `~matplotlib.gridspec.GridSpec`
        The GridSpec, which the subplot is referencing.
    num1, num2 : int
        The subplot will occupy the *num1*-th cell of the given
        *gridspec*.  If *num2* is provided, the subplot will span between
        *num1*-th cell and *num2*-th cell **inclusive**.
        The index starts from 0.
    """
    def __init__(self, gridspec, num1, num2=None):
        self._gridspec = gridspec
        self.num1 = num1
        self.num2 = num2
    def __repr__(self):
        return (f"{self.get_gridspec()}["
                f"{self.rowspan.start}:{self.rowspan.stop}, "
                f"{self.colspan.start}:{self.colspan.stop}]")
    @staticmethod
    def _from_subplot_args(figure, args):
        """
        Construct a `.SubplotSpec` from a parent `.Figure` and either
        - a `.SubplotSpec` -- returned as is;
        - one or three numbers -- a MATLAB-style subplot specifier.
        """
        if len(args) == 1:
            arg, = args
            if isinstance(arg, SubplotSpec):
                return arg
            elif not isinstance(arg, Integral):
                raise ValueError(
                    f"Single argument to subplot must be a three-digit "
                    f"integer, not {arg!r}")
            try:
                rows, cols, num = map(int, str(arg))
            except ValueError:
                raise ValueError(
                    f"Single argument to subplot must be a three-digit "
                    f"integer, not {arg!r}") from None
        elif len(args) == 3:
            rows, cols, num = args
        else:
            raise _api.nargs_error("subplot", takes="1 or 3", given=len(args))
        gs = GridSpec._check_gridspec_exists(figure, rows, cols)
        if gs is None:
            gs = GridSpec(rows, cols, figure=figure)
        if isinstance(num, tuple) and len(num) == 2:
            if not all(isinstance(n, Integral) for n in num):
                raise ValueError(
                    f"Subplot specifier tuple must contain integers, not {num}"
                )
            i, j = num
        else:
            if not isinstance(num, Integral) or num < 1 or num > rows*cols:
                raise ValueError(
                    f"num must be an integer with 1 <= num <= {rows*cols}, "
                    f"not {num!r}"
                )
            i = j = num
        return gs[i-1:j]
    @property
    def num2(self):
        return self.num1 if self._num2 is None else self._num2
    @num2.setter
    def num2(self, value):
        self._num2 = value
    def get_gridspec(self):
        return self._gridspec
```

```
    def get_geometry(self):
        """
        Return the subplot geometry as tuple ``(n_rows, n_cols, start, stop)``.

        The indices *start* and *stop* define the range of the subplot within
        the `GridSpec`. *stop* is inclusive (i.e. for a single cell
        ``start == stop``).
        """
        rows, cols = self.get_gridspec().get_geometry()
        return rows, cols, self.num1, self.num2
    @property
    def rowspan(self):
        """The rows spanned by this subplot, as a `range` object."""
        ncols = self.get_gridspec().ncols
        return range(self.num1 // ncols, self.num2 // ncols + 1)
    @property
    def colspan(self):
        """The columns spanned by this subplot, as a `range` object."""
        ncols = self.get_gridspec().ncols
        c1, c2 = sorted([self.num1 % ncols, self.num2 % ncols])
        return range(c1, c2 + 1)
    def is_first_row(self):
        return self.rowspan.start == 0
    def is_last_row(self):
        return self.rowspan.stop == self.get_gridspec().nrows
    def is_first_col(self):
        return self.colspan.start == 0
    def is_last_col(self):
        return self.colspan.stop == self.get_gridspec().ncols
    def get_position(self, figure):
        """
        Update the subplot position from ``figure.subplotpars``.
        """
        gridspec = self.get_gridspec()
        nrows, ncols = gridspec.get_geometry()
        rows, cols = np.unravel_index([self.num1, self.num2], (nrows, ncols))
        fig_bottoms, fig_tops, fig_lefts, fig_rights = \
            gridspec.get_grid_positions(figure)
        fig_bottom = fig_bottoms[rows].min()
        fig_top = fig_tops[rows].max()
        fig_left = fig_lefts[cols].min()
        fig_right = fig_rights[cols].max()
        return Bbox.from_extents(fig_left, fig_bottom, fig_right, fig_top)
    def get_topmost_subplotspec(self):
        """
        Return the topmost `SubplotSpec` instance associated with the subplot.
        """
        gridspec = self.get_gridspec()
        if hasattr(gridspec, "get_topmost_subplotspec"):
            return gridspec.get_topmost_subplotspec()
        else:
            return self
    def __eq__(self, other):
        """
        Two SubplotSpecs are considered equal if they refer to the same
        position(s) in the same `GridSpec`.
        """
        return ((self._gridspec, self.num1, self.num2)
                == (getattr(other, "_gridspec", object()),
                    getattr(other, "num1", object()),
                    getattr(other, "num2", object())))
    def __hash__(self):
        return hash((self._gridspec, self.num1, self.num2))
    def subgridspec(self, nrows, ncols, **kwargs):
        """
        Create a GridSpec within this subplot.
        The created `.GridSpecFromSubplotSpec` will have this `SubplotSpec` as
        a parent.
        Parameters
```

```
(8)                     ----------
(8)                     nrows : int
(12)                        Number of rows in grid.
(8)                     ncols : int
(12)                        Number of columns in grid.
(8)                     Returns
(8)                     -------
(8)                     `.GridSpecFromSubplotSpec`
(8)                     Other Parameters
(8)                     ----------------
(8)                     **kwargs
(12)                        All other parameters are passed to `.GridSpecFromSubplotSpec`.
(8)                     See Also
(8)                     --------
(8)                     matplotlib.pyplot.subplots
(8)                     Examples
(8)                     --------
(8)                     Adding three subplots in the space occupied by a single subplot::
(12)                        fig = plt.figure()
(12)                        gs0 = fig.add_gridspec(3, 1)
(12)                        ax1 = fig.add_subplot(gs0[0])
(12)                        ax2 = fig.add_subplot(gs0[1])
(12)                        gssub = gs0[2].subgridspec(1, 3)
(12)                        for i in range(3):
(16)                            fig.add_subplot(gssub[0, i])
(8)                     """
(8)                     return GridSpecFromSubplotSpec(nrows, ncols, self, **kwargs)
(0)             class SubplotParams:
(4)                 """
(4)                 Parameters defining the positioning of a subplots grid in a figure.
(4)                 """
(4)                 def __init__(self, left=None, bottom=None, right=None, top=None,
(17)                             wspace=None, hspace=None):
(8)                     """
(8)                     Defaults are given by :rc:`figure.subplot.[name]`.
(8)                     Parameters
(8)                     ----------
(8)                     left : float
(12)                        The position of the left edge of the subplots,
(12)                        as a fraction of the figure width.
(8)                     right : float
(12)                        The position of the right edge of the subplots,
(12)                        as a fraction of the figure width.
(8)                     bottom : float
(12)                        The position of the bottom edge of the subplots,
(12)                        as a fraction of the figure height.
(8)                     top : float
(12)                        The position of the top edge of the subplots,
(12)                        as a fraction of the figure height.
(8)                     wspace : float
(12)                        The width of the padding between subplots,
(12)                        as a fraction of the average Axes width.
(8)                     hspace : float
(12)                        The height of the padding between subplots,
(12)                        as a fraction of the average Axes height.
(8)                     """
(8)                     for key in ["left", "bottom", "right", "top", "wspace", "hspace"]:
(12)                        setattr(self, key, mpl.rcParams[f"figure.subplot.{key}"])
(8)                     self.update(left, bottom, right, top, wspace, hspace)
(4)                 def update(self, left=None, bottom=None, right=None, top=None,
(15)                           wspace=None, hspace=None):
(8)                     """
(8)                     Update the dimensions of the passed parameters. *None* means
unchanged.
(8)                     """
(8)                     if ((left if left is not None else self.left)
(16)                            >= (right if right is not None else self.right)):
(12)                        raise ValueError('left cannot be >= right')
(8)                     if ((bottom if bottom is not None else self.bottom)
```

```
639: (16)                           >= (top if top is not None else self.top)):
640: (12)                       raise ValueError('bottom cannot be >= top')
641: (8)                    if left is not None:
642: (12)                       self.left = left
643: (8)                    if right is not None:
644: (12)                       self.right = right
645: (8)                    if bottom is not None:
646: (12)                       self.bottom = bottom
647: (8)                    if top is not None:
648: (12)                       self.top = top
649: (8)                    if wspace is not None:
650: (12)                       self.wspace = wspace
651: (8)                    if hspace is not None:
652: (12)                       self.hspace = hspace
```

File 41 - hatch.py:

```
1: (0)              """Contains classes for generating hatch patterns."""
2: (0)              import numpy as np
3: (0)              from matplotlib import _api
4: (0)              from matplotlib.path import Path
5: (0)              class HatchPatternBase:
6: (4)                  """The base class for a hatch pattern."""
7: (4)                  pass
8: (0)              class HorizontalHatch(HatchPatternBase):
9: (4)                  def __init__(self, hatch, density):
10: (8)                     self.num_lines = int((hatch.count('-') + hatch.count('+')) * density)
11: (8)                     self.num_vertices = self.num_lines * 2
12: (4)                 def set_vertices_and_codes(self, vertices, codes):
13: (8)                     steps, stepsize = np.linspace(0.0, 1.0, self.num_lines, False,
14: (38)                                                   retstep=True)
15: (8)                     steps += stepsize / 2.
16: (8)                     vertices[0::2, 0] = 0.0
17: (8)                     vertices[0::2, 1] = steps
18: (8)                     vertices[1::2, 0] = 1.0
19: (8)                     vertices[1::2, 1] = steps
20: (8)                     codes[0::2] = Path.MOVETO
21: (8)                     codes[1::2] = Path.LINETO
22: (0)             class VerticalHatch(HatchPatternBase):
23: (4)                 def __init__(self, hatch, density):
24: (8)                     self.num_lines = int((hatch.count('|') + hatch.count('+')) * density)
25: (8)                     self.num_vertices = self.num_lines * 2
26: (4)                 def set_vertices_and_codes(self, vertices, codes):
27: (8)                     steps, stepsize = np.linspace(0.0, 1.0, self.num_lines, False,
28: (38)                                                   retstep=True)
29: (8)                     steps += stepsize / 2.
30: (8)                     vertices[0::2, 0] = steps
31: (8)                     vertices[0::2, 1] = 0.0
32: (8)                     vertices[1::2, 0] = steps
33: (8)                     vertices[1::2, 1] = 1.0
34: (8)                     codes[0::2] = Path.MOVETO
35: (8)                     codes[1::2] = Path.LINETO
36: (0)             class NorthEastHatch(HatchPatternBase):
37: (4)                 def __init__(self, hatch, density):
38: (8)                     self.num_lines = int(
39: (12)                        (hatch.count('/') + hatch.count('x') + hatch.count('X')) *
density)
40: (8)                     if self.num_lines:
41: (12)                        self.num_vertices = (self.num_lines + 1) * 2
42: (8)                     else:
43: (12)                        self.num_vertices = 0
44: (4)                 def set_vertices_and_codes(self, vertices, codes):
45: (8)                     steps = np.linspace(-0.5, 0.5, self.num_lines + 1)
46: (8)                     vertices[0::2, 0] = 0.0 + steps
47: (8)                     vertices[0::2, 1] = 0.0 - steps
48: (8)                     vertices[1::2, 0] = 1.0 + steps
49: (8)                     vertices[1::2, 1] = 1.0 - steps
```

```
 50: (8)                    codes[0::2] = Path.MOVETO
 51: (8)                    codes[1::2] = Path.LINETO
 52: (0)             class SouthEastHatch(HatchPatternBase):
 53: (4)                 def __init__(self, hatch, density):
 54: (8)                     self.num_lines = int(
 55: (12)                        (hatch.count('\\') + hatch.count('x') + hatch.count('X'))
 56: (12)                        * density)
 57: (8)                     if self.num_lines:
 58: (12)                        self.num_vertices = (self.num_lines + 1) * 2
 59: (8)                     else:
 60: (12)                        self.num_vertices = 0
 61: (4)                 def set_vertices_and_codes(self, vertices, codes):
 62: (8)                     steps = np.linspace(-0.5, 0.5, self.num_lines + 1)
 63: (8)                     vertices[0::2, 0] = 0.0 + steps
 64: (8)                     vertices[0::2, 1] = 1.0 + steps
 65: (8)                     vertices[1::2, 0] = 1.0 + steps
 66: (8)                     vertices[1::2, 1] = 0.0 + steps
 67: (8)                     codes[0::2] = Path.MOVETO
 68: (8)                     codes[1::2] = Path.LINETO
 69: (0)             class Shapes(HatchPatternBase):
 70: (4)                 filled = False
 71: (4)                 def __init__(self, hatch, density):
 72: (8)                     if self.num_rows == 0:
 73: (12)                        self.num_shapes = 0
 74: (12)                        self.num_vertices = 0
 75: (8)                     else:
 76: (12)                        self.num_shapes = ((self.num_rows // 2 + 1) * (self.num_rows + 1)
+
 77: (31)                                           (self.num_rows // 2) * self.num_rows)
 78: (12)                        self.num_vertices = (self.num_shapes *
 79: (33)                                             len(self.shape_vertices) *
 80: (33)                                             (1 if self.filled else 2))
 81: (4)                 def set_vertices_and_codes(self, vertices, codes):
 82: (8)                     offset = 1.0 / self.num_rows
 83: (8)                     shape_vertices = self.shape_vertices * offset * self.size
 84: (8)                     shape_codes = self.shape_codes
 85: (8)                     if not self.filled:
 86: (12)                        shape_vertices = np.concatenate(  # Forward, then backward.
 87: (16)                            [shape_vertices, shape_vertices[::-1] * 0.9])
 88: (12)                        shape_codes = np.concatenate([shape_codes, shape_codes])
 89: (8)                     vertices_parts = []
 90: (8)                     codes_parts = []
 91: (8)                     for row in range(self.num_rows + 1):
 92: (12)                        if row % 2 == 0:
 93: (16)                            cols = np.linspace(0, 1, self.num_rows + 1)
 94: (12)                        else:
 95: (16)                            cols = np.linspace(offset / 2, 1 - offset / 2, self.num_rows)
 96: (12)                        row_pos = row * offset
 97: (12)                        for col_pos in cols:
 98: (16)                            vertices_parts.append(shape_vertices + [col_pos, row_pos])
 99: (16)                            codes_parts.append(shape_codes)
 100: (8)                    np.concatenate(vertices_parts, out=vertices)
 101: (8)                    np.concatenate(codes_parts, out=codes)
 102: (0)            class Circles(Shapes):
 103: (4)                def __init__(self, hatch, density):
 104: (8)                    path = Path.unit_circle()
 105: (8)                    self.shape_vertices = path.vertices
 106: (8)                    self.shape_codes = path.codes
 107: (8)                    super().__init__(hatch, density)
 108: (0)            class SmallCircles(Circles):
 109: (4)                size = 0.2
 110: (4)                def __init__(self, hatch, density):
 111: (8)                    self.num_rows = (hatch.count('o')) * density
 112: (8)                    super().__init__(hatch, density)
 113: (0)            class LargeCircles(Circles):
 114: (4)                size = 0.35
 115: (4)                def __init__(self, hatch, density):
 116: (8)                    self.num_rows = (hatch.count('O')) * density
 117: (8)                    super().__init__(hatch, density)
```

```
118: (0)            class SmallFilledCircles(Circles):
119: (4)                size = 0.1
120: (4)                filled = True
121: (4)                def __init__(self, hatch, density):
122: (8)                    self.num_rows = (hatch.count('.')) * density
123: (8)                    super().__init__(hatch, density)
124: (0)            class Stars(Shapes):
125: (4)                size = 1.0 / 3.0
126: (4)                filled = True
127: (4)                def __init__(self, hatch, density):
128: (8)                    self.num_rows = (hatch.count('*')) * density
129: (8)                    path = Path.unit_regular_star(5)
130: (8)                    self.shape_vertices = path.vertices
131: (8)                    self.shape_codes = np.full(len(self.shape_vertices), Path.LINETO,
132: (35)                                               dtype=Path.code_type)
133: (8)                    self.shape_codes[0] = Path.MOVETO
134: (8)                    super().__init__(hatch, density)
135: (0)            _hatch_types = [
136: (4)                HorizontalHatch,
137: (4)                VerticalHatch,
138: (4)                NorthEastHatch,
139: (4)                SouthEastHatch,
140: (4)                SmallCircles,
141: (4)                LargeCircles,
142: (4)                SmallFilledCircles,
143: (4)                Stars
144: (4)                ]
145: (0)            def _validate_hatch_pattern(hatch):
146: (4)                valid_hatch_patterns = set(r'-+|/\xXoO.*')
147: (4)                if hatch is not None:
148: (8)                    invalids = set(hatch).difference(valid_hatch_patterns)
149: (8)                    if invalids:
150: (12)                       valid = ''.join(sorted(valid_hatch_patterns))
151: (12)                       invalids = ''.join(sorted(invalids))
152: (12)                       _api.warn_deprecated(
153: (16)                           '3.4',
154: (16)                           removal='3.11',  # one release after custom hatches (#20690)
155: (16)                           message=f'hatch must consist of a string of "{valid}" or '
156: (24)                                   'None, but found the following invalid values '
157: (24)                                   f'"{invalids}". Passing invalid values is deprecated '
158: (24)                                   'since %(since)s and will become an error %
(removal)s.'
159: (12)                       )
160: (0)            def get_path(hatchpattern, density=6):
161: (4)                """
162: (4)                Given a hatch specifier, *hatchpattern*, generates Path to render
163: (4)                the hatch in a unit square.  *density* is the number of lines per
164: (4)                unit square.
165: (4)                """
166: (4)                density = int(density)
167: (4)                patterns = [hatch_type(hatchpattern, density)
168: (16)                           for hatch_type in _hatch_types]
169: (4)                num_vertices = sum([pattern.num_vertices for pattern in patterns])
170: (4)                if num_vertices == 0:
171: (8)                    return Path(np.empty((0, 2)))
172: (4)                vertices = np.empty((num_vertices, 2))
173: (4)                codes = np.empty(num_vertices, Path.code_type)
174: (4)                cursor = 0
175: (4)                for pattern in patterns:
176: (8)                    if pattern.num_vertices != 0:
177: (12)                       vertices_chunk = vertices[cursor:cursor + pattern.num_vertices]
178: (12)                       codes_chunk = codes[cursor:cursor + pattern.num_vertices]
179: (12)                       pattern.set_vertices_and_codes(vertices_chunk, codes_chunk)
180: (12)                       cursor += pattern.num_vertices
181: (4)                return Path(vertices, codes)
```

File 42 - image.py:

```
1: (0)              """
2: (0)              The image module supports basic image loading, rescaling and display
3: (0)              operations.
4: (0)              """
5: (0)              import math
6: (0)              import os
7: (0)              import logging
8: (0)              from pathlib import Path
9: (0)              import warnings
10: (0)             import numpy as np
11: (0)             import PIL.Image
12: (0)             import PIL.PngImagePlugin
13: (0)             import matplotlib as mpl
14: (0)             from matplotlib import _api, cbook, cm
15: (0)             from matplotlib import _image
16: (0)             from matplotlib._image import *  # noqa: F401, F403
17: (0)             import matplotlib.artist as martist
18: (0)             from matplotlib.backend_bases import FigureCanvasBase
19: (0)             import matplotlib.colors as mcolors
20: (0)             from matplotlib.transforms import (
21: (4)                 Affine2D, BboxBase, Bbox, BboxTransform, BboxTransformTo,
22: (4)                 IdentityTransform, TransformedBbox)
23: (0)             _log = logging.getLogger(__name__)
24: (0)             _interpd_ = {
25: (4)                 'antialiased': _image.NEAREST,  # this will use nearest or Hanning...
26: (4)                 'none': _image.NEAREST,  # fall back to nearest when not supported
27: (4)                 'nearest': _image.NEAREST,
28: (4)                 'bilinear': _image.BILINEAR,
29: (4)                 'bicubic': _image.BICUBIC,
30: (4)                 'spline16': _image.SPLINE16,
31: (4)                 'spline36': _image.SPLINE36,
32: (4)                 'hanning': _image.HANNING,
33: (4)                 'hamming': _image.HAMMING,
34: (4)                 'hermite': _image.HERMITE,
35: (4)                 'kaiser': _image.KAISER,
36: (4)                 'quadric': _image.QUADRIC,
37: (4)                 'catrom': _image.CATROM,
38: (4)                 'gaussian': _image.GAUSSIAN,
39: (4)                 'bessel': _image.BESSEL,
40: (4)                 'mitchell': _image.MITCHELL,
41: (4)                 'sinc': _image.SINC,
42: (4)                 'lanczos': _image.LANCZOS,
43: (4)                 'blackman': _image.BLACKMAN,
44: (0)             }
45: (0)             interpolations_names = set(_interpd_)
46: (0)             def composite_images(images, renderer, magnification=1.0):
47: (4)                 """
48: (4)                 Composite a number of RGBA images into one.  The images are
49: (4)                 composited in the order in which they appear in the *images* list.
50: (4)                 Parameters
51: (4)                 ----------
52: (4)                 images : list of Images
53: (8)                     Each must have a `make_image` method.  For each image,
54: (8)                     `can_composite` should return `True`, though this is not
55: (8)                     enforced by this function.  Each image must have a purely
56: (8)                     affine transformation with no shear.
57: (4)                 renderer : `.RendererBase`
58: (4)                 magnification : float, default: 1
59: (8)                     The additional magnification to apply for the renderer in use.
60: (4)                 Returns
61: (4)                 -------
62: (4)                 image : (M, N, 4) `numpy.uint8` array
63: (8)                     The composited RGBA image.
64: (4)                 offset_x, offset_y : float
65: (8)                     The (left, bottom) offset where the composited image should be placed
66: (8)                     in the output figure.
67: (4)                 """
68: (4)                 if len(images) == 0:
```

```
 69: (8)                    return np.empty((0, 0, 4), dtype=np.uint8), 0, 0
 70: (4)                parts = []
 71: (4)                bboxes = []
 72: (4)                for image in images:
 73: (8)                    data, x, y, trans = image.make_image(renderer, magnification)
 74: (8)                    if data is not None:
 75: (12)                       x *= magnification
 76: (12)                       y *= magnification
 77: (12)                       parts.append((data, x, y, image._get_scalar_alpha()))
 78: (12)                       bboxes.append(
 79: (16)                           Bbox([[x, y], [x + data.shape[1], y + data.shape[0]]]))
 80: (4)                if len(parts) == 0:
 81: (8)                    return np.empty((0, 0, 4), dtype=np.uint8), 0, 0
 82: (4)                bbox = Bbox.union(bboxes)
 83: (4)                output = np.zeros(
 84: (8)                    (int(bbox.height), int(bbox.width), 4), dtype=np.uint8)
 85: (4)                for data, x, y, alpha in parts:
 86: (8)                    trans = Affine2D().translate(x - bbox.x0, y - bbox.y0)
 87: (8)                    _image.resample(data, output, trans, _image.NEAREST,
 88: (24)                                   resample=False, alpha=alpha)
 89: (4)                return output, bbox.x0 / magnification, bbox.y0 / magnification
 90: (0)            def _draw_list_compositing_images(
 91: (8)                    renderer, parent, artists, suppress_composite=None):
 92: (4)                """
 93: (4)                Draw a sorted list of artists, compositing images into a single
 94: (4)                image where possible.
 95: (4)                For internal Matplotlib use only: It is here to reduce duplication
 96: (4)                between `Figure.draw` and `Axes.draw`, but otherwise should not be
 97: (4)                generally useful.
 98: (4)                """
 99: (4)                has_images = any(isinstance(x, _ImageBase) for x in artists)
100: (4)                not_composite = (suppress_composite if suppress_composite is not None
101: (21)                                else renderer.option_image_nocomposite())
102: (4)                if not_composite or not has_images:
103: (8)                    for a in artists:
104: (12)                       a.draw(renderer)
105: (4)                else:
106: (8)                    image_group = []
107: (8)                    mag = renderer.get_image_magnification()
108: (8)                    def flush_images():
109: (12)                       if len(image_group) == 1:
110: (16)                           image_group[0].draw(renderer)
111: (12)                       elif len(image_group) > 1:
112: (16)                           data, l, b = composite_images(image_group, renderer, mag)
113: (16)                           if data.size != 0:
114: (20)                               gc = renderer.new_gc()
115: (20)                               gc.set_clip_rectangle(parent.bbox)
116: (20)                               gc.set_clip_path(parent.get_clip_path())
117: (20)                               renderer.draw_image(gc, round(l), round(b), data)
118: (20)                               gc.restore()
119: (12)                       del image_group[:]
120: (8)                    for a in artists:
121: (12)                       if (isinstance(a, _ImageBase) and a.can_composite() and
122: (20)                               a.get_clip_on() and not a.get_clip_path()):
123: (16)                           image_group.append(a)
124: (12)                       else:
125: (16)                           flush_images()
126: (16)                           a.draw(renderer)
127: (8)                    flush_images()
128: (0)            def _resample(
129: (8)                    image_obj, data, out_shape, transform, *, resample=None, alpha=1):
130: (4)                """
131: (4)                Convenience wrapper around `._image.resample` to resample *data* to
132: (4)                *out_shape* (with a third dimension if *data* is RGBA) that takes care of
133: (4)                allocating the output array and fetching the relevant properties from the
134: (4)                Image object *image_obj*.
135: (4)                """
136: (4)                msg = ('Data with more than {n} cannot be accurately displayed. '
137: (11)                      'Downsampling to less than {n} before displaying. '
```

```
(11)                        'To remove this warning, manually downsample your data.')
(4)                 if data.shape[1] > 2**23:
(8)                     warnings.warn(msg.format(n='2**23 columns'))
(8)                     step = int(np.ceil(data.shape[1] / 2**23))
(8)                     data = data[:, ::step]
(8)                     transform = Affine2D().scale(step, 1) + transform
(4)                 if data.shape[0] > 2**24:
(8)                     warnings.warn(msg.format(n='2**24 rows'))
(8)                     step = int(np.ceil(data.shape[0] / 2**24))
(8)                     data = data[::step, :]
(8)                     transform = Affine2D().scale(1, step) + transform
(4)                 interpolation = image_obj.get_interpolation()
(4)                 if interpolation == 'antialiased':
(8)                     pos = np.array([[0, 0], [data.shape[1], data.shape[0]]])
(8)                     disp = transform.transform(pos)
(8)                     dispx = np.abs(np.diff(disp[:, 0]))
(8)                     dispy = np.abs(np.diff(disp[:, 1]))
(8)                     if ((dispx > 3 * data.shape[1] or
(16)                            dispx == data.shape[1] or
(16)                            dispx == 2 * data.shape[1]) and
(12)                        (dispy > 3 * data.shape[0] or
(16)                            dispy == data.shape[0] or
(16)                            dispy == 2 * data.shape[0])):
(12)                        interpolation = 'nearest'
(8)                     else:
(12)                        interpolation = 'hanning'
(4)                 out = np.zeros(out_shape + data.shape[2:], data.dtype)  # 2D->2D, 3D->3D.
(4)                 if resample is None:
(8)                     resample = image_obj.get_resample()
(4)                 _image.resample(data, out, transform,
(20)                                _interpd_[interpolation],
(20)                                resample,
(20)                                alpha,
(20)                                image_obj.get_filternorm(),
(20)                                image_obj.get_filterrad())
(4)                 return out
(0)             def _rgb_to_rgba(A):
(4)                 """
(4)                 Convert an RGB image to RGBA, as required by the image resample C++
(4)                 extension.
(4)                 """
(4)                 rgba = np.zeros((A.shape[0], A.shape[1], 4), dtype=A.dtype)
(4)                 rgba[:, :, :3] = A
(4)                 if rgba.dtype == np.uint8:
(8)                     rgba[:, :, 3] = 255
(4)                 else:
(8)                     rgba[:, :, 3] = 1.0
(4)                 return rgba
(0)             class _ImageBase(martist.Artist, cm.ScalarMappable):
(4)                 """
(4)                 Base class for images.
(4)                 interpolation and cmap default to their rc settings
(4)                 cmap is a colors.Colormap instance
(4)                 norm is a colors.Normalize instance to map luminance to 0-1
(4)                 extent is data axes (left, right, bottom, top) for making image plots
(4)                 registered with data plots.  Default is to label the pixel
(4)                 centers with the zero-based row and column indices.
(4)                 Additional kwargs are matplotlib.artist properties
(4)                 """
(4)                 zorder = 0
(4)                 def __init__(self, ax,
(17)                             cmap=None,
(17)                             norm=None,
(17)                             interpolation=None,
(17)                             origin=None,
(17)                             filternorm=True,
(17)                             filterrad=4.0,
(17)                             resample=False,
(17)                             *,
```

```
207: (17)                             interpolation_stage=None,
208: (17)                             **kwargs
209: (17)                             ):
210: (8)                     martist.Artist.__init__(self)
211: (8)                     cm.ScalarMappable.__init__(self, norm, cmap)
212: (8)                     if origin is None:
213: (12)                        origin = mpl.rcParams['image.origin']
214: (8)                     _api.check_in_list(["upper", "lower"], origin=origin)
215: (8)                     self.origin = origin
216: (8)                     self.set_filternorm(filternorm)
217: (8)                     self.set_filterrad(filterrad)
218: (8)                     self.set_interpolation(interpolation)
219: (8)                     self.set_interpolation_stage(interpolation_stage)
220: (8)                     self.set_resample(resample)
221: (8)                     self.axes = ax
222: (8)                     self._imcache = None
223: (8)                     self._internal_update(kwargs)
224: (4)                 def __str__(self):
225: (8)                     try:
226: (12)                        shape = self.get_shape()
227: (12)                        return f"{type(self).__name__}(shape={shape!r})"
228: (8)                     except RuntimeError:
229: (12)                        return type(self).__name__
230: (4)                 def __getstate__(self):
231: (8)                     return {**super().__getstate__(), "_imcache": None}
232: (4)                 def get_size(self):
233: (8)                     """Return the size of the image as tuple (numrows, numcols)."""
234: (8)                     return self.get_shape()[:2]
235: (4)                 def get_shape(self):
236: (8)                     """
237: (8)                     Return the shape of the image as tuple (numrows, numcols, channels).
238: (8)                     """
239: (8)                     if self._A is None:
240: (12)                        raise RuntimeError('You must first set the image array')
241: (8)                     return self._A.shape
242: (4)                 def set_alpha(self, alpha):
243: (8)                     """
244: (8)                     Set the alpha value used for blending - not supported on all backends.
245: (8)                     Parameters
246: (8)                     ----------
247: (8)                     alpha : float or 2D array-like or None
248: (8)                     """
249: (8)                     martist.Artist._set_alpha_for_array(self, alpha)
250: (8)                     if np.ndim(alpha) not in (0, 2):
251: (12)                        raise TypeError('alpha must be a float, two-dimensional '
252: (28)                                        'array, or None')
253: (8)                     self._imcache = None
254: (4)                 def _get_scalar_alpha(self):
255: (8)                     """
256: (8)                     Get a scalar alpha value to be applied to the artist as a whole.
257: (8)                     If the alpha value is a matrix, the method returns 1.0 because pixels
258: (8)                     have individual alpha values (see `~._ImageBase._make_image` for
259: (8)                     details). If the alpha value is a scalar, the method returns said
value
260: (8)                     to be applied to the artist as a whole because pixels do not have
261: (8)                     individual alpha values.
262: (8)                     """
263: (8)                     return 1.0 if self._alpha is None or np.ndim(self._alpha) > 0 \
264: (12)                        else self._alpha
265: (4)                 def changed(self):
266: (8)                     """
267: (8)                     Call this whenever the mappable is changed so observers can update.
268: (8)                     """
269: (8)                     self._imcache = None
270: (8)                     cm.ScalarMappable.changed(self)
271: (4)                 def _make_image(self, A, in_bbox, out_bbox, clip_bbox, magnification=1.0,
272: (20)                                unsampled=False, round_to_pixel_border=True):
273: (8)                     """
274: (8)                     Normalize, rescale, and colormap the image *A* from the given
```

```
*in_bbox*
                    (in data space), to the given *out_bbox* (in pixel space) clipped to
                    the given *clip_bbox* (also in pixel space), and magnified by the
                    *magnification* factor.
                    *A* may be a greyscale image (M, N) with a dtype of `~numpy.float32`,
                    `~numpy.float64`, `~numpy.float128`, `~numpy.uint16` or
`~numpy.uint8`,
                    or an (M, N, 4) RGBA image with a dtype of `~numpy.float32`,
                    `~numpy.float64`, `~numpy.float128`, or `~numpy.uint8`.
                    If *unsampled* is True, the image will not be scaled, but an
                    appropriate affine transformation will be returned instead.
                    If *round_to_pixel_border* is True, the output image size will be
                    rounded to the nearest pixel boundary.  This makes the images align
                    correctly with the Axes.  It should not be used if exact scaling is
                    needed, such as for `FigureImage`.
                    Returns
                    -------
                    image : (M, N, 4) `numpy.uint8` array
                        The RGBA image, resampled unless *unsampled* is True.
                    x, y : float
                        The upper left corner where the image should be drawn, in pixel
                        space.
                    trans : `~matplotlib.transforms.Affine2D`
                        The affine transformation from image to pixel space.
                    """
                    if A is None:
                        raise RuntimeError('You must first set the image '
                                           'array or the image attribute')
                    if A.size == 0:
                        raise RuntimeError("_make_image must get a non-empty image. "
                                           "Your Artist's draw method must filter before "
                                           "this method is called.")
                    clipped_bbox = Bbox.intersection(out_bbox, clip_bbox)
                    if clipped_bbox is None:
                        return None, 0, 0, None
                    out_width_base = clipped_bbox.width * magnification
                    out_height_base = clipped_bbox.height * magnification
                    if out_width_base == 0 or out_height_base == 0:
                        return None, 0, 0, None
                    if self.origin == 'upper':
                        t0 = Affine2D().translate(0, -A.shape[0]).scale(1, -1)
                    else:
                        t0 = IdentityTransform()
                    t0 += (
                        Affine2D()
                        .scale(
                            in_bbox.width / A.shape[1],
                            in_bbox.height / A.shape[0])
                        .translate(in_bbox.x0, in_bbox.y0)
                        + self.get_transform())
                    t = (t0
                         + (Affine2D()
                            .translate(-clipped_bbox.x0, -clipped_bbox.y0)
                            .scale(magnification)))
                    if ((not unsampled) and t.is_affine and round_to_pixel_border and
                            (out_width_base % 1.0 != 0.0 or out_height_base % 1.0 !=
0.0)):
                        out_width = math.ceil(out_width_base)
                        out_height = math.ceil(out_height_base)
                        extra_width = (out_width - out_width_base) / out_width_base
                        extra_height = (out_height - out_height_base) / out_height_base
                        t += Affine2D().scale(1.0 + extra_width, 1.0 + extra_height)
                    else:
                        out_width = int(out_width_base)
                        out_height = int(out_height_base)
                    out_shape = (out_height, out_width)
                    if not unsampled:
                        if not (A.ndim == 2 or A.ndim == 3 and A.shape[-1] in (3, 4)):
                            raise ValueError(f"Invalid shape {A.shape} for image data")
```

```
(12)                        if A.ndim == 2 and self._interpolation_stage != 'rgba':
(16)                            a_min = A.min()
(16)                            a_max = A.max()
(16)                            if a_min is np.ma.masked:  # All masked; values don't matter.
(20)                                a_min, a_max = np.int32(0), np.int32(1)
(16)                            if A.dtype.kind == 'f':  # Float dtype: scale to same dtype.
(20)                                scaled_dtype = np.dtype(
(24)                                    np.float64 if A.dtype.itemsize > 4 else np.float32)
(20)                                if scaled_dtype.itemsize < A.dtype.itemsize:
(24)                                    _api.warn_external(f"Casting input data from
{A.dtype}"
(43)                                                       f" to {scaled_dtype} for imshow.")
(16)                            else:  # Int dtype, likely.
(20)                                da = a_max.astype(np.float64) - a_min.astype(np.float64)
(20)                                scaled_dtype = np.float64 if da > 1e8 else np.float32
(16)                            A_scaled = np.array(A, dtype=scaled_dtype)
(16)                            self.norm.autoscale_None(A)
(16)                            dv = np.float64(self.norm.vmax) - np.float64(self.norm.vmin)
(16)                            vmid = np.float64(self.norm.vmin) + dv / 2
(16)                            fact = 1e7 if scaled_dtype == np.float64 else 1e4
(16)                            newmin = vmid - dv * fact
(16)                            if newmin < a_min:
(20)                                newmin = None
(16)                            else:
(20)                                a_min = np.float64(newmin)
(16)                            newmax = vmid + dv * fact
(16)                            if newmax > a_max:
(20)                                newmax = None
(16)                            else:
(20)                                a_max = np.float64(newmax)
(16)                            if newmax is not None or newmin is not None:
(20)                                np.clip(A_scaled, newmin, newmax, out=A_scaled)
(16)                            offset = .1
(16)                            frac = .8
(16)                            vmin, vmax = self.norm.vmin, self.norm.vmax
(16)                            if vmin is np.ma.masked:
(20)                                vmin, vmax = a_min, a_max
(16)                            vrange = np.array([vmin, vmax], dtype=scaled_dtype)
(16)                            A_scaled -= a_min
(16)                            vrange -= a_min
(16)                            a_min = a_min.astype(scaled_dtype).item()
(16)                            a_max = a_max.astype(scaled_dtype).item()
(16)                            if a_min != a_max:
(20)                                A_scaled /= ((a_max - a_min) / frac)
(20)                                vrange /= ((a_max - a_min) / frac)
(16)                            A_scaled += offset
(16)                            vrange += offset
(16)                            A_resampled = _resample(self, A_scaled, out_shape, t)
(16)                            del A_scaled  # Make sure we don't use A_scaled anymore!
(16)                            A_resampled -= offset
(16)                            vrange -= offset
(16)                            if a_min != a_max:
(20)                                A_resampled *= ((a_max - a_min) / frac)
(20)                                vrange *= ((a_max - a_min) / frac)
(16)                            A_resampled += a_min
(16)                            vrange += a_min
(16)                            if isinstance(self.norm, mcolors.NoNorm):
(20)                                A_resampled = A_resampled.astype(A.dtype)
(16)                            mask = (np.where(A.mask, np.float32(np.nan), np.float32(1))
(24)                                    if A.mask.shape == A.shape  # nontrivial mask
(24)                                    else np.ones_like(A, np.float32))
(16)                            out_alpha = _resample(self, mask, out_shape, t, resample=True)
(16)                            del mask  # Make sure we don't use mask anymore!
(16)                            out_mask = np.isnan(out_alpha)
(16)                            out_alpha[out_mask] = 1
(16)                            alpha = self.get_alpha()
(16)                            if alpha is not None and np.ndim(alpha) > 0:
(20)                                out_alpha *= _resample(self, alpha, out_shape,
(43)                                                       t, resample=True)
```

```
409: (16)                            resampled_masked = np.ma.masked_array(A_resampled, out_mask)
410: (16)                            s_vmin, s_vmax = vrange
411: (16)                            if isinstance(self.norm, mcolors.LogNorm) and s_vmin <= 0:
412: (20)                                s_vmin = np.finfo(scaled_dtype).eps
413: (16)                            with self.norm.callbacks.blocked(), \
414: (21)                                 cbook._setattr_cm(self.norm, vmin=s_vmin, vmax=s_vmax):
415: (20)                                output = self.norm(resampled_masked)
416: (12)                        else:
417: (16)                            if A.ndim == 2:  # _interpolation_stage == 'rgba'
418: (20)                                self.norm.autoscale_None(A)
419: (20)                                A = self.to_rgba(A)
420: (16)                            alpha = self._get_scalar_alpha()
421: (16)                            if A.shape[2] == 3:
422: (20)                                output_alpha = (255 * alpha) if A.dtype == np.uint8 else
alpha
423: (16)                            else:
424: (20)                                output_alpha = _resample(  # resample alpha channel
425: (24)                                    self, A[..., 3], out_shape, t, alpha=alpha)
426: (16)                            output = _resample(  # resample rgb channels
427: (20)                                self, _rgb_to_rgba(A[..., :3]), out_shape, t, alpha=alpha)
428: (16)                            output[..., 3] = output_alpha  # recombine rgb and alpha
429: (12)                        output = self.to_rgba(output, bytes=True, norm=False)
430: (12)                        if A.ndim == 2:
431: (16)                            alpha = self._get_scalar_alpha()
432: (16)                            alpha_channel = output[:, :, 3]
433: (16)                            alpha_channel[:] = (  # Assignment will cast to uint8.
434: (20)                                alpha_channel.astype(np.float32) * out_alpha * alpha)
435: (8)                    else:
436: (12)                        if self._imcache is None:
437: (16)                            self._imcache = self.to_rgba(A, bytes=True, norm=(A.ndim ==
2))
438: (12)                        output = self._imcache
439: (12)                        subset = TransformedBbox(clip_bbox, t0.inverted()).frozen()
440: (12)                        output = output[
441: (16)                            int(max(subset.ymin, 0)):
442: (16)                            int(min(subset.ymax + 1, output.shape[0])),
443: (16)                            int(max(subset.xmin, 0)):
444: (16)                            int(min(subset.xmax + 1, output.shape[1]))]
445: (12)                        t = Affine2D().translate(
446: (16)                            int(max(subset.xmin, 0)), int(max(subset.ymin, 0))) + t
447: (8)                    return output, clipped_bbox.x0, clipped_bbox.y0, t
448: (4)                def make_image(self, renderer, magnification=1.0, unsampled=False):
449: (8)                    """
450: (8)                    Normalize, rescale, and colormap this image's data for rendering using
451: (8)                    *renderer*, with the given *magnification*.
452: (8)                    If *unsampled* is True, the image will not be scaled, but an
453: (8)                    appropriate affine transformation will be returned instead.
454: (8)                    Returns
455: (8)                    -------
456: (8)                    image : (M, N, 4) `numpy.uint8` array
457: (12)                       The RGBA image, resampled unless *unsampled* is True.
458: (8)                    x, y : float
459: (12)                       The upper left corner where the image should be drawn, in pixel
460: (12)                       space.
461: (8)                    trans : `~matplotlib.transforms.Affine2D`
462: (12)                       The affine transformation from image to pixel space.
463: (8)                    """
464: (8)                    raise NotImplementedError('The make_image method must be overridden')
465: (4)                def _check_unsampled_image(self):
466: (8)                    """
467: (8)                    Return whether the image is better to be drawn unsampled.
468: (8)                    The derived class needs to override it.
469: (8)                    """
470: (8)                    return False
471: (4)                @martist.allow_rasterization
472: (4)                def draw(self, renderer):
473: (8)                    if not self.get_visible():
474: (12)                       self.stale = False
475: (12)                       return
```

```
476: (8)                    if self.get_array().size == 0:
477: (12)                       self.stale = False
478: (12)                       return
479: (8)                    gc = renderer.new_gc()
480: (8)                    self._set_gc_clip(gc)
481: (8)                    gc.set_alpha(self._get_scalar_alpha())
482: (8)                    gc.set_url(self.get_url())
483: (8)                    gc.set_gid(self.get_gid())
484: (8)                    if (renderer.option_scale_image()  # Renderer supports transform
kwarg.
485: (16)                           and self._check_unsampled_image()
486: (16)                           and self.get_transform().is_affine):
487: (12)                       im, l, b, trans = self.make_image(renderer, unsampled=True)
488: (12)                       if im is not None:
489: (16)                           trans = Affine2D().scale(im.shape[1], im.shape[0]) + trans
490: (16)                           renderer.draw_image(gc, l, b, im, trans)
491: (8)                    else:
492: (12)                       im, l, b, trans = self.make_image(
493: (16)                           renderer, renderer.get_image_magnification())
494: (12)                       if im is not None:
495: (16)                           renderer.draw_image(gc, l, b, im)
496: (8)                    gc.restore()
497: (8)                    self.stale = False
498: (4)                def contains(self, mouseevent):
499: (8)                    """Test whether the mouse event occurred within the image."""
500: (8)                    if (self._different_canvas(mouseevent)
501: (16)                           or not self.axes.contains(mouseevent)[0]):
502: (12)                       return False, {}
503: (8)                    trans = self.get_transform().inverted()
504: (8)                    x, y = trans.transform([mouseevent.x, mouseevent.y])
505: (8)                    xmin, xmax, ymin, ymax = self.get_extent()
506: (8)                    inside = (x is not None and (x - xmin) * (x - xmax) <= 0
507: (18)                             and y is not None and (y - ymin) * (y - ymax) <= 0)
508: (8)                    return inside, {}
509: (4)                def write_png(self, fname):
510: (8)                    """Write the image to png file *fname*."""
511: (8)                    im = self.to_rgba(self._A[::-1] if self.origin == 'lower' else
self._A,
512: (26)                                      bytes=True, norm=True)
513: (8)                    PIL.Image.fromarray(im).save(fname, format="png")
514: (4)                @staticmethod
515: (4)                def _normalize_image_array(A):
516: (8)                    """
517: (8)                    Check validity of image-like input *A* and normalize it to a format
suitable for
518: (8)                    Image subclasses.
519: (8)                    """
520: (8)                    A = cbook.safe_masked_invalid(A, copy=True)
521: (8)                    if A.dtype != np.uint8 and not np.can_cast(A.dtype, float,
"same_kind"):
522: (12)                       raise TypeError(f"Image data of dtype {A.dtype} cannot be "
523: (28)                                       f"converted to float")
524: (8)                    if A.ndim == 3 and A.shape[-1] == 1:
525: (12)                       A = A.squeeze(-1)  # If just (M, N, 1), assume scalar and apply
colormap.
526: (8)                    if not (A.ndim == 2 or A.ndim == 3 and A.shape[-1] in [3, 4]):
527: (12)                       raise TypeError(f"Invalid shape {A.shape} for image data")
528: (8)                    if A.ndim == 3:
529: (12)                       high = 255 if np.issubdtype(A.dtype, np.integer) else 1
530: (12)                       if A.min() < 0 or high < A.max():
531: (16)                           _log.warning(
532: (20)                               'Clipping input data to the valid range for imshow with '
533: (20)                               'RGB data ([0..1] for floats or [0..255] for integers). '
534: (20)                               'Got range [%s..%s].',
535: (20)                              A.min(), A.max()
536: (16)                           )
537: (16)                           A = np.clip(A, 0, high)
538: (12)                       if A.dtype != np.uint8 and np.issubdtype(A.dtype, np.integer):
539: (16)                           A = A.astype(np.uint8)
```

```
(8)                 return A
(4)             def set_data(self, A):
(8)                 """
(8)                 Set the image array.
(8)                 Note that this function does *not* update the normalization used.
(8)                 Parameters
(8)                 ----------
(8)                 A : array-like or `PIL.Image.Image`
(8)                 """
(8)                 if isinstance(A, PIL.Image.Image):
(12)                    A = pil_to_array(A)  # Needed e.g. to apply png palette.
(8)                 self._A = self._normalize_image_array(A)
(8)                 self._imcache = None
(8)                 self.stale = True
(4)             def set_array(self, A):
(8)                 """
(8)                 Retained for backwards compatibility - use set_data instead.
(8)                 Parameters
(8)                 ----------
(8)                 A : array-like
(8)                 """
(8)                 self.set_data(A)
(4)             def get_interpolation(self):
(8)                 """
(8)                 Return the interpolation method the image uses when resizing.
(8)                 One of 'antialiased', 'nearest', 'bilinear', 'bicubic', 'spline16',
(8)                 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric',
(8)                 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos',
(8)                 or 'none'.
(8)                 """
(8)                 return self._interpolation
(4)             def set_interpolation(self, s):
(8)                 """
(8)                 Set the interpolation method the image uses when resizing.
(8)                 If None, use :rc:`image.interpolation`. If 'none', the image is
(8)                 shown as is without interpolating. 'none' is only supported in
(8)                 agg, ps and pdf backends and will fall back to 'nearest' mode
(8)                 for other backends.
(8)                 Parameters
(8)                 ----------
(8)                 s : {'antialiased', 'nearest', 'bilinear', 'bicubic', 'spline16', \
(0)         'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', \
(0)         'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos', 'none'} or None
(8)                 """
(8)                 s = mpl._val_or_rc(s, 'image.interpolation').lower()
(8)                 _api.check_in_list(interpolations_names, interpolation=s)
(8)                 self._interpolation = s
(8)                 self.stale = True
(4)             def get_interpolation_stage(self):
(8)                 """
(8)                 Return when interpolation happens during the transform to RGBA.
(8)                 One of 'data', 'rgba'.
(8)                 """
(8)                 return self._interpolation_stage
(4)             def set_interpolation_stage(self, s):
(8)                 """
(8)                 Set when interpolation happens during the transform to RGBA.
(8)                 Parameters
(8)                 ----------
(8)                 s : {'data', 'rgba'} or None
(12)                    Whether to apply up/downsampling interpolation in data or RGBA
(12)                    space.  If None, use :rc:`image.interpolation_stage`.
(8)                 """
(8)                 s = mpl._val_or_rc(s, 'image.interpolation_stage')
(8)                 _api.check_in_list(['data', 'rgba'], s=s)
(8)                 self._interpolation_stage = s
(8)                 self.stale = True
(4)             def can_composite(self):
(8)                 """Return whether the image can be composited with its neighbors."""
```

```
(8)                     trans = self.get_transform()
(8)                     return (
(12)                        self._interpolation != 'none' and
(12)                        trans.is_affine and
(12)                        trans.is_separable)
(4)                 def set_resample(self, v):
(8)                     """
(8)                     Set whether image resampling is used.
(8)                     Parameters
(8)                     ----------
(8)                     v : bool or None
(12)                        If None, use :rc:`image.resample`.
(8)                     """
(8)                     v = mpl._val_or_rc(v, 'image.resample')
(8)                     self._resample = v
(8)                     self.stale = True
(4)                 def get_resample(self):
(8)                     """Return whether image resampling is used."""
(8)                     return self._resample
(4)                 def set_filternorm(self, filternorm):
(8)                     """
(8)                     Set whether the resize filter normalizes the weights.
(8)                     See help for `~.Axes.imshow`.
(8)                     Parameters
(8)                     ----------
(8)                     filternorm : bool
(8)                     """
(8)                     self._filternorm = bool(filternorm)
(8)                     self.stale = True
(4)                 def get_filternorm(self):
(8)                     """Return whether the resize filter normalizes the weights."""
(8)                     return self._filternorm
(4)                 def set_filterrad(self, filterrad):
(8)                     """
(8)                     Set the resize filter radius only applicable to some
(8)                     interpolation schemes -- see help for imshow
(8)                     Parameters
(8)                     ----------
(8)                     filterrad : positive float
(8)                     """
(8)                     r = float(filterrad)
(8)                     if r <= 0:
(12)                        raise ValueError("The filter radius must be a positive number")
(8)                     self._filterrad = r
(8)                     self.stale = True
(4)                 def get_filterrad(self):
(8)                     """Return the filterrad setting."""
(8)                     return self._filterrad
(0)             class AxesImage(_ImageBase):
(4)                 """
(4)                 An image attached to an Axes.
(4)                 Parameters
(4)                 ----------
(4)                 ax : `~matplotlib.axes.Axes`
(8)                     The Axes the image will belong to.
(4)                 cmap : str or `~matplotlib.colors.Colormap`, default: :rc:`image.cmap`
(8)                     The Colormap instance or registered colormap name used to map scalar
(8)                     data to colors.
(4)                 norm : str or `~matplotlib.colors.Normalize`
(8)                     Maps luminance to 0-1.
(4)                 interpolation : str, default: :rc:`image.interpolation`
(8)                     Supported values are 'none', 'antialiased', 'nearest', 'bilinear',
(8)                     'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite',
(8)                     'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell',
(8)                     'sinc', 'lanczos', 'blackman'.
(4)                 interpolation_stage : {'data', 'rgba'}, default: 'data'
(8)                     If 'data', interpolation
(8)                     is carried out on the data provided by the user.  If 'rgba', the
(8)                     interpolation is carried out after the colormapping has been
```

```
(8)                 applied (visual interpolation).
(4)             origin : {'upper', 'lower'}, default: :rc:`image.origin`
(8)                 Place the [0, 0] index of the array in the upper left or lower left
(8)                 corner of the Axes. The convention 'upper' is typically used for
(8)                 matrices and images.
(4)             extent : tuple, optional
(8)                 The data axes (left, right, bottom, top) for making image plots
(8)                 registered with data plots.  Default is to label the pixel
(8)                 centers with the zero-based row and column indices.
(4)             filternorm : bool, default: True
(8)                 A parameter for the antigrain image resize filter
(8)                 (see the antigrain documentation).
(8)                 If filternorm is set, the filter normalizes integer values and
corrects
(8)                 the rounding errors. It doesn't do anything with the source floating
(8)                 point values, it corrects only integers according to the rule of 1.0
(8)                 which means that any sum of pixel weights must be equal to 1.0. So,
(8)                 the filter function must produce a graph of the proper shape.
(4)             filterrad : float > 0, default: 4
(8)                 The filter radius for filters that have a radius parameter, i.e. when
(8)                 interpolation is one of: 'sinc', 'lanczos' or 'blackman'.
(4)             resample : bool, default: False
(8)                 When True, use a full resampling method. When False, only resample
when
(8)                 the output image is larger than the input image.
(4)             **kwargs : `~matplotlib.artist.Artist` properties
(4)             """
(4)             def __init__(self, ax,
(17)                         *,
(17)                         cmap=None,
(17)                         norm=None,
(17)                         interpolation=None,
(17)                         origin=None,
(17)                         extent=None,
(17)                         filternorm=True,
(17)                         filterrad=4.0,
(17)                         resample=False,
(17)                         interpolation_stage=None,
(17)                         **kwargs
(17)                         ):
(8)                 self._extent = extent
(8)                 super().__init__(
(12)                    ax,
(12)                    cmap=cmap,
(12)                    norm=norm,
(12)                    interpolation=interpolation,
(12)                    origin=origin,
(12)                    filternorm=filternorm,
(12)                    filterrad=filterrad,
(12)                    resample=resample,
(12)                    interpolation_stage=interpolation_stage,
(12)                    **kwargs
(8)                 )
(4)             def get_window_extent(self, renderer=None):
(8)                 x0, x1, y0, y1 = self._extent
(8)                 bbox = Bbox.from_extents([x0, y0, x1, y1])
(8)                 return bbox.transformed(self.get_transform())
(4)             def make_image(self, renderer, magnification=1.0, unsampled=False):
(8)                 trans = self.get_transform()
(8)                 x1, x2, y1, y2 = self.get_extent()
(8)                 bbox = Bbox(np.array([[x1, y1], [x2, y2]]))
(8)                 transformed_bbox = TransformedBbox(bbox, trans)
(8)                 clip = ((self.get_clip_box() or self.axes.bbox) if self.get_clip_on()
(16)                        else self.figure.bbox)
(8)                 return self._make_image(self._A, bbox, transformed_bbox, clip,
(32)                                        magnification, unsampled=unsampled)
(4)             def _check_unsampled_image(self):
(8)                 """Return whether the image would be better drawn unsampled."""
(8)                 return self.get_interpolation() == "none"
```

```
(4)     def set_extent(self, extent, **kwargs):
(8)         """
(8)         Set the image extent.
(8)         Parameters
(8)         ----------
(8)         extent : 4-tuple of float
(12)            The position and size of the image as tuple
(12)            ``(left, right, bottom, top)`` in data coordinates.
(8)         **kwargs
(12)            Other parameters from which unit info (i.e., the *xunits*,
(12)            *yunits*, *zunits* (for 3D Axes), *runits* and *thetaunits* (for
(12)            polar Axes) entries are applied, if present.
(8)         Notes
(8)         -----
(8)         This updates ``ax.dataLim``, and, if autoscaling, sets ``ax.viewLim``
(8)         to tightly fit the image, regardless of ``dataLim``.  Autoscaling
(8)         state is not changed, so following this with ``ax.autoscale_view()``
(8)         will redo the autoscaling in accord with ``dataLim``.
(8)         """
(8)         (xmin, xmax), (ymin, ymax) = self.axes._process_unit_info(
(12)            [("x", [extent[0], extent[1]]),
(13)             ("y", [extent[2], extent[3]])],
(12)            kwargs)
(8)         if kwargs:
(12)            raise _api.kwarg_error("set_extent", kwargs)
(8)         xmin = self.axes._validate_converted_limits(
(12)            xmin, self.convert_xunits)
(8)         xmax = self.axes._validate_converted_limits(
(12)            xmax, self.convert_xunits)
(8)         ymin = self.axes._validate_converted_limits(
(12)            ymin, self.convert_yunits)
(8)         ymax = self.axes._validate_converted_limits(
(12)            ymax, self.convert_yunits)
(8)         extent = [xmin, xmax, ymin, ymax]
(8)         self._extent = extent
(8)         corners = (xmin, ymin), (xmax, ymax)
(8)         self.axes.update_datalim(corners)
(8)         self.sticky_edges.x[:] = [xmin, xmax]
(8)         self.sticky_edges.y[:] = [ymin, ymax]
(8)         if self.axes.get_autoscalex_on():
(12)            self.axes.set_xlim((xmin, xmax), auto=None)
(8)         if self.axes.get_autoscaley_on():
(12)            self.axes.set_ylim((ymin, ymax), auto=None)
(8)         self.stale = True
(4)     def get_extent(self):
(8)         """Return the image extent as tuple (left, right, bottom, top)."""
(8)         if self._extent is not None:
(12)            return self._extent
(8)         else:
(12)            sz = self.get_size()
(12)            numrows, numcols = sz
(12)            if self.origin == 'upper':
(16)                return (-0.5, numcols-0.5, numrows-0.5, -0.5)
(12)            else:
(16)                return (-0.5, numcols-0.5, -0.5, numrows-0.5)
(4)     def get_cursor_data(self, event):
(8)         """
(8)         Return the image value at the event position or *None* if the event is
(8)         outside the image.
(8)         See Also
(8)         --------
(8)         matplotlib.artist.Artist.get_cursor_data
(8)         """
(8)         xmin, xmax, ymin, ymax = self.get_extent()
(8)         if self.origin == 'upper':
(12)            ymin, ymax = ymax, ymin
(8)         arr = self.get_array()
(8)         data_extent = Bbox([[xmin, ymin], [xmax, ymax]])
(8)         array_extent = Bbox([[0, 0], [arr.shape[1], arr.shape[0]]])
```

```
(8)                     trans = self.get_transform().inverted()
(8)                     trans += BboxTransform(boxin=data_extent, boxout=array_extent)
(8)                     point = trans.transform([event.x, event.y])
(8)                     if any(np.isnan(point)):
(12)                        return None
(8)                     j, i = point.astype(int)
(8)                     if not (0 <= i < arr.shape[0]) or not (0 <= j < arr.shape[1]):
(12)                        return None
(8)                     else:
(12)                        return arr[i, j]
(0)             class NonUniformImage(AxesImage):
(4)                 def __init__(self, ax, *, interpolation='nearest', **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     ax : `~matplotlib.axes.Axes`
(12)                        The Axes the image will belong to.
(8)                     interpolation : {'nearest', 'bilinear'}, default: 'nearest'
(12)                        The interpolation scheme used in the resampling.
(8)                     **kwargs
(12)                        All other keyword arguments are identical to those of
`.AxesImage`.
(8)                     """
(8)                     super().__init__(ax, **kwargs)
(8)                     self.set_interpolation(interpolation)
(4)                 def _check_unsampled_image(self):
(8)                     """Return False. Do not use unsampled image."""
(8)                     return False
(4)                 def make_image(self, renderer, magnification=1.0, unsampled=False):
(8)                     if self._A is None:
(12)                        raise RuntimeError('You must first set the image array')
(8)                     if unsampled:
(12)                        raise ValueError('unsampled not supported on NonUniformImage')
(8)                     A = self._A
(8)                     if A.ndim == 2:
(12)                        if A.dtype != np.uint8:
(16)                            A = self.to_rgba(A, bytes=True)
(12)                        else:
(16)                            A = np.repeat(A[:, :, np.newaxis], 4, 2)
(16)                            A[:, :, 3] = 255
(8)                     else:
(12)                        if A.dtype != np.uint8:
(16)                            A = (255*A).astype(np.uint8)
(12)                        if A.shape[2] == 3:
(16)                            B = np.zeros(tuple([*A.shape[0:2], 4]), np.uint8)
(16)                            B[:, :, 0:3] = A
(16)                            B[:, :, 3] = 255
(16)                            A = B
(8)                     vl = self.axes.viewLim
(8)                     l, b, r, t = self.axes.bbox.extents
(8)                     width = int(((round(r) + 0.5) - (round(l) - 0.5)) * magnification)
(8)                     height = int(((round(t) + 0.5) - (round(b) - 0.5)) * magnification)
(8)                     x_pix = np.linspace(vl.x0, vl.x1, width)
(8)                     y_pix = np.linspace(vl.y0, vl.y1, height)
(8)                     if self._interpolation == "nearest":
(12)                        x_mid = (self._Ax[:-1] + self._Ax[1:]) / 2
(12)                        y_mid = (self._Ay[:-1] + self._Ay[1:]) / 2
(12)                        x_int = x_mid.searchsorted(x_pix)
(12)                        y_int = y_mid.searchsorted(y_pix)
(12)                        im = (
(16)                            np.ascontiguousarray(A).view(np.uint32).ravel()[
(20)                                np.add.outer(y_int * A.shape[1], x_int)]
(16)                            .view(np.uint8).reshape((height, width, 4)))
(8)                     else:  # self._interpolation == "bilinear"
(12)                        x_int = np.clip(
(16)                            self._Ax.searchsorted(x_pix) - 1, 0, len(self._Ax) - 2)
(12)                        y_int = np.clip(
(16)                            self._Ay.searchsorted(y_pix) - 1, 0, len(self._Ay) - 2)
(12)                        idx_int = np.add.outer(y_int * A.shape[1], x_int)
```

```
882: (12)                        x_frac = np.clip(
883: (16)                            np.divide(x_pix - self._Ax[x_int], np.diff(self._Ax)[x_int],
884: (26)                                      dtype=np.float32),  # Downcasting helps with speed.
885: (16)                            0, 1)
886: (12)                        y_frac = np.clip(
887: (16)                            np.divide(y_pix - self._Ay[y_int], np.diff(self._Ay)[y_int],
888: (26)                                      dtype=np.float32),
889: (16)                            0, 1)
890: (12)                        f00 = np.outer(1 - y_frac, 1 - x_frac)
891: (12)                        f10 = np.outer(y_frac, 1 - x_frac)
892: (12)                        f01 = np.outer(1 - y_frac, x_frac)
893: (12)                        f11 = np.outer(y_frac, x_frac)
894: (12)                        im = np.empty((height, width, 4), np.uint8)
895: (12)                        for chan in range(4):
896: (16)                            ac = A[:, :, chan].reshape(-1)  # reshape(-1) avoids a copy.
897: (16)                            buf = f00 * ac[idx_int]
898: (16)                            buf += f10 * ac[A.shape[1]:][idx_int]
899: (16)                            buf += f01 * ac[1:][idx_int]
900: (16)                            buf += f11 * ac[A.shape[1] + 1:][idx_int]
901: (16)                            im[:, :, chan] = buf  # Implicitly casts to uint8.
902: (8)                    return im, l, b, IdentityTransform()
903: (4)                def set_data(self, x, y, A):
904: (8)                    """
905: (8)                    Set the grid for the pixel centers, and the pixel values.
906: (8)                    Parameters
907: (8)                    ----------
908: (8)                    x, y : 1D array-like
909: (12)                       Monotonic arrays of shapes (N,) and (M,), respectively, specifying
910: (12)                       pixel centers.
911: (8)                    A : array-like
912: (12)                       (M, N) `~numpy.ndarray` or masked array of values to be
913: (12)                       colormapped, or (M, N, 3) RGB array, or (M, N, 4) RGBA array.
914: (8)                    """
915: (8)                    A = self._normalize_image_array(A)
916: (8)                    x = np.array(x, np.float32)
917: (8)                    y = np.array(y, np.float32)
918: (8)                    if not (x.ndim == y.ndim == 1 and A.shape[:2] == y.shape + x.shape):
919: (12)                       raise TypeError("Axes don't match array shape")
920: (8)                    self._A = A
921: (8)                    self._Ax = x
922: (8)                    self._Ay = y
923: (8)                    self._imcache = None
924: (8)                    self.stale = True
925: (4)                def set_array(self, *args):
926: (8)                    raise NotImplementedError('Method not supported')
927: (4)                def set_interpolation(self, s):
928: (8)                    """
929: (8)                    Parameters
930: (8)                    ----------
931: (8)                    s : {'nearest', 'bilinear'} or None
932: (12)                       If None, use :rc:`image.interpolation`.
933: (8)                    """
934: (8)                    if s is not None and s not in ('nearest', 'bilinear'):
935: (12)                       raise NotImplementedError('Only nearest neighbor and '
936: (38)                                                 'bilinear interpolations are supported')
937: (8)                    super().set_interpolation(s)
938: (4)                def get_extent(self):
939: (8)                    if self._A is None:
940: (12)                       raise RuntimeError('Must set data first')
941: (8)                    return self._Ax[0], self._Ax[-1], self._Ay[0], self._Ay[-1]
942: (4)                @_api.rename_parameter("3.8", "s", "filternorm")
943: (4)                def set_filternorm(self, filternorm):
944: (8)                    pass
945: (4)                @_api.rename_parameter("3.8", "s", "filterrad")
946: (4)                def set_filterrad(self, filterrad):
947: (8)                    pass
948: (4)                def set_norm(self, norm):
949: (8)                    if self._A is not None:
950: (12)                       raise RuntimeError('Cannot change colors after loading data')
```

```
(8)                 super().set_norm(norm)
(4)             def set_cmap(self, cmap):
(8)                 if self._A is not None:
(12)                    raise RuntimeError('Cannot change colors after loading data')
(8)                 super().set_cmap(cmap)
(4)             def get_cursor_data(self, event):
(8)                 x, y = event.xdata, event.ydata
(8)                 if (x < self._Ax[0] or x > self._Ax[-1] or
(16)                        y < self._Ay[0] or y > self._Ay[-1]):
(12)                    return None
(8)                 j = np.searchsorted(self._Ax, x) - 1
(8)                 i = np.searchsorted(self._Ay, y) - 1
(8)                 return self._A[i, j]
(0)         class PcolorImage(AxesImage):
(4)             """
(4)             Make a pcolor-style plot with an irregular rectangular grid.
(4)             This uses a variation of the original irregular image code,
(4)             and it is used by pcolorfast for the corresponding grid type.
(4)             """
(4)             def __init__(self, ax,
(17)                         x=None,
(17)                         y=None,
(17)                         A=None,
(17)                         *,
(17)                         cmap=None,
(17)                         norm=None,
(17)                         **kwargs
(17)                         ):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 ax : `~matplotlib.axes.Axes`
(12)                    The Axes the image will belong to.
(8)                 x, y : 1D array-like, optional
(12)                    Monotonic arrays of length N+1 and M+1, respectively, specifying
(12)                    rectangle boundaries.  If not given, will default to
(12)                    ``range(N + 1)`` and ``range(M + 1)``, respectively.
(8)                 A : array-like
(12)                    The data to be color-coded. The interpretation depends on the
(12)                    shape:
(12)                    - (M, N) `~numpy.ndarray` or masked array: values to be
colormapped
(12)                    - (M, N, 3): RGB array
(12)                    - (M, N, 4): RGBA array
(8)                 cmap : str or `~matplotlib.colors.Colormap`, default: :rc:`image.cmap`
(12)                    The Colormap instance or registered colormap name used to map
(12)                    scalar data to colors.
(8)                 norm : str or `~matplotlib.colors.Normalize`
(12)                    Maps luminance to 0-1.
(8)                 **kwargs : `~matplotlib.artist.Artist` properties
(8)                 """
(8)                 super().__init__(ax, norm=norm, cmap=cmap)
(8)                 self._internal_update(kwargs)
(8)                 if A is not None:
(12)                    self.set_data(x, y, A)
(4)             def make_image(self, renderer, magnification=1.0, unsampled=False):
(8)                 if self._A is None:
(12)                    raise RuntimeError('You must first set the image array')
(8)                 if unsampled:
(12)                    raise ValueError('unsampled not supported on PColorImage')
(8)                 if self._imcache is None:
(12)                    A = self.to_rgba(self._A, bytes=True)
(12)                    self._imcache = np.pad(A, [(1, 1), (1, 1), (0, 0)], "constant")
(8)                 padded_A = self._imcache
(8)                 bg = mcolors.to_rgba(self.axes.patch.get_facecolor(), 0)
(8)                 bg = (np.array(bg) * 255).astype(np.uint8)
(8)                 if (padded_A[0, 0] != bg).all():
(12)                    padded_A[[0, -1], :] = padded_A[:, [0, -1]] = bg
(8)                 l, b, r, t = self.axes.bbox.extents
```

```
1019: (8)                    width = (round(r) + 0.5) - (round(l) - 0.5)
1020: (8)                    height = (round(t) + 0.5) - (round(b) - 0.5)
1021: (8)                    width = round(width * magnification)
1022: (8)                    height = round(height * magnification)
1023: (8)                    vl = self.axes.viewLim
1024: (8)                    x_pix = np.linspace(vl.x0, vl.x1, width)
1025: (8)                    y_pix = np.linspace(vl.y0, vl.y1, height)
1026: (8)                    x_int = self._Ax.searchsorted(x_pix)
1027: (8)                    y_int = self._Ay.searchsorted(y_pix)
1028: (8)                    im = (  # See comment in NonUniformImage.make_image re: performance.
1029: (12)                       padded_A.view(np.uint32).ravel()[
1030: (16)                           np.add.outer(y_int * padded_A.shape[1], x_int)]
1031: (12)                       .view(np.uint8).reshape((height, width, 4)))
1032: (8)                    return im, l, b, IdentityTransform()
1033: (4)                def _check_unsampled_image(self):
1034: (8)                    return False
1035: (4)                def set_data(self, x, y, A):
1036: (8)                    """
1037: (8)                    Set the grid for the rectangle boundaries, and the data values.
1038: (8)                    Parameters
1039: (8)                    ----------
1040: (8)                    x, y : 1D array-like, optional
1041: (12)                       Monotonic arrays of length N+1 and M+1, respectively, specifying
1042: (12)                       rectangle boundaries.  If not given, will default to
1043: (12)                       ``range(N + 1)`` and ``range(M + 1)``, respectively.
1044: (8)                    A : array-like
1045: (12)                       The data to be color-coded. The interpretation depends on the
1046: (12)                       shape:
1047: (12)                       - (M, N) `~numpy.ndarray` or masked array: values to be
colormapped
1048: (12)                       - (M, N, 3): RGB array
1049: (12)                       - (M, N, 4): RGBA array
1050: (8)                    """
1051: (8)                    A = self._normalize_image_array(A)
1052: (8)                    x = np.arange(0., A.shape[1] + 1) if x is None else np.array(x,
float).ravel()
1053: (8)                    y = np.arange(0., A.shape[0] + 1) if y is None else np.array(y,
float).ravel()
1054: (8)                    if A.shape[:2] != (y.size - 1, x.size - 1):
1055: (12)                       raise ValueError(
1056: (16)                           "Axes don't match array shape. Got %s, expected %s." %
1057: (16)                           (A.shape[:2], (y.size - 1, x.size - 1)))
1058: (8)                    if x[-1] < x[0]:
1059: (12)                       x = x[::-1]
1060: (12)                       A = A[:, ::-1]
1061: (8)                    if y[-1] < y[0]:
1062: (12)                       y = y[::-1]
1063: (12)                       A = A[::-1]
1064: (8)                    self._A = A
1065: (8)                    self._Ax = x
1066: (8)                    self._Ay = y
1067: (8)                    self._imcache = None
1068: (8)                    self.stale = True
1069: (4)                def set_array(self, *args):
1070: (8)                    raise NotImplementedError('Method not supported')
1071: (4)                def get_cursor_data(self, event):
1072: (8)                    x, y = event.xdata, event.ydata
1073: (8)                    if (x < self._Ax[0] or x > self._Ax[-1] or
1074: (16)                           y < self._Ay[0] or y > self._Ay[-1]):
1075: (12)                       return None
1076: (8)                    j = np.searchsorted(self._Ax, x) - 1
1077: (8)                    i = np.searchsorted(self._Ay, y) - 1
1078: (8)                    return self._A[i, j]
1079: (0)            class FigureImage(_ImageBase):
1080: (4)                """An image attached to a figure."""
1081: (4)                zorder = 0
1082: (4)                _interpolation = 'nearest'
1083: (4)                def __init__(self, fig,
1084: (17)                             *,
```

```
(17)                            cmap=None,
(17)                            norm=None,
(17)                            offsetx=0,
(17)                            offsety=0,
(17)                            origin=None,
(17)                            **kwargs
(17)                            ):
(8)                     """
(8)                     cmap is a colors.Colormap instance
(8)                     norm is a colors.Normalize instance to map luminance to 0-1
(8)                     kwargs are an optional list of Artist keyword args
(8)                     """
(8)                     super().__init__(
(12)                        None,
(12)                        norm=norm,
(12)                        cmap=cmap,
(12)                        origin=origin
(8)                     )
(8)                     self.figure = fig
(8)                     self.ox = offsetx
(8)                     self.oy = offsety
(8)                     self._internal_update(kwargs)
(8)                     self.magnification = 1.0
(4)                 def get_extent(self):
(8)                     """Return the image extent as tuple (left, right, bottom, top)."""
(8)                     numrows, numcols = self.get_size()
(8)                     return (-0.5 + self.ox, numcols-0.5 + self.ox,
(16)                            -0.5 + self.oy, numrows-0.5 + self.oy)
(4)                 def make_image(self, renderer, magnification=1.0, unsampled=False):
(8)                     fac = renderer.dpi/self.figure.dpi
(8)                     bbox = Bbox([[self.ox/fac, self.oy/fac],
(21)                                 [(self.ox/fac + self._A.shape[1]),
(21)                                 (self.oy/fac + self._A.shape[0])]])
(8)                     width, height = self.figure.get_size_inches()
(8)                     width *= renderer.dpi
(8)                     height *= renderer.dpi
(8)                     clip = Bbox([[0, 0], [width, height]])
(8)                     return self._make_image(
(12)                        self._A, bbox, bbox, clip, magnification=magnification / fac,
(12)                        unsampled=unsampled, round_to_pixel_border=False)
(4)                 def set_data(self, A):
(8)                     """Set the image array."""
(8)                     cm.ScalarMappable.set_array(self, A)
(8)                     self.stale = True
(0)             class BboxImage(_ImageBase):
(4)                 """The Image class whose size is determined by the given bbox."""
(4)                 def __init__(self, bbox,
(17)                            *,
(17)                            cmap=None,
(17)                            norm=None,
(17)                            interpolation=None,
(17)                            origin=None,
(17)                            filternorm=True,
(17)                            filterrad=4.0,
(17)                            resample=False,
(17)                            **kwargs
(17)                            ):
(8)                     """
(8)                     cmap is a colors.Colormap instance
(8)                     norm is a colors.Normalize instance to map luminance to 0-1
(8)                     kwargs are an optional list of Artist keyword args
(8)                     """
(8)                     super().__init__(
(12)                        None,
(12)                        cmap=cmap,
(12)                        norm=norm,
(12)                        interpolation=interpolation,
(12)                        origin=origin,
(12)                        filternorm=filternorm,
```

```
(12)                        filterrad=filterrad,
(12)                        resample=resample,
(12)                        **kwargs
(8)                     )
(8)                     self.bbox = bbox
(4)                 def get_window_extent(self, renderer=None):
(8)                     if renderer is None:
(12)                        renderer = self.get_figure()._get_renderer()
(8)                     if isinstance(self.bbox, BboxBase):
(12)                        return self.bbox
(8)                     elif callable(self.bbox):
(12)                        return self.bbox(renderer)
(8)                     else:
(12)                        raise ValueError("Unknown type of bbox")
(4)                 def contains(self, mouseevent):
(8)                     """Test whether the mouse event occurred within the image."""
(8)                     if self._different_canvas(mouseevent) or not self.get_visible():
(12)                        return False, {}
(8)                     x, y = mouseevent.x, mouseevent.y
(8)                     inside = self.get_window_extent().contains(x, y)
(8)                     return inside, {}
(4)                 def make_image(self, renderer, magnification=1.0, unsampled=False):
(8)                     width, height = renderer.get_canvas_width_height()
(8)                     bbox_in = self.get_window_extent(renderer).frozen()
(8)                     bbox_in._points /= [width, height]
(8)                     bbox_out = self.get_window_extent(renderer)
(8)                     clip = Bbox([[0, 0], [width, height]])
(8)                     self._transform = BboxTransformTo(clip)
(8)                     return self._make_image(
(12)                        self._A,
(12)                        bbox_in, bbox_out, clip, magnification, unsampled=unsampled)
(0)             def imread(fname, format=None):
(4)                 """
(4)                 Read an image from a file into an array.
(4)                 .. note::
(8)                     This function exists for historical reasons.  It is recommended to
(8)                     use `PIL.Image.open` instead for loading images.
(4)                 Parameters
(4)                 ----------
(4)                 fname : str or file-like
(8)                     The image file to read: a filename, a URL or a file-like object opened
(8)                     in read-binary mode.
(8)                     Passing a URL is deprecated.  Please open the URL
(8)                     for reading and pass the result to Pillow, e.g. with
(8)                     ``np.array(PIL.Image.open(urllib.request.urlopen(url)))``.
(4)                 format : str, optional
(8)                     The image file format assumed for reading the data.  The image is
(8)                     loaded as a PNG file if *format* is set to "png", if *fname* is a path
(8)                     or opened file with a ".png" extension, or if it is a URL.  In all
(8)                     other cases, *format* is ignored and the format is auto-detected by
(8)                     `PIL.Image.open`.
(4)                 Returns
(4)                 -------
(4)                 `numpy.array`
(8)                     The image data. The returned array has shape
(8)                     - (M, N) for grayscale images.
(8)                     - (M, N, 3) for RGB images.
(8)                     - (M, N, 4) for RGBA images.
(8)                     PNG images are returned as float arrays (0-1).  All other formats are
(8)                     returned as int arrays, with a bit depth determined by the file's
(8)                     contents.
(4)                 """
(4)                 from urllib import parse
(4)                 if format is None:
(8)                     if isinstance(fname, str):
(12)                        parsed = parse.urlparse(fname)
(12)                        if len(parsed.scheme) > 1:
(16)                            ext = 'png'
(12)                        else:
```

```
(16)                        ext = Path(fname).suffix.lower()[1:]
(8)                 elif hasattr(fname, 'geturl'):  # Returned by urlopen().
(12)                    ext = 'png'
(8)                 elif hasattr(fname, 'name'):
(12)                    ext = Path(fname.name).suffix.lower()[1:]
(8)                 else:
(12)                    ext = 'png'
(4)             else:
(8)                 ext = format
(4)             img_open = (
(8)                 PIL.PngImagePlugin.PngImageFile if ext == 'png' else PIL.Image.open)
(4)             if isinstance(fname, str) and len(parse.urlparse(fname).scheme) > 1:
(8)                 raise ValueError(
(12)                    "Please open the URL for reading and pass the "
(12)                    "result to Pillow, e.g. with "
(12)                    "``np.array(PIL.Image.open(urllib.request.urlopen(url)))``."
(12)                    )
(4)             with img_open(fname) as image:
(8)                 return (_pil_png_to_float_array(image)
(16)                        if isinstance(image, PIL.PngImagePlugin.PngImageFile) else
(16)                        pil_to_array(image))
(0)         def imsave(fname, arr, vmin=None, vmax=None, cmap=None, format=None,
(11)                   origin=None, dpi=100, *, metadata=None, pil_kwargs=None):
(4)             """
(4)             Colormap and save an array as an image file.
(4)             RGB(A) images are passed through.  Single channel images will be
(4)             colormapped according to *cmap* and *norm*.
(4)             .. note::
(7)                If you want to save a single channel image as gray scale please use an
(7)                image I/O library (such as pillow, tifffile, or imageio) directly.
(4)             Parameters
(4)             ----------
(4)             fname : str or path-like or file-like
(8)                 A path or a file-like object to store the image in.
(8)                 If *format* is not set, then the output format is inferred from the
(8)                 extension of *fname*, if any, and from :rc:`savefig.format` otherwise.
(8)                 If *format* is set, it determines the output format.
(4)             arr : array-like
(8)                 The image data. The shape can be one of
(8)                 MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA).
(4)             vmin, vmax : float, optional
(8)                 *vmin* and *vmax* set the color scaling for the image by fixing the
(8)                 values that map to the colormap color limits. If either *vmin*
(8)                 or *vmax* is None, that limit is determined from the *arr*
(8)                 min/max value.
(4)             cmap : str or `~matplotlib.colors.Colormap`, default: :rc:`image.cmap`
(8)                 A Colormap instance or registered colormap name. The colormap
(8)                 maps scalar data to colors. It is ignored for RGB(A) data.
(4)             format : str, optional
(8)                 The file format, e.g. 'png', 'pdf', 'svg', ...  The behavior when this
(8)                 is unset is documented under *fname*.
(4)             origin : {'upper', 'lower'}, default: :rc:`image.origin`
(8)                 Indicates whether the ``(0, 0)`` index of the array is in the upper
(8)                 left or lower left corner of the Axes.
(4)             dpi : float
(8)                 The DPI to store in the metadata of the file.  This does not affect
the
(8)                 resolution of the output image.  Depending on file format, this may be
(8)                 rounded to the nearest integer.
(4)             metadata : dict, optional
(8)                 Metadata in the image file.  The supported keys depend on the output
(8)                 format, see the documentation of the respective backends for more
(8)                 information.
(8)                 Currently only supported for "png", "pdf", "ps", "eps", and "svg".
(4)             pil_kwargs : dict, optional
(8)                 Keyword arguments passed to `PIL.Image.Image.save`.  If the 'pnginfo'
(8)                 key is present, it completely overrides *metadata*, including the
(8)                 default 'Software' key.
(4)             """
```

```
(4)                 from matplotlib.figure import Figure
(4)                 if isinstance(fname, os.PathLike):
(8)                     fname = os.fspath(fname)
(4)                 if format is None:
(8)                     format = (Path(fname).suffix[1:] if isinstance(fname, str)
(18)                              else mpl.rcParams["savefig.format"]).lower()
(4)                 if format in ["pdf", "ps", "eps", "svg"]:
(8)                     if pil_kwargs is not None:
(12)                        raise ValueError(
(16)                            f"Cannot use 'pil_kwargs' when saving to {format}")
(8)                     fig = Figure(dpi=dpi, frameon=False)
(8)                     fig.figimage(arr, cmap=cmap, vmin=vmin, vmax=vmax, origin=origin,
(21)                                 resize=True)
(8)                     fig.savefig(fname, dpi=dpi, format=format, transparent=True,
(20)                                metadata=metadata)
(4)                 else:
(8)                     if origin is None:
(12)                        origin = mpl.rcParams["image.origin"]
(8)                     else:
(12)                        _api.check_in_list(('upper', 'lower'), origin=origin)
(8)                     if origin == "lower":
(12)                        arr = arr[::-1]
(8)                     if (isinstance(arr, memoryview) and arr.format == "B"
(16)                            and arr.ndim == 3 and arr.shape[-1] == 4):
(12)                        rgba = arr
(8)                     else:
(12)                        sm = cm.ScalarMappable(cmap=cmap)
(12)                        sm.set_clim(vmin, vmax)
(12)                        rgba = sm.to_rgba(arr, bytes=True)
(8)                     if pil_kwargs is None:
(12)                        pil_kwargs = {}
(8)                     else:
(12)                        pil_kwargs = pil_kwargs.copy()
(8)                     pil_shape = (rgba.shape[1], rgba.shape[0])
(8)                     rgba = np.require(rgba, requirements='C')
(8)                     image = PIL.Image.frombuffer(
(12)                        "RGBA", pil_shape, rgba, "raw", "RGBA", 0, 1)
(8)                     if format == "png":
(12)                        if "pnginfo" in pil_kwargs:
(16)                            if metadata:
(20)                                _api.warn_external("'metadata' is overridden by the "
(39)                                                   "'pnginfo' entry in 'pil_kwargs'.")
(12)                        else:
(16)                            metadata = {
(20)                                "Software": (f"Matplotlib version{mpl.__version__}, "
(33)                                            f"https://matplotlib.org/"),
(20)                                **(metadata if metadata is not None else {}),
(16)                            }
(16)                            pil_kwargs["pnginfo"] = pnginfo = PIL.PngImagePlugin.PngInfo()
(16)                            for k, v in metadata.items():
(20)                                if v is not None:
(24)                                    pnginfo.add_text(k, v)
(8)                     elif metadata is not None:
(12)                        raise ValueError(f"metadata not supported for format {format!r}")
(8)                     if format in ["jpg", "jpeg"]:
(12)                        format = "jpeg"  # Pillow doesn't recognize "jpg".
(12)                        facecolor = mpl.rcParams["savefig.facecolor"]
(12)                        if cbook._str_equal(facecolor, "auto"):
(16)                            facecolor = mpl.rcParams["figure.facecolor"]
(12)                        color = tuple(int(x * 255) for x in mcolors.to_rgb(facecolor))
(12)                        background = PIL.Image.new("RGB", pil_shape, color)
(12)                        background.paste(image, image)
(12)                        image = background
(8)                     pil_kwargs.setdefault("format", format)
(8)                     pil_kwargs.setdefault("dpi", (dpi, dpi))
(8)                     image.save(fname, **pil_kwargs)
(0)             def pil_to_array(pilImage):
(4)                 """
(4)                 Load a `PIL image`_ and return it as a numpy int array.
```

```
(4)                .. _PIL image:
https://pillow.readthedocs.io/en/latest/reference/Image.html
(4)                Returns
(4)                -------
(4)                numpy.array
(8)                    The array shape depends on the image type:
(8)                    - (M, N) for grayscale images.
(8)                    - (M, N, 3) for RGB images.
(8)                    - (M, N, 4) for RGBA images.
(4)                """
(4)                if pilImage.mode in ['RGBA', 'RGBX', 'RGB', 'L']:
(8)                    return np.asarray(pilImage)
(4)                elif pilImage.mode.startswith('I;16'):
(8)                    raw = pilImage.tobytes('raw', pilImage.mode)
(8)                    if pilImage.mode.endswith('B'):
(12)                       x = np.frombuffer(raw, '>u2')
(8)                    else:
(12)                       x = np.frombuffer(raw, '<u2')
(8)                    return x.reshape(pilImage.size[::-1]).astype('=u2')
(4)                else:  # try to convert to an rgba image
(8)                    try:
(12)                       pilImage = pilImage.convert('RGBA')
(8)                    except ValueError as err:
(12)                       raise RuntimeError('Unknown image mode') from err
(8)                    return np.asarray(pilImage)  # return MxNx4 RGBA array
(0)            def _pil_png_to_float_array(pil_png):
(4)                """Convert a PIL `PNGImageFile` to a 0-1 float array."""
(4)                mode = pil_png.mode
(4)                rawmode = pil_png.png.im_rawmode
(4)                if rawmode == "1":  # Grayscale.
(8)                    return np.asarray(pil_png, np.float32)
(4)                if rawmode == "L;2":  # Grayscale.
(8)                    return np.divide(pil_png, 2**2 - 1, dtype=np.float32)
(4)                if rawmode == "L;4":  # Grayscale.
(8)                    return np.divide(pil_png, 2**4 - 1, dtype=np.float32)
(4)                if rawmode == "L":  # Grayscale.
(8)                    return np.divide(pil_png, 2**8 - 1, dtype=np.float32)
(4)                if rawmode == "I;16B":  # Grayscale.
(8)                    return np.divide(pil_png, 2**16 - 1, dtype=np.float32)
(4)                if mode == "RGB":  # RGB.
(8)                    return np.divide(pil_png, 2**8 - 1, dtype=np.float32)
(4)                if mode == "P":  # Palette.
(8)                    return np.divide(pil_png.convert("RGBA"), 2**8 - 1, dtype=np.float32)
(4)                if mode == "LA":  # Grayscale + alpha.
(8)                    return np.divide(pil_png.convert("RGBA"), 2**8 - 1, dtype=np.float32)
(4)                if mode == "RGBA":  # RGBA.
(8)                    return np.divide(pil_png, 2**8 - 1, dtype=np.float32)
(4)                raise ValueError(f"Unknown PIL rawmode: {rawmode}")
(0)            def thumbnail(infile, thumbfile, scale=0.1, interpolation='bilinear',
(14)                      preview=False):
(4)                """
(4)                Make a thumbnail of image in *infile* with output filename *thumbfile*.
(4)                See :doc:`/gallery/misc/image_thumbnail_sgskip`.
(4)                Parameters
(4)                ----------
(4)                infile : str or file-like
(8)                    The image file. Matplotlib relies on Pillow_ for image reading, and
(8)                    thus supports a wide range of file formats, including PNG, JPG, TIFF
(8)                    and others.
(8)                    .. _Pillow: https://python-pillow.org/
(4)                thumbfile : str or file-like
(8)                    The thumbnail filename.
(4)                scale : float, default: 0.1
(8)                    The scale factor for the thumbnail.
(4)                interpolation : str, default: 'bilinear'
(8)                    The interpolation scheme used in the resampling. See the
(8)                    *interpolation* parameter of `~.Axes.imshow` for possible values.
(4)                preview : bool, default: False
(8)                    If True, the default backend (presumably a user interface
```

```
1428: (8)                   backend) will be used which will cause a figure to be raised if
1429: (8)                   `~matplotlib.pyplot.show` is called.  If it is False, the figure is
1430: (8)                   created using `.FigureCanvasBase` and the drawing backend is selected
1431: (8)                   as `.Figure.savefig` would normally do.
1432: (4)               Returns
1433: (4)               -------
1434: (4)               `.Figure`
1435: (8)                   The figure instance containing the thumbnail.
1436: (4)               """
1437: (4)               im = imread(infile)
1438: (4)               rows, cols, depth = im.shape
1439: (4)               dpi = 100
1440: (4)               height = rows / dpi * scale
1441: (4)               width = cols / dpi * scale
1442: (4)               if preview:
1443: (8)                   import matplotlib.pyplot as plt
1444: (8)                   fig = plt.figure(figsize=(width, height), dpi=dpi)
1445: (4)               else:
1446: (8)                   from matplotlib.figure import Figure
1447: (8)                   fig = Figure(figsize=(width, height), dpi=dpi)
1448: (8)                   FigureCanvasBase(fig)
1449: (4)               ax = fig.add_axes([0, 0, 1, 1], aspect='auto',
1450: (22)                                frameon=False, xticks=[], yticks=[])
1451: (4)               ax.imshow(im, aspect='auto', resample=True, interpolation=interpolation)
1452: (4)               fig.savefig(thumbfile, dpi=dpi)
1453: (4)               return fig
```

File 43 - layout_engine.py:

```
1: (0)              """
2: (0)              Classes to layout elements in a `.Figure`.
3: (0)              Figures have a ``layout_engine`` property that holds a subclass of
4: (0)              `~.LayoutEngine` defined here (or *None* for no layout).  At draw time
5: (0)              ``figure.get_layout_engine().execute()`` is called, the goal of which is
6: (0)              usually to rearrange Axes on the figure to produce a pleasing layout. This is
7: (0)              like a ``draw`` callback but with two differences.  First, when printing we
8: (0)              disable the layout engine for the final draw. Second, it is useful to know the
9: (0)              layout engine while the figure is being created.  In particular, colorbars are
10: (0)             made differently with different layout engines (for historical reasons).
11: (0)             Matplotlib supplies two layout engines, `.TightLayoutEngine` and
12: (0)             `.ConstrainedLayoutEngine`.  Third parties can create their own layout engine
13: (0)             by subclassing `.LayoutEngine`.
14: (0)             """
15: (0)             from contextlib import nullcontext
16: (0)             import matplotlib as mpl
17: (0)             from matplotlib._constrained_layout import do_constrained_layout
18: (0)             from matplotlib._tight_layout import (get_subplotspec_list,
19: (38)                                                  get_tight_layout_figure)
20: (0)             class LayoutEngine:
21: (4)                 """
22: (4)                 Base class for Matplotlib layout engines.
23: (4)                 A layout engine can be passed to a figure at instantiation or at any time
24: (4)                 with `~.figure.Figure.set_layout_engine`.  Once attached to a figure, the
25: (4)                 layout engine ``execute`` function is called at draw time by
26: (4)                 `~.figure.Figure.draw`, providing a special draw-time hook.
27: (4)                 .. note::
28: (7)                    However, note that layout engines affect the creation of colorbars, so
29: (7)                    `~.figure.Figure.set_layout_engine` should be called before any
30: (7)                    colorbars are created.
31: (4)                 Currently, there are two properties of `LayoutEngine` classes that are
32: (4)                 consulted while manipulating the figure:
33: (4)                 - ``engine.colorbar_gridspec`` tells `.Figure.colorbar` whether to make
the
34: (7)                    axes using the gridspec method (see `.colorbar.make_axes_gridspec`) or
35: (7)                    not (see `.colorbar.make_axes`);
36: (4)                 - ``engine.adjust_compatible`` stops `.Figure.subplots_adjust` from being
37: (8)                     run if it is not compatible with the layout engine.
```

```
38: (4)                 To implement a custom `LayoutEngine`:
39: (4)                 1. override ``_adjust_compatible`` and ``_colorbar_gridspec``
40: (4)                 2. override `LayoutEngine.set` to update *self._params*
41: (4)                 3. override `LayoutEngine.execute` with your implementation
42: (4)                 """
43: (4)                 _adjust_compatible = None
44: (4)                 _colorbar_gridspec = None
45: (4)                 def __init__(self, **kwargs):
46: (8)                     super().__init__(**kwargs)
47: (8)                     self._params = {}
48: (4)                 def set(self, **kwargs):
49: (8)                     """
50: (8)                     Set the parameters for the layout engine.
51: (8)                     """
52: (8)                     raise NotImplementedError
53: (4)                 @property
54: (4)                 def colorbar_gridspec(self):
55: (8)                     """
56: (8)                     Return a boolean if the layout engine creates colorbars using a
57: (8)                     gridspec.
58: (8)                     """
59: (8)                     if self._colorbar_gridspec is None:
60: (12)                        raise NotImplementedError
61: (8)                     return self._colorbar_gridspec
62: (4)                 @property
63: (4)                 def adjust_compatible(self):
64: (8)                     """
65: (8)                     Return a boolean if the layout engine is compatible with
66: (8)                     `~.Figure.subplots_adjust`.
67: (8)                     """
68: (8)                     if self._adjust_compatible is None:
69: (12)                        raise NotImplementedError
70: (8)                     return self._adjust_compatible
71: (4)                 def get(self):
72: (8)                     """
73: (8)                     Return copy of the parameters for the layout engine.
74: (8)                     """
75: (8)                     return dict(self._params)
76: (4)                 def execute(self, fig):
77: (8)                     """
78: (8)                     Execute the layout on the figure given by *fig*.
79: (8)                     """
80: (8)                     raise NotImplementedError
81: (0)             class PlaceHolderLayoutEngine(LayoutEngine):
82: (4)                 """
83: (4)                 This layout engine does not adjust the figure layout at all.
84: (4)                 The purpose of this `.LayoutEngine` is to act as a placeholder when the
user removes
85: (4)                 a layout engine to ensure an incompatible `.LayoutEngine` cannot be set
later.
86: (4)                 Parameters
87: (4)                 ----------
88: (4)                 adjust_compatible, colorbar_gridspec : bool
89: (8)                     Allow the PlaceHolderLayoutEngine to mirror the behavior of whatever
90: (8)                     layout engine it is replacing.
91: (4)                 """
92: (4)                 def __init__(self, adjust_compatible, colorbar_gridspec, **kwargs):
93: (8)                     self._adjust_compatible = adjust_compatible
94: (8)                     self._colorbar_gridspec = colorbar_gridspec
95: (8)                     super().__init__(**kwargs)
96: (4)                 def execute(self, fig):
97: (8)                     """
98: (8)                     Do nothing.
99: (8)                     """
100: (8)                    return
101: (0)            class TightLayoutEngine(LayoutEngine):
102: (4)                """
103: (4)                Implements the ``tight_layout`` geometry management.  See
104: (4)                :ref:`tight_layout_guide` for details.
```

```
(4)                 """
(4)                 _adjust_compatible = True
(4)                 _colorbar_gridspec = True
(4)                 def __init__(self, *, pad=1.08, h_pad=None, w_pad=None,
(17)                             rect=(0, 0, 1, 1), **kwargs):
(8)                     """
(8)                     Initialize tight_layout engine.
(8)                     Parameters
(8)                     ----------
(8)                     pad : float, default: 1.08
(12)                        Padding between the figure edge and the edges of subplots, as a
(12)                        fraction of the font size.
(8)                     h_pad, w_pad : float
(12)                        Padding (height/width) between edges of adjacent subplots.
(12)                        Defaults to *pad*.
(8)                     rect : tuple (left, bottom, right, top), default: (0, 0, 1, 1).
(12)                        rectangle in normalized figure coordinates that the subplots
(12)                        (including labels) will fit into.
(8)                     """
(8)                     super().__init__(**kwargs)
(8)                     for td in ['pad', 'h_pad', 'w_pad', 'rect']:
(12)                        self._params[td] = None
(8)                     self.set(pad=pad, h_pad=h_pad, w_pad=w_pad, rect=rect)
(4)                 def execute(self, fig):
(8)                     """
(8)                     Execute tight_layout.
(8)                     This decides the subplot parameters given the padding that
(8)                     will allow the Axes labels to not be covered by other labels
(8)                     and Axes.
(8)                     Parameters
(8)                     ----------
(8)                     fig : `.Figure` to perform layout on.
(8)                     See Also
(8)                     --------
(8)                     .figure.Figure.tight_layout
(8)                     .pyplot.tight_layout
(8)                     """
(8)                     info = self._params
(8)                     renderer = fig._get_renderer()
(8)                     with getattr(renderer, "_draw_disabled", nullcontext)():
(12)                        kwargs = get_tight_layout_figure(
(16)                            fig, fig.axes, get_subplotspec_list(fig.axes), renderer,
(16)                            pad=info['pad'], h_pad=info['h_pad'], w_pad=info['w_pad'],
(16)                            rect=info['rect'])
(8)                     if kwargs:
(12)                        fig.subplots_adjust(**kwargs)
(4)                 def set(self, *, pad=None, w_pad=None, h_pad=None, rect=None):
(8)                     """
(8)                     Set the pads for tight_layout.
(8)                     Parameters
(8)                     ----------
(8)                     pad : float
(12)                        Padding between the figure edge and the edges of subplots, as a
(12)                        fraction of the font size.
(8)                     w_pad, h_pad : float
(12)                        Padding (width/height) between edges of adjacent subplots.
(12)                        Defaults to *pad*.
(8)                     rect : tuple (left, bottom, right, top)
(12)                        rectangle in normalized figure coordinates that the subplots
(12)                        (including labels) will fit into.
(8)                     """
(8)                     for td in self.set.__kwdefaults__:
(12)                        if locals()[td] is not None:
(16)                            self._params[td] = locals()[td]
(0)             class ConstrainedLayoutEngine(LayoutEngine):
(4)                 """
(4)                 Implements the ``constrained_layout`` geometry management.  See
(4)                 :ref:`constrainedlayout_guide` for details.
(4)                 """
```

```
(4)     _adjust_compatible = False
(4)     _colorbar_gridspec = False
(4)     def __init__(self, *, h_pad=None, w_pad=None,
(17)                 hspace=None, wspace=None, rect=(0, 0, 1, 1),
(17)                 compress=False, **kwargs):
(8)         """
(8)         Initialize ``constrained_layout`` settings.
(8)         Parameters
(8)         ----------
(8)         h_pad, w_pad : float
(12)            Padding around the Axes elements in inches.
(12)            Default to :rc:`figure.constrained_layout.h_pad` and
(12)            :rc:`figure.constrained_layout.w_pad`.
(8)         hspace, wspace : float
(12)            Fraction of the figure to dedicate to space between the
(12)            axes.  These are evenly spread between the gaps between the Axes.
(12)            A value of 0.2 for a three-column layout would have a space
(12)            of 0.1 of the figure width between each column.
(12)            If h/wspace < h/w_pad, then the pads are used instead.
(12)            Default to :rc:`figure.constrained_layout.hspace` and
(12)            :rc:`figure.constrained_layout.wspace`.
(8)         rect : tuple of 4 floats
(12)            Rectangle in figure coordinates to perform constrained layout in
(12)            (left, bottom, width, height), each from 0-1.
(8)         compress : bool
(12)            Whether to shift Axes so that white space in between them is
(12)            removed. This is useful for simple grids of fixed-aspect Axes
(e.g.
(12)            a grid of images).  See :ref:`compressed_layout`.
(8)         """
(8)         super().__init__(**kwargs)
(8)         self.set(w_pad=mpl.rcParams['figure.constrained_layout.w_pad'],
(17)                 h_pad=mpl.rcParams['figure.constrained_layout.h_pad'],
(17)                 wspace=mpl.rcParams['figure.constrained_layout.wspace'],
(17)                 hspace=mpl.rcParams['figure.constrained_layout.hspace'],
(17)                 rect=(0, 0, 1, 1))
(8)         self.set(w_pad=w_pad, h_pad=h_pad, wspace=wspace, hspace=hspace,
(17)                 rect=rect)
(8)         self._compress = compress
(4)     def execute(self, fig):
(8)         """
(8)         Perform constrained_layout and move and resize Axes accordingly.
(8)         Parameters
(8)         ----------
(8)         fig : `.Figure` to perform layout on.
(8)         """
(8)         width, height = fig.get_size_inches()
(8)         w_pad = self._params['w_pad'] / width
(8)         h_pad = self._params['h_pad'] / height
(8)         return do_constrained_layout(fig, w_pad=w_pad, h_pad=h_pad,
(37)                                      wspace=self._params['wspace'],
(37)                                      hspace=self._params['hspace'],
(37)                                      rect=self._params['rect'],
(37)                                      compress=self._compress)
(4)     def set(self, *, h_pad=None, w_pad=None,
(12)            hspace=None, wspace=None, rect=None):
(8)         """
(8)         Set the pads for constrained_layout.
(8)         Parameters
(8)         ----------
(8)         h_pad, w_pad : float
(12)            Padding around the Axes elements in inches.
(12)            Default to :rc:`figure.constrained_layout.h_pad` and
(12)            :rc:`figure.constrained_layout.w_pad`.
(8)         hspace, wspace : float
(12)            Fraction of the figure to dedicate to space between the
(12)            axes.  These are evenly spread between the gaps between the Axes.
(12)            A value of 0.2 for a three-column layout would have a space
(12)            of 0.1 of the figure width between each column.
```

```
242: (12)                        If h/wspace < h/w_pad, then the pads are used instead.
243: (12)                        Default to :rc:`figure.constrained_layout.hspace` and
244: (12)                        :rc:`figure.constrained_layout.wspace`.
245: (8)                    rect : tuple of 4 floats
246: (12)                        Rectangle in figure coordinates to perform constrained layout in
247: (12)                        (left, bottom, width, height), each from 0-1.
248: (8)                    """
249: (8)                    for td in self.set.__kwdefaults__:
250: (12)                        if locals()[td] is not None:
251: (16)                            self._params[td] = locals()[td]
```

File 44 - legend_handler.py:

```
1: (0)              """
2: (0)              Default legend handlers.
3: (0)              .. important::
4: (4)                  This is a low-level legend API, which most end users do not need.
5: (4)                  We recommend that you are familiar with the :ref:`legend guide
6: (4)                  <legend_guide>` before reading this documentation.
7: (0)              Legend handlers are expected to be a callable object with a following
8: (0)              signature::
9: (4)                  legend_handler(legend, orig_handle, fontsize, handlebox)
10: (0)             Where *legend* is the legend itself, *orig_handle* is the original
11: (0)             plot, *fontsize* is the fontsize in pixels, and *handlebox* is an
12: (0)             `.OffsetBox` instance. Within the call, you should create relevant
13: (0)             artists (using relevant properties from the *legend* and/or
14: (0)             *orig_handle*) and add them into the *handlebox*. The artists need to
15: (0)             be scaled according to the *fontsize* (note that the size is in pixels,
16: (0)             i.e., this is dpi-scaled value).
17: (0)             This module includes definition of several legend handler classes
18: (0)             derived from the base class (HandlerBase) with the following method::
19: (4)                 def legend_artist(self, legend, orig_handle, fontsize, handlebox)
20: (0)             """
21: (0)             from itertools import cycle
22: (0)             import numpy as np
23: (0)             from matplotlib import cbook
24: (0)             from matplotlib.lines import Line2D
25: (0)             from matplotlib.patches import Rectangle
26: (0)             import matplotlib.collections as mcoll
27: (0)             def update_from_first_child(tgt, src):
28: (4)                 first_child = next(iter(src.get_children()), None)
29: (4)                 if first_child is not None:
30: (8)                     tgt.update_from(first_child)
31: (0)             class HandlerBase:
32: (4)                 """
33: (4)                 A base class for default legend handlers.
34: (4)                 The derived classes are meant to override *create_artists* method, which
35: (4)                 has the following signature::
36: (6)                   def create_artists(self, legend, orig_handle,
37: (25)                                     xdescent, ydescent, width, height, fontsize,
38: (25)                                     trans):
39: (4)                 The overridden method needs to create artists of the given
40: (4)                 transform that fits in the given dimension (xdescent, ydescent,
41: (4)                 width, height) that are scaled by fontsize if necessary.
42: (4)                 """
43: (4)                 def __init__(self, xpad=0., ypad=0., update_func=None):
44: (8)                     """
45: (8)                     Parameters
46: (8)                     ----------
47: (8)                     xpad : float, optional
48: (12)                        Padding in x-direction.
49: (8)                     ypad : float, optional
50: (12)                        Padding in y-direction.
51: (8)                     update_func : callable, optional
52: (12)                        Function for updating the legend handler properties from another
53: (12)                        legend handler, used by `~HandlerBase.update_prop`.
54: (8)                     """
```

```
(8)                     self._xpad, self._ypad = xpad, ypad
(8)                     self._update_prop_func = update_func
(4)                 def _update_prop(self, legend_handle, orig_handle):
(8)                     if self._update_prop_func is None:
(12)                        self._default_update_prop(legend_handle, orig_handle)
(8)                     else:
(12)                        self._update_prop_func(legend_handle, orig_handle)
(4)                 def _default_update_prop(self, legend_handle, orig_handle):
(8)                     legend_handle.update_from(orig_handle)
(4)                 def update_prop(self, legend_handle, orig_handle, legend):
(8)                     self._update_prop(legend_handle, orig_handle)
(8)                     legend._set_artist_props(legend_handle)
(8)                     legend_handle.set_clip_box(None)
(8)                     legend_handle.set_clip_path(None)
(4)                 def adjust_drawing_area(self, legend, orig_handle,
(28)                                        xdescent, ydescent, width, height, fontsize,
(28)                                        ):
(8)                     xdescent = xdescent - self._xpad * fontsize
(8)                     ydescent = ydescent - self._ypad * fontsize
(8)                     width = width - self._xpad * fontsize
(8)                     height = height - self._ypad * fontsize
(8)                     return xdescent, ydescent, width, height
(4)                 def legend_artist(self, legend, orig_handle,
(22)                                  fontsize, handlebox):
(8)                     """
(8)                     Return the artist that this HandlerBase generates for the given
(8)                     original artist/handle.
(8)                     Parameters
(8)                     ----------
(8)                     legend : `~matplotlib.legend.Legend`
(12)                        The legend for which these legend artists are being created.
(8)                     orig_handle : :class:`matplotlib.artist.Artist` or similar
(12)                        The object for which these legend artists are being created.
(8)                     fontsize : int
(12)                        The fontsize in pixels. The artists being created should
(12)                        be scaled according to the given fontsize.
(8)                     handlebox : `~matplotlib.offsetbox.OffsetBox`
(12)                        The box which has been created to hold this legend entry's
(12)                        artists. Artists created in the `legend_artist` method must
(12)                        be added to this handlebox inside this method.
(8)                     """
(8)                     xdescent, ydescent, width, height = self.adjust_drawing_area(
(17)                             legend, orig_handle,
(17)                             handlebox.xdescent, handlebox.ydescent,
(17)                             handlebox.width, handlebox.height,
(17)                             fontsize)
(8)                     artists = self.create_artists(legend, orig_handle,
(38)                                                   xdescent, ydescent, width, height,
(38)                                                   fontsize, handlebox.get_transform())
(8)                     for a in artists:
(12)                        handlebox.add_artist(a)
(8)                     return artists[0]
(4)                 def create_artists(self, legend, orig_handle,
(23)                                   xdescent, ydescent, width, height, fontsize,
(23)                                   trans):
(8)                     """
(8)                     Return the legend artists generated.
(8)                     Parameters
(8)                     ----------
(8)                     legend : `~matplotlib.legend.Legend`
(12)                        The legend for which these legend artists are being created.
(8)                     orig_handle : `~matplotlib.artist.Artist` or similar
(12)                        The object for which these legend artists are being created.
(8)                     xdescent, ydescent, width, height : int
(12)                        The rectangle (*xdescent*, *ydescent*, *width*, *height*) that the
(12)                        legend artists being created should fit within.
(8)                     fontsize : int
(12)                        The fontsize in pixels. The legend artists being created should
(12)                        be scaled according to the given fontsize.
```

```
(8)                     trans : `~matplotlib.transforms.Transform`
(12)                        The transform that is applied to the legend artists being created.
(12)                        Typically from unit coordinates in the handler box to screen
(12)                        coordinates.
(8)                     """
(8)                     raise NotImplementedError('Derived must override')
(0)             class HandlerNpoints(HandlerBase):
(4)                 """
(4)                 A legend handler that shows *numpoints* points in the legend entry.
(4)                 """
(4)                 def __init__(self, marker_pad=0.3, numpoints=None, **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     marker_pad : float
(12)                        Padding between points in legend entry.
(8)                     numpoints : int
(12)                        Number of points to show in legend entry.
(8)                     **kwargs
(12)                        Keyword arguments forwarded to `.HandlerBase`.
(8)                     """
(8)                     super().__init__(**kwargs)
(8)                     self._numpoints = numpoints
(8)                     self._marker_pad = marker_pad
(4)                 def get_numpoints(self, legend):
(8)                     if self._numpoints is None:
(12)                        return legend.numpoints
(8)                     else:
(12)                        return self._numpoints
(4)                 def get_xdata(self, legend, xdescent, ydescent, width, height, fontsize):
(8)                     numpoints = self.get_numpoints(legend)
(8)                     if numpoints > 1:
(12)                        pad = self._marker_pad * fontsize
(12)                        xdata = np.linspace(-xdescent + pad,
(32)                                            -xdescent + width - pad,
(32)                                            numpoints)
(12)                        xdata_marker = xdata
(8)                     else:
(12)                        xdata = [-xdescent, -xdescent + width]
(12)                        xdata_marker = [-xdescent + 0.5 * width]
(8)                     return xdata, xdata_marker
(0)             class HandlerNpointsYoffsets(HandlerNpoints):
(4)                 """
(4)                 A legend handler that shows *numpoints* in the legend, and allows them to
(4)                 be individually offset in the y-direction.
(4)                 """
(4)                 def __init__(self, numpoints=None, yoffsets=None, **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     numpoints : int
(12)                        Number of points to show in legend entry.
(8)                     yoffsets : array of floats
(12)                        Length *numpoints* list of y offsets for each point in
(12)                        legend entry.
(8)                     **kwargs
(12)                        Keyword arguments forwarded to `.HandlerNpoints`.
(8)                     """
(8)                     super().__init__(numpoints=numpoints, **kwargs)
(8)                     self._yoffsets = yoffsets
(4)                 def get_ydata(self, legend, xdescent, ydescent, width, height, fontsize):
(8)                     if self._yoffsets is None:
(12)                        ydata = height * legend._scatteryoffsets
(8)                     else:
(12)                        ydata = height * np.asarray(self._yoffsets)
(8)                     return ydata
(0)             class HandlerLine2DCompound(HandlerNpoints):
(4)                 """
(4)                 Original handler for `.Line2D` instances, that relies on combining
```

```
(4)                 a line-only with a marker-only artist.  May be deprecated in the future.
(4)                 """
(4)                 def create_artists(self, legend, orig_handle,
(23)                                   xdescent, ydescent, width, height, fontsize,
(23)                                   trans):
(8)                     xdata, xdata_marker = self.get_xdata(legend, xdescent, ydescent,
(45)                                                         width, height, fontsize)
(8)                     ydata = np.full_like(xdata, ((height - ydescent) / 2))
(8)                     legline = Line2D(xdata, ydata)
(8)                     self.update_prop(legline, orig_handle, legend)
(8)                     legline.set_drawstyle('default')
(8)                     legline.set_marker("")
(8)                     legline_marker = Line2D(xdata_marker, ydata[:len(xdata_marker)])
(8)                     self.update_prop(legline_marker, orig_handle, legend)
(8)                     legline_marker.set_linestyle('None')
(8)                     if legend.markerscale != 1:
(12)                        newsz = legline_marker.get_markersize() * legend.markerscale
(12)                        legline_marker.set_markersize(newsz)
(8)                     legline._legmarker = legline_marker
(8)                     legline.set_transform(trans)
(8)                     legline_marker.set_transform(trans)
(8)                     return [legline, legline_marker]
(0)             class HandlerLine2D(HandlerNpoints):
(4)                 """
(4)                 Handler for `.Line2D` instances.
(4)                 See Also
(4)                 --------
(4)                 HandlerLine2DCompound : An earlier handler implementation, which used one
(28)                                        artist for the line and another for the marker(s).
(4)                 """
(4)                 def create_artists(self, legend, orig_handle,
(23)                                   xdescent, ydescent, width, height, fontsize,
(23)                                   trans):
(8)                     xdata, xdata_marker = self.get_xdata(legend, xdescent, ydescent,
(45)                                                         width, height, fontsize)
(8)                     markevery = None
(8)                     if self.get_numpoints(legend) == 1:
(12)                        xdata = np.linspace(xdata[0], xdata[-1], 3)
(12)                        markevery = [1]
(8)                     ydata = np.full_like(xdata, (height - ydescent) / 2)
(8)                     legline = Line2D(xdata, ydata, markevery=markevery)
(8)                     self.update_prop(legline, orig_handle, legend)
(8)                     if legend.markerscale != 1:
(12)                        newsz = legline.get_markersize() * legend.markerscale
(12)                        legline.set_markersize(newsz)
(8)                     legline.set_transform(trans)
(8)                     return [legline]
(0)             class HandlerPatch(HandlerBase):
(4)                 """
(4)                 Handler for `.Patch` instances.
(4)                 """
(4)                 def __init__(self, patch_func=None, **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     patch_func : callable, optional
(12)                        The function that creates the legend key artist.
(12)                        *patch_func* should have the signature::
(16)                            def patch_func(legend=legend, orig_handle=orig_handle,
(31)                                           xdescent=xdescent, ydescent=ydescent,
(31)                                           width=width, height=height, fontsize=fontsize)
(12)                        Subsequently, the created artist will have its ``update_prop``
(12)                        method called and the appropriate transform will be applied.
(8)                     **kwargs
(12)                        Keyword arguments forwarded to `.HandlerBase`.
(8)                     """
(8)                     super().__init__(**kwargs)
(8)                     self._patch_func = patch_func
(4)                 def _create_patch(self, legend, orig_handle,
```

```
262: (22)                                xdescent, ydescent, width, height, fontsize):
263: (8)                     if self._patch_func is None:
264: (12)                        p = Rectangle(xy=(-xdescent, -ydescent),
265: (26)                                      width=width, height=height)
266: (8)                     else:
267: (12)                        p = self._patch_func(legend=legend, orig_handle=orig_handle,
268: (33)                                             xdescent=xdescent, ydescent=ydescent,
269: (33)                                             width=width, height=height,
fontsize=fontsize)
270: (8)                     return p
271: (4)                 def create_artists(self, legend, orig_handle,
272: (23)                                   xdescent, ydescent, width, height, fontsize, trans):
273: (8)                     p = self._create_patch(legend, orig_handle,
274: (31)                                           xdescent, ydescent, width, height, fontsize)
275: (8)                     self.update_prop(p, orig_handle, legend)
276: (8)                     p.set_transform(trans)
277: (8)                     return [p]
278: (0)             class HandlerStepPatch(HandlerBase):
279: (4)                 """
280: (4)                 Handler for `~.matplotlib.patches.StepPatch` instances.
281: (4)                 """
282: (4)                 @staticmethod
283: (4)                 def _create_patch(orig_handle, xdescent, ydescent, width, height):
284: (8)                     return Rectangle(xy=(-xdescent, -ydescent), width=width,
285: (25)                                     height=height, color=orig_handle.get_facecolor())
286: (4)                 @staticmethod
287: (4)                 def _create_line(orig_handle, width, height):
288: (8)                     legline = Line2D([0, width], [height/2, height/2],
289: (25)                                     color=orig_handle.get_edgecolor(),
290: (25)                                     linestyle=orig_handle.get_linestyle(),
291: (25)                                     linewidth=orig_handle.get_linewidth(),
292: (25)                                     )
293: (8)                     legline.set_drawstyle('default')
294: (8)                     legline.set_marker("")
295: (8)                     return legline
296: (4)                 def create_artists(self, legend, orig_handle,
297: (23)                                   xdescent, ydescent, width, height, fontsize, trans):
298: (8)                     if orig_handle.get_fill() or (orig_handle.get_hatch() is not None):
299: (12)                        p = self._create_patch(orig_handle, xdescent, ydescent, width,
300: (35)                                               height)
301: (12)                        self.update_prop(p, orig_handle, legend)
302: (8)                     else:
303: (12)                        p = self._create_line(orig_handle, width, height)
304: (8)                     p.set_transform(trans)
305: (8)                     return [p]
306: (0)             class HandlerLineCollection(HandlerLine2D):
307: (4)                 """
308: (4)                 Handler for `.LineCollection` instances.
309: (4)                 """
310: (4)                 def get_numpoints(self, legend):
311: (8)                     if self._numpoints is None:
312: (12)                        return legend.scatterpoints
313: (8)                     else:
314: (12)                        return self._numpoints
315: (4)                 def _default_update_prop(self, legend_handle, orig_handle):
316: (8)                     lw = orig_handle.get_linewidths()[0]
317: (8)                     dashes = orig_handle._us_linestyles[0]
318: (8)                     color = orig_handle.get_colors()[0]
319: (8)                     legend_handle.set_color(color)
320: (8)                     legend_handle.set_linestyle(dashes)
321: (8)                     legend_handle.set_linewidth(lw)
322: (4)                 def create_artists(self, legend, orig_handle,
323: (23)                                   xdescent, ydescent, width, height, fontsize, trans):
324: (8)                     xdata, xdata_marker = self.get_xdata(legend, xdescent, ydescent,
325: (45)                                                         width, height, fontsize)
326: (8)                     ydata = np.full_like(xdata, (height - ydescent) / 2)
327: (8)                     legline = Line2D(xdata, ydata)
328: (8)                     self.update_prop(legline, orig_handle, legend)
329: (8)                     legline.set_transform(trans)
```

```
330: (8)                    return [legline]
331: (0)            class HandlerRegularPolyCollection(HandlerNpointsYoffsets):
332: (4)                r"""Handler for `.RegularPolyCollection`\s."""
333: (4)                def __init__(self, yoffsets=None, sizes=None, **kwargs):
334: (8)                    super().__init__(yoffsets=yoffsets, **kwargs)
335: (8)                    self._sizes = sizes
336: (4)                def get_numpoints(self, legend):
337: (8)                    if self._numpoints is None:
338: (12)                       return legend.scatterpoints
339: (8)                    else:
340: (12)                       return self._numpoints
341: (4)                def get_sizes(self, legend, orig_handle,
342: (18)                             xdescent, ydescent, width, height, fontsize):
343: (8)                    if self._sizes is None:
344: (12)                       handle_sizes = orig_handle.get_sizes()
345: (12)                       if not len(handle_sizes):
346: (16)                           handle_sizes = [1]
347: (12)                       size_max = max(handle_sizes) * legend.markerscale ** 2
348: (12)                       size_min = min(handle_sizes) * legend.markerscale ** 2
349: (12)                       numpoints = self.get_numpoints(legend)
350: (12)                       if numpoints < 4:
351: (16)                           sizes = [.5 * (size_max + size_min), size_max,
352: (25)                                    size_min][:numpoints]
353: (12)                       else:
354: (16)                           rng = (size_max - size_min)
355: (16)                           sizes = rng * np.linspace(0, 1, numpoints) + size_min
356: (8)                    else:
357: (12)                       sizes = self._sizes
358: (8)                    return sizes
359: (4)                def update_prop(self, legend_handle, orig_handle, legend):
360: (8)                    self._update_prop(legend_handle, orig_handle)
361: (8)                    legend_handle.set_figure(legend.figure)
362: (8)                    legend_handle.set_clip_box(None)
363: (8)                    legend_handle.set_clip_path(None)
364: (4)                def create_collection(self, orig_handle, sizes, offsets,
offset_transform):
365: (8)                    return type(orig_handle)(
366: (12)                       orig_handle.get_numsides(),
367: (12)                       rotation=orig_handle.get_rotation(), sizes=sizes,
368: (12)                       offsets=offsets, offset_transform=offset_transform,
369: (8)                    )
370: (4)                def create_artists(self, legend, orig_handle,
371: (23)                                  xdescent, ydescent, width, height, fontsize,
372: (23)                                  trans):
373: (8)                    xdata, xdata_marker = self.get_xdata(legend, xdescent, ydescent,
374: (45)                                                        width, height, fontsize)
375: (8)                    ydata = self.get_ydata(legend, xdescent, ydescent,
376: (31)                                          width, height, fontsize)
377: (8)                    sizes = self.get_sizes(legend, orig_handle, xdescent, ydescent,
378: (31)                                          width, height, fontsize)
379: (8)                    p = self.create_collection(
380: (12)                       orig_handle, sizes,
381: (12)                       offsets=list(zip(xdata_marker, ydata)), offset_transform=trans)
382: (8)                    self.update_prop(p, orig_handle, legend)
383: (8)                    p.set_offset_transform(trans)
384: (8)                    return [p]
385: (0)            class HandlerPathCollection(HandlerRegularPolyCollection):
386: (4)                r"""Handler for `.PathCollection`\s, which are used by
`~.Axes.scatter`."""
387: (4)                def create_collection(self, orig_handle, sizes, offsets,
offset_transform):
388: (8)                    return type(orig_handle)(
389: (12)                       [orig_handle.get_paths()[0]], sizes=sizes,
390: (12)                       offsets=offsets, offset_transform=offset_transform,
391: (8)                    )
392: (0)            class HandlerCircleCollection(HandlerRegularPolyCollection):
393: (4)                r"""Handler for `.CircleCollection`\s."""
394: (4)                def create_collection(self, orig_handle, sizes, offsets,
offset_transform):
```

```
(8)                     return type(orig_handle)(
(12)                        sizes, offsets=offsets, offset_transform=offset_transform)
(0)             class HandlerErrorbar(HandlerLine2D):
(4)                 """Handler for Errorbars."""
(4)                 def __init__(self, xerr_size=0.5, yerr_size=None,
(17)                         marker_pad=0.3, numpoints=None, **kwargs):
(8)                     self._xerr_size = xerr_size
(8)                     self._yerr_size = yerr_size
(8)                     super().__init__(marker_pad=marker_pad, numpoints=numpoints, **kwargs)
(4)                 def get_err_size(self, legend, xdescent, ydescent,
(21)                             width, height, fontsize):
(8)                     xerr_size = self._xerr_size * fontsize
(8)                     if self._yerr_size is None:
(12)                        yerr_size = xerr_size
(8)                     else:
(12)                        yerr_size = self._yerr_size * fontsize
(8)                     return xerr_size, yerr_size
(4)                 def create_artists(self, legend, orig_handle,
(23)                                xdescent, ydescent, width, height, fontsize,
(23)                                trans):
(8)                     plotlines, caplines, barlinecols = orig_handle
(8)                     xdata, xdata_marker = self.get_xdata(legend, xdescent, ydescent,
(45)                                                     width, height, fontsize)
(8)                     ydata = np.full_like(xdata, (height - ydescent) / 2)
(8)                     legline = Line2D(xdata, ydata)
(8)                     xdata_marker = np.asarray(xdata_marker)
(8)                     ydata_marker = np.asarray(ydata[:len(xdata_marker)])
(8)                     xerr_size, yerr_size = self.get_err_size(legend, xdescent, ydescent,
(49)                                                         width, height, fontsize)
(8)                     legline_marker = Line2D(xdata_marker, ydata_marker)
(8)                     if plotlines is None:
(12)                        legline.set_visible(False)
(12)                        legline_marker.set_visible(False)
(8)                     else:
(12)                        self.update_prop(legline, plotlines, legend)
(12)                        legline.set_drawstyle('default')
(12)                        legline.set_marker('none')
(12)                        self.update_prop(legline_marker, plotlines, legend)
(12)                        legline_marker.set_linestyle('None')
(12)                        if legend.markerscale != 1:
(16)                            newsz = legline_marker.get_markersize() * legend.markerscale
(16)                            legline_marker.set_markersize(newsz)
(8)                     handle_barlinecols = []
(8)                     handle_caplines = []
(8)                     if orig_handle.has_xerr:
(12)                        verts = [((x - xerr_size, y), (x + xerr_size, y))
(21)                                 for x, y in zip(xdata_marker, ydata_marker)]
(12)                        coll = mcoll.LineCollection(verts)
(12)                        self.update_prop(coll, barlinecols[0], legend)
(12)                        handle_barlinecols.append(coll)
(12)                        if caplines:
(16)                            capline_left = Line2D(xdata_marker - xerr_size, ydata_marker)
(16)                            capline_right = Line2D(xdata_marker + xerr_size, ydata_marker)
(16)                            self.update_prop(capline_left, caplines[0], legend)
(16)                            self.update_prop(capline_right, caplines[0], legend)
(16)                            capline_left.set_marker("|")
(16)                            capline_right.set_marker("|")
(16)                            handle_caplines.append(capline_left)
(16)                            handle_caplines.append(capline_right)
(8)                     if orig_handle.has_yerr:
(12)                        verts = [((x, y - yerr_size), (x, y + yerr_size))
(21)                                 for x, y in zip(xdata_marker, ydata_marker)]
(12)                        coll = mcoll.LineCollection(verts)
(12)                        self.update_prop(coll, barlinecols[0], legend)
(12)                        handle_barlinecols.append(coll)
(12)                        if caplines:
(16)                            capline_left = Line2D(xdata_marker, ydata_marker - yerr_size)
(16)                            capline_right = Line2D(xdata_marker, ydata_marker + yerr_size)
(16)                            self.update_prop(capline_left, caplines[0], legend)
```

```
464: (16)                            self.update_prop(capline_right, caplines[0], legend)
465: (16)                            capline_left.set_marker("_")
466: (16)                            capline_right.set_marker("_")
467: (16)                            handle_caplines.append(capline_left)
468: (16)                            handle_caplines.append(capline_right)
469: (8)                        artists = [
470: (12)                           *handle_barlinecols, *handle_caplines, legline, legline_marker,
471: (8)                        ]
472: (8)                        for artist in artists:
473: (12)                           artist.set_transform(trans)
474: (8)                        return artists
475: (0)                class HandlerStem(HandlerNpointsYoffsets):
476: (4)                    """
477: (4)                    Handler for plots produced by `~.Axes.stem`.
478: (4)                    """
479: (4)                    def __init__(self, marker_pad=0.3, numpoints=None,
480: (17)                                 bottom=None, yoffsets=None, **kwargs):
481: (8)                        """
482: (8)                        Parameters
483: (8)                        ----------
484: (8)                        marker_pad : float, default: 0.3
485: (12)                           Padding between points in legend entry.
486: (8)                        numpoints : int, optional
487: (12)                           Number of points to show in legend entry.
488: (8)                        bottom : float, optional
489: (8)                        yoffsets : array of floats, optional
490: (12)                           Length *numpoints* list of y offsets for each point in
491: (12)                           legend entry.
492: (8)                        **kwargs
493: (12)                           Keyword arguments forwarded to `.HandlerNpointsYoffsets`.
494: (8)                        """
495: (8)                        super().__init__(marker_pad=marker_pad, numpoints=numpoints,
496: (25)                                        yoffsets=yoffsets, **kwargs)
497: (8)                        self._bottom = bottom
498: (4)                    def get_ydata(self, legend, xdescent, ydescent, width, height, fontsize):
499: (8)                        if self._yoffsets is None:
500: (12)                           ydata = height * (0.5 * legend._scatteryoffsets + 0.5)
501: (8)                        else:
502: (12)                           ydata = height * np.asarray(self._yoffsets)
503: (8)                        return ydata
504: (4)                    def create_artists(self, legend, orig_handle,
505: (23)                                      xdescent, ydescent, width, height, fontsize,
506: (23)                                      trans):
507: (8)                        markerline, stemlines, baseline = orig_handle
508: (8)                        using_linecoll = isinstance(stemlines, mcoll.LineCollection)
509: (8)                        xdata, xdata_marker = self.get_xdata(legend, xdescent, ydescent,
510: (45)                                                            width, height, fontsize)
511: (8)                        ydata = self.get_ydata(legend, xdescent, ydescent,
512: (31)                                              width, height, fontsize)
513: (8)                        if self._bottom is None:
514: (12)                           bottom = 0.
515: (8)                        else:
516: (12)                           bottom = self._bottom
517: (8)                        leg_markerline = Line2D(xdata_marker, ydata[:len(xdata_marker)])
518: (8)                        self.update_prop(leg_markerline, markerline, legend)
519: (8)                        leg_stemlines = [Line2D([x, x], [bottom, y])
520: (25)                                        for x, y in zip(xdata_marker, ydata)]
521: (8)                        if using_linecoll:
522: (12)                           with cbook._setattr_cm(
523: (20)                                   self, _update_prop_func=self._copy_collection_props):
524: (16)                               for line in leg_stemlines:
525: (20)                                   self.update_prop(line, stemlines, legend)
526: (8)                        else:
527: (12)                           for lm, m in zip(leg_stemlines, stemlines):
528: (16)                               self.update_prop(lm, m, legend)
529: (8)                        leg_baseline = Line2D([np.min(xdata), np.max(xdata)],
530: (30)                                             [bottom, bottom])
531: (8)                        self.update_prop(leg_baseline, baseline, legend)
532: (8)                        artists = [*leg_stemlines, leg_baseline, leg_markerline]
```

```
(8)                     for artist in artists:
(12)                        artist.set_transform(trans)
(8)                     return artists
(4)                 def _copy_collection_props(self, legend_handle, orig_handle):
(8)                     """
(8)                     Copy properties from the `.LineCollection` *orig_handle* to the
(8)                     `.Line2D` *legend_handle*.
(8)                     """
(8)                     legend_handle.set_color(orig_handle.get_color()[0])
(8)                     legend_handle.set_linestyle(orig_handle.get_linestyle()[0])
(0)             class HandlerTuple(HandlerBase):
(4)                 """
(4)                 Handler for Tuple.
(4)                 """
(4)                 def __init__(self, ndivide=1, pad=None, **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     ndivide : int or None, default: 1
(12)                        The number of sections to divide the legend area into.  If None,
(12)                        use the length of the input tuple.
(8)                     pad : float, default: :rc:`legend.borderpad`
(12)                        Padding in units of fraction of font size.
(8)                     **kwargs
(12)                        Keyword arguments forwarded to `.HandlerBase`.
(8)                     """
(8)                     self._ndivide = ndivide
(8)                     self._pad = pad
(8)                     super().__init__(**kwargs)
(4)                 def create_artists(self, legend, orig_handle,
(23)                                   xdescent, ydescent, width, height, fontsize,
(23)                                   trans):
(8)                     handler_map = legend.get_legend_handler_map()
(8)                     if self._ndivide is None:
(12)                        ndivide = len(orig_handle)
(8)                     else:
(12)                        ndivide = self._ndivide
(8)                     if self._pad is None:
(12)                        pad = legend.borderpad * fontsize
(8)                     else:
(12)                        pad = self._pad * fontsize
(8)                     if ndivide > 1:
(12)                        width = (width - pad * (ndivide - 1)) / ndivide
(8)                     xds_cycle = cycle(xdescent - (width + pad) * np.arange(ndivide))
(8)                     a_list = []
(8)                     for handle1 in orig_handle:
(12)                        handler = legend.get_legend_handler(handler_map, handle1)
(12)                        _a_list = handler.create_artists(
(16)                            legend, handle1,
(16)                            next(xds_cycle), ydescent, width, height, fontsize, trans)
(12)                        a_list.extend(_a_list)
(8)                     return a_list
(0)             class HandlerPolyCollection(HandlerBase):
(4)                 """
(4)                 Handler for `.PolyCollection` used in `~.Axes.fill_between` and
(4)                 `~.Axes.stackplot`.
(4)                 """
(4)                 def _update_prop(self, legend_handle, orig_handle):
(8)                     def first_color(colors):
(12)                        if colors.size == 0:
(16)                            return (0, 0, 0, 0)
(12)                        return tuple(colors[0])
(8)                     def get_first(prop_array):
(12)                        if len(prop_array):
(16)                            return prop_array[0]
(12)                        else:
(16)                            return None
(8)                     legend_handle._facecolor = first_color(orig_handle.get_facecolor())
(8)                     legend_handle._edgecolor = first_color(orig_handle.get_edgecolor())
```

```
602: (8)                    legend_handle._original_facecolor = orig_handle._original_facecolor
603: (8)                    legend_handle._original_edgecolor = orig_handle._original_edgecolor
604: (8)                    legend_handle._fill = orig_handle.get_fill()
605: (8)                    legend_handle._hatch = orig_handle.get_hatch()
606: (8)                    legend_handle._hatch_color = orig_handle._hatch_color
607: (8)                    legend_handle.set_linewidth(get_first(orig_handle.get_linewidths()))
608: (8)                    legend_handle.set_linestyle(get_first(orig_handle.get_linestyles()))
609: (8)                    legend_handle.set_transform(get_first(orig_handle.get_transforms()))
610: (8)                    legend_handle.set_figure(orig_handle.get_figure())
611: (4)                def create_artists(self, legend, orig_handle,
612: (23)                                   xdescent, ydescent, width, height, fontsize, trans):
613: (8)                    p = Rectangle(xy=(-xdescent, -ydescent),
614: (22)                                 width=width, height=height)
615: (8)                    self.update_prop(p, orig_handle, legend)
616: (8)                    p.set_transform(trans)
617: (8)                    return [p]

----------------------------------------

File 45 - legend.py:

1: (0)              """
2: (0)              The legend module defines the Legend class, which is responsible for
3: (0)              drawing legends associated with Axes and/or figures.
4: (0)              .. important::
5: (4)                  It is unlikely that you would ever create a Legend instance manually.
6: (4)                  Most users would normally create a legend via the `~.Axes.legend`
7: (4)                  function. For more details on legends there is also a :ref:`legend guide
8: (4)                  <legend_guide>`.
9: (0)              The `Legend` class is a container of legend handles and legend texts.
10: (0)             The legend handler map specifies how to create legend handles from artists
11: (0)             (lines, patches, etc.) in the Axes or figures. Default legend handlers are
12: (0)             defined in the :mod:`~matplotlib.legend_handler` module. While not all artist
13: (0)             types are covered by the default legend handlers, custom legend handlers can
be
14: (0)             defined to support arbitrary objects.
15: (0)             See the :ref`<legend_guide>` for more
16: (0)             information.
17: (0)             """
18: (0)             import itertools
19: (0)             import logging
20: (0)             import numbers
21: (0)             import time
22: (0)             import numpy as np
23: (0)             import matplotlib as mpl
24: (0)             from matplotlib import _api, _docstring, cbook, colors, offsetbox
25: (0)             from matplotlib.artist import Artist, allow_rasterization
26: (0)             from matplotlib.cbook import silent_list
27: (0)             from matplotlib.font_manager import FontProperties
28: (0)             from matplotlib.lines import Line2D
29: (0)             from matplotlib.patches import (Patch, Rectangle, Shadow, FancyBboxPatch,
30: (32)                                            StepPatch)
31: (0)             from matplotlib.collections import (
32: (4)                 Collection, CircleCollection, LineCollection, PathCollection,
33: (4)                 PolyCollection, RegularPolyCollection)
34: (0)             from matplotlib.text import Text
35: (0)             from matplotlib.transforms import Bbox, BboxBase, TransformedBbox
36: (0)             from matplotlib.transforms import BboxTransformTo, BboxTransformFrom
37: (0)             from matplotlib.offsetbox import (
38: (4)                 AnchoredOffsetbox, DraggableOffsetBox,
39: (4)                 HPacker, VPacker,
40: (4)                 DrawingArea, TextArea,
41: (0)             )
42: (0)             from matplotlib.container import ErrorbarContainer, BarContainer,
StemContainer
43: (0)             from . import legend_handler
44: (0)             class DraggableLegend(DraggableOffsetBox):
45: (4)                 def __init__(self, legend, use_blit=False, update="loc"):
46: (8)                     """
```

```
47: (8)                     Wrapper around a `.Legend` to support mouse dragging.
48: (8)                     Parameters
49: (8)                     ----------
50: (8)                     legend : `.Legend`
51: (12)                        The `.Legend` instance to wrap.
52: (8)                     use_blit : bool, optional
53: (12)                        Use blitting for faster image composition. For details see
54: (12)                        :ref:`func-animation`.
55: (8)                     update : {'loc', 'bbox'}, optional
56: (12)                        If "loc", update the *loc* parameter of the legend upon
finalizing.
57: (12)                        If "bbox", update the *bbox_to_anchor* parameter.
58: (8)                     """
59: (8)                     self.legend = legend
60: (8)                     _api.check_in_list(["loc", "bbox"], update=update)
61: (8)                     self._update = update
62: (8)                     super().__init__(legend, legend._legend_box, use_blit=use_blit)
63: (4)                 def finalize_offset(self):
64: (8)                     if self._update == "loc":
65: (12)                        self._update_loc(self.get_loc_in_canvas())
66: (8)                     elif self._update == "bbox":
67: (12)                        self._update_bbox_to_anchor(self.get_loc_in_canvas())
68: (4)                 def _update_loc(self, loc_in_canvas):
69: (8)                     bbox = self.legend.get_bbox_to_anchor()
70: (8)                     if bbox.width == 0 or bbox.height == 0:
71: (12)                        self.legend.set_bbox_to_anchor(None)
72: (12)                        bbox = self.legend.get_bbox_to_anchor()
73: (8)                     _bbox_transform = BboxTransformFrom(bbox)
74: (8)                     self.legend._loc = tuple(_bbox_transform.transform(loc_in_canvas))
75: (4)                 def _update_bbox_to_anchor(self, loc_in_canvas):
76: (8)                     loc_in_bbox = self.legend.axes.transAxes.transform(loc_in_canvas)
77: (8)                     self.legend.set_bbox_to_anchor(loc_in_bbox)
78: (0)             _legend_kw_doc_base = """
79: (0)             bbox_to_anchor : `.BboxBase`, 2-tuple, or 4-tuple of floats
80: (4)                 Box that is used to position the legend in conjunction with *loc*.
81: (4)                 Defaults to `axes.bbox` (if called as a method to `.Axes.legend`) or
82: (4)                 `figure.bbox` (if `.Figure.legend`).  This argument allows arbitrary
83: (4)                 placement of the legend.
84: (4)                 Bbox coordinates are interpreted in the coordinate system given by
85: (4)                 *bbox_transform*, with the default transform
86: (4)                 Axes or Figure coordinates, depending on which ``legend`` is called.
87: (4)                 If a 4-tuple or `.BboxBase` is given, then it specifies the bbox
88: (4)                 ``(x, y, width, height)`` that the legend is placed in.
89: (4)                 To put the legend in the best location in the bottom right
90: (4)                 quadrant of the Axes (or figure)::
91: (8)                     loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
92: (4)                 A 2-tuple ``(x, y)`` places the corner of the legend specified by *loc* at
93: (4)                 x, y.  For example, to put the legend's upper right-hand corner in the
94: (4)                 center of the Axes (or figure) the following keywords can be used::
95: (8)                     loc='upper right', bbox_to_anchor=(0.5, 0.5)
96: (0)             ncols : int, default: 1
97: (4)                 The number of columns that the legend has.
98: (4)                 For backward compatibility, the spelling *ncol* is also supported
99: (4)                 but it is discouraged. If both are given, *ncols* takes precedence.
100: (0)            prop : None or `~matplotlib.font_manager.FontProperties` or dict
101: (4)                The font properties of the legend. If None (default), the current
102: (4)                :data:`matplotlib.rcParams` will be used.
103: (0)            fontsize : int or {'xx-small', 'x-small', 'small', 'medium', 'large', \
104: (0)            'x-large', 'xx-large'}
105: (4)                The font size of the legend. If the value is numeric the size will be the
106: (4)                absolute font size in points. String values are relative to the current
107: (4)                default font size. This argument is only used if *prop* is not specified.
108: (0)            labelcolor : str or list, default: :rc:`legend.labelcolor`
109: (4)                The color of the text in the legend. Either a valid color string
110: (4)                (for example, 'red'), or a list of color strings. The labelcolor can
111: (4)                also be made to match the color of the line or marker using 'linecolor',
112: (4)                'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').
113: (4)                Labelcolor can be set globally using :rc:`legend.labelcolor`. If None,
114: (4)                use :rc:`text.color`.
```

```
(0)             numpoints : int, default: :rc:`legend.numpoints`
(4)                 The number of marker points in the legend when creating a legend
(4)                 entry for a `.Line2D` (line).
(0)             scatterpoints : int, default: :rc:`legend.scatterpoints`
(4)                 The number of marker points in the legend when creating
(4)                 a legend entry for a `.PathCollection` (scatter plot).
(0)             scatteryoffsets : iterable of floats, default: ``[0.375, 0.5, 0.3125]``
(4)                 The vertical offset (relative to the font size) for the markers
(4)                 created for a scatter plot legend entry. 0.0 is at the base the
(4)                 legend text, and 1.0 is at the top. To draw all markers at the
(4)                 same height, set to ``[0.5]``.
(0)             markerscale : float, default: :rc:`legend.markerscale`
(4)                 The relative size of legend markers compared to the originally drawn ones.
(0)             markerfirst : bool, default: True
(4)                 If *True*, legend marker is placed to the left of the legend label.
(4)                 If *False*, legend marker is placed to the right of the legend label.
(0)             reverse : bool, default: False
(4)                 If *True*, the legend labels are displayed in reverse order from the
input.
(4)                 If *False*, the legend labels are displayed in the same order as the
input.
(4)                 .. versionadded:: 3.7
(0)             frameon : bool, default: :rc:`legend.frameon`
(4)                 Whether the legend should be drawn on a patch (frame).
(0)             fancybox : bool, default: :rc:`legend.fancybox`
(4)                 Whether round edges should be enabled around the `.FancyBboxPatch` which
(4)                 makes up the legend's background.
(0)             shadow : None, bool or dict, default: :rc:`legend.shadow`
(4)                 Whether to draw a shadow behind the legend.
(4)                 The shadow can be configured using `.Patch` keywords.
(4)                 Customization via :rc:`legend.shadow` is currently not supported.
(0)             framealpha : float, default: :rc:`legend.framealpha`
(4)                 The alpha transparency of the legend's background.
(4)                 If *shadow* is activated and *framealpha* is ``None``, the default value
is
(4)                 ignored.
(0)             facecolor : "inherit" or color, default: :rc:`legend.facecolor`
(4)                 The legend's background color.
(4)                 If ``"inherit"``, use :rc:`axes.facecolor`.
(0)             edgecolor : "inherit" or color, default: :rc:`legend.edgecolor`
(4)                 The legend's background patch edge color.
(4)                 If ``"inherit"``, use :rc:`axes.edgecolor`.
(0)             mode : {"expand", None}
(4)                 If *mode* is set to ``"expand"`` the legend will be horizontally
(4)                 expanded to fill the Axes area (or *bbox_to_anchor* if defines
(4)                 the legend's size).
(0)             bbox_transform : None or `~matplotlib.transforms.Transform`
(4)                 The transform for the bounding box (*bbox_to_anchor*). For a value
(4)                 of ``None`` (default) the Axes'
(4)                 :data:`~matplotlib.axes.Axes.transAxes` transform will be used.
(0)             title : str or None
(4)                 The legend's title. Default is no title (``None``).
(0)             title_fontproperties : None or `~matplotlib.font_manager.FontProperties` or
dict
(4)                 The font properties of the legend's title. If None (default), the
(4)                 *title_fontsize* argument will be used if present; if *title_fontsize* is
(4)                 also None, the current :rc:`legend.title_fontsize` will be used.
(0)             title_fontsize : int or {'xx-small', 'x-small', 'small', 'medium', 'large', \
(0)             'x-large', 'xx-large'}, default: :rc:`legend.title_fontsize`
(4)                 The font size of the legend's title.
(4)                 Note: This cannot be combined with *title_fontproperties*. If you want
(4)                 to set the fontsize alongside other font properties, use the *size*
(4)                 parameter in *title_fontproperties*.
(0)             alignment : {'center', 'left', 'right'}, default: 'center'
(4)                 The alignment of the legend title and the box of entries. The entries
(4)                 are aligned as a single block, so that markers always lined up.
(0)             borderpad : float, default: :rc:`legend.borderpad`
(4)                 The fractional whitespace inside the legend border, in font-size units.
(0)             labelspacing : float, default: :rc:`legend.labelspacing`
```

```
(4)                 The vertical space between the legend entries, in font-size units.
(0)             handlelength : float, default: :rc:`legend.handlelength`
(4)                 The length of the legend handles, in font-size units.
(0)             handleheight : float, default: :rc:`legend.handleheight`
(4)                 The height of the legend handles, in font-size units.
(0)             handletextpad : float, default: :rc:`legend.handletextpad`
(4)                 The pad between the legend handle and text, in font-size units.
(0)             borderaxespad : float, default: :rc:`legend.borderaxespad`
(4)                 The pad between the Axes and legend border, in font-size units.
(0)             columnspacing : float, default: :rc:`legend.columnspacing`
(4)                 The spacing between columns, in font-size units.
(0)             handler_map : dict or None
(4)                 The custom dictionary mapping instances or types to a legend
(4)                 handler. This *handler_map* updates the default handler map
(4)                 found at `matplotlib.legend.Legend.get_legend_handler_map`.
(0)             draggable : bool, default: False
(4)                 Whether the legend can be dragged with the mouse.
(0)             """
(0)             _loc_doc_base = """
(0)             loc : str or pair of floats, default: {default}
(4)                 The location of the legend.
(4)                 The strings ``'upper left'``, ``'upper right'``, ``'lower left'``,
(4)                 ``'lower right'`` place the legend at the corresponding corner of the
(4)                 {parent}.
(4)                 The strings ``'upper center'``, ``'lower center'``, ``'center left'``,
(4)                 ``'center right'`` place the legend at the center of the corresponding
edge
(4)                 of the {parent}.
(4)                 The string ``'center'`` places the legend at the center of the {parent}.
(0)             {best}
(4)                 The location can also be a 2-tuple giving the coordinates of the lower-
left
(4)                 corner of the legend in {parent} coordinates (in which case
*bbox_to_anchor*
(4)                 will be ignored).
(4)                 For back-compatibility, ``'center right'`` (but no other location) can
also
(4)                 be spelled ``'right'``, and each "string" location can also be given as a
(4)                 numeric value:
(4)                 ==================   =============
(4)                 Location String      Location Code
(4)                 ==================   =============
(4)                 'best' (Axes only)   0
(4)                 'upper right'        1
(4)                 'upper left'         2
(4)                 'lower left'         3
(4)                 'lower right'        4
(4)                 'right'              5
(4)                 'center left'        6
(4)                 'center right'       7
(4)                 'lower center'       8
(4)                 'upper center'       9
(4)                 'center'             10
(4)                 ==================   =============
(4)                 {outside}"""
(0)             _loc_doc_best = """
(4)                 The string ``'best'`` places the legend at the location, among the nine
(4)                 locations defined so far, with the minimum overlap with other drawn
(4)                 artists.  This option can be quite slow for plots with large amounts of
(4)                 data; your plotting speed may benefit from providing a specific location.
(0)             """
(0)             _legend_kw_axes_st = (
(4)                 _loc_doc_base.format(parent='axes', default=':rc:`legend.loc`',
(25)                                     best=_loc_doc_best, outside='') +
(4)                 _legend_kw_doc_base)
(0)             _docstring.interpd.update(_legend_kw_axes=_legend_kw_axes_st)
(0)             _outside_doc = """
(4)                 If a figure is using the constrained layout manager, the string codes
(4)                 of the *loc* keyword argument can get better layout behaviour using the
```

```
245: (4)                prefix 'outside'. There is ambiguity at the corners, so 'outside
246: (4)                upper right' will make space for the legend above the rest of the
247: (4)                axes in the layout, and 'outside right upper' will make space on the
248: (4)                right side of the layout.  In addition to the values of *loc*
249: (4)                listed above, we have 'outside right upper', 'outside right lower',
250: (4)                'outside left upper', and 'outside left lower'.  See
251: (4)                :ref:`legend_guide` for more details.
252: (0)            """
253: (0)            _legend_kw_figure_st = (
254: (4)                _loc_doc_base.format(parent='figure', default="'upper right'",
255: (25)                                     best='', outside=_outside_doc) +
256: (4)                _legend_kw_doc_base)
257: (0)            _docstring.interpd.update(_legend_kw_figure=_legend_kw_figure_st)
258: (0)            _legend_kw_both_st = (
259: (4)                _loc_doc_base.format(parent='axes/figure',
260: (25)                                     default=":rc:`legend.loc` for Axes, 'upper right' for
Figure",
261: (25)                                     best=_loc_doc_best, outside=_outside_doc) +
262: (4)                _legend_kw_doc_base)
263: (0)            _docstring.interpd.update(_legend_kw_doc=_legend_kw_both_st)
264: (0)            _legend_kw_set_loc_st = (
265: (4)                _loc_doc_base.format(parent='axes/figure',
266: (25)                                     default=":rc:`legend.loc` for Axes, 'upper right' for
Figure",
267: (25)                                     best=_loc_doc_best, outside=_outside_doc))
268: (0)            _docstring.interpd.update(_legend_kw_set_loc_doc=_legend_kw_set_loc_st)
269: (0)            class Legend(Artist):
270: (4)                """
271: (4)                Place a legend on the figure/axes.
272: (4)                """
273: (4)                codes = {'best': 0, **AnchoredOffsetbox.codes}
274: (4)                zorder = 5
275: (4)                def __str__(self):
276: (8)                    return "Legend"
277: (4)                @_docstring.dedent_interpd
278: (4)                def __init__(
279: (8)                    self, parent, handles, labels,
280: (8)                    *,
281: (8)                    loc=None,
282: (8)                    numpoints=None,      # number of points in the legend line
283: (8)                    markerscale=None,    # relative size of legend markers vs. original
284: (8)                    markerfirst=True,    # left/right ordering of legend marker and label
285: (8)                    reverse=False,       # reverse ordering of legend marker and label
286: (8)                    scatterpoints=None,  # number of scatter points
287: (8)                    scatteryoffsets=None,
288: (8)                    prop=None,           # properties for the legend texts
289: (8)                    fontsize=None,       # keyword to set font size directly
290: (8)                    labelcolor=None,     # keyword to set the text color
291: (8)                    borderpad=None,      # whitespace inside the legend border
292: (8)                    labelspacing=None,   # vertical space between the legend entries
293: (8)                    handlelength=None,   # length of the legend handles
294: (8)                    handleheight=None,   # height of the legend handles
295: (8)                    handletextpad=None,  # pad between the legend handle and text
296: (8)                    borderaxespad=None,  # pad between the Axes and legend border
297: (8)                    columnspacing=None,  # spacing between columns
298: (8)                    ncols=1,     # number of columns
299: (8)                    mode=None,  # horizontal distribution of columns: None or "expand"
300: (8)                    fancybox=None,  # True: fancy box, False: rounded box, None: rcParam
301: (8)                    shadow=None,
302: (8)                    title=None,           # legend title
303: (8)                    title_fontsize=None,  # legend title font size
304: (8)                    framealpha=None,      # set frame alpha
305: (8)                    edgecolor=None,       # frame patch edgecolor
306: (8)                    facecolor=None,       # frame patch facecolor
307: (8)                    bbox_to_anchor=None,  # bbox to which the legend will be anchored
308: (8)                    bbox_transform=None,  # transform for the bbox
309: (8)                    frameon=None,         # draw frame
310: (8)                    handler_map=None,
311: (8)                    title_fontproperties=None,  # properties for the legend title
```

```
312: (8)                    alignment="center",       # control the alignment within the legend
box
313: (8)                    ncol=1,  # synonym for ncols (backward compatibility)
314: (8)                    draggable=False  # whether the legend can be dragged with the mouse
315: (4)                ):
316: (8)                    """
317: (8)                    Parameters
318: (8)                    ----------
319: (8)                    parent : `~matplotlib.axes.Axes` or `.Figure`
320: (12)                       The artist that contains the legend.
321: (8)                    handles : list of (`.Artist` or tuple of `.Artist`)
322: (12)                       A list of Artists (lines, patches) to be added to the legend.
323: (8)                    labels : list of str
324: (12)                       A list of labels to show next to the artists. The length of
handles
325: (12)                       and labels should be the same. If they are not, they are truncated
326: (12)                       to the length of the shorter list.
327: (8)                    Other Parameters
328: (8)                    ----------------
329: (8)                    %(_legend_kw_doc)s
330: (8)                    Attributes
331: (8)                    ----------
332: (8)                    legend_handles
333: (12)                       List of `.Artist` objects added as legend entries.
334: (12)                       .. versionadded:: 3.7
335: (8)                    """
336: (8)                    from matplotlib.axes import Axes
337: (8)                    from matplotlib.figure import FigureBase
338: (8)                    super().__init__()
339: (8)                    if prop is None:
340: (12)                       self.prop = FontProperties(size=mpl._val_or_rc(fontsize,
"legend.fontsize"))
341: (8)                    else:
342: (12)                       self.prop = FontProperties._from_any(prop)
343: (12)                       if isinstance(prop, dict) and "size" not in prop:
344: (16)                           self.prop.set_size(mpl.rcParams["legend.fontsize"])
345: (8)                    self._fontsize = self.prop.get_size_in_points()
346: (8)                    self.texts = []
347: (8)                    self.legend_handles = []
348: (8)                    self._legend_title_box = None
349: (8)                    self._custom_handler_map = handler_map
350: (8)                    self.numpoints = mpl._val_or_rc(numpoints, 'legend.numpoints')
351: (8)                    self.markerscale = mpl._val_or_rc(markerscale, 'legend.markerscale')
352: (8)                    self.scatterpoints = mpl._val_or_rc(scatterpoints,
'legend.scatterpoints')
353: (8)                    self.borderpad = mpl._val_or_rc(borderpad, 'legend.borderpad')
354: (8)                    self.labelspacing = mpl._val_or_rc(labelspacing,
'legend.labelspacing')
355: (8)                    self.handlelength = mpl._val_or_rc(handlelength,
'legend.handlelength')
356: (8)                    self.handleheight = mpl._val_or_rc(handleheight,
'legend.handleheight')
357: (8)                    self.handletextpad = mpl._val_or_rc(handletextpad,
'legend.handletextpad')
358: (8)                    self.borderaxespad = mpl._val_or_rc(borderaxespad,
'legend.borderaxespad')
359: (8)                    self.columnspacing = mpl._val_or_rc(columnspacing,
'legend.columnspacing')
360: (8)                    self.shadow = mpl._val_or_rc(shadow, 'legend.shadow')
361: (8)                    _lab, _hand = [], []
362: (8)                    for label, handle in zip(labels, handles):
363: (12)                       if isinstance(label, str) and label.startswith('_'):
364: (16)                           _api.warn_deprecated("3.8", message=(
365: (20)                               "An artist whose label starts with an underscore was
passed to "
366: (20)                               "legend(); such artists will no longer be ignored in the
future.  "
367: (20)                               "To suppress this warning, explicitly filter out such
artists, "
```

```
368: (20)                                "e.g. with `[art for art in artists if not "
369: (20)                                "art.get_label().startswith('_')]`."))
370: (12)                        else:
371: (16)                            _lab.append(label)
372: (16)                            _hand.append(handle)
373: (8)                    labels, handles = _lab, _hand
374: (8)                    if reverse:
375: (12)                        labels.reverse()
376: (12)                        handles.reverse()
377: (8)                    if len(handles) < 2:
378: (12)                        ncols = 1
379: (8)                    self._ncols = ncols if ncols != 1 else ncol
380: (8)                    if self.numpoints <= 0:
381: (12)                        raise ValueError("numpoints must be > 0; it was %d" % numpoints)
382: (8)                    if scatteryoffsets is None:
383: (12)                        self._scatteryoffsets = np.array([3. / 8., 4. / 8., 2.5 / 8.])
384: (8)                    else:
385: (12)                        self._scatteryoffsets = np.asarray(scatteryoffsets)
386: (8)                    reps = self.scatterpoints // len(self._scatteryoffsets) + 1
387: (8)                    self._scatteryoffsets = np.tile(self._scatteryoffsets,
388: (40)                                                   reps)[:self.scatterpoints]
389: (8)                    self._legend_box = None
390: (8)                    if isinstance(parent, Axes):
391: (12)                        self.isaxes = True
392: (12)                        self.axes = parent
393: (12)                        self.set_figure(parent.figure)
394: (8)                    elif isinstance(parent, FigureBase):
395: (12)                        self.isaxes = False
396: (12)                        self.set_figure(parent)
397: (8)                    else:
398: (12)                        raise TypeError(
399: (16)                            "Legend needs either Axes or FigureBase as parent"
400: (12)                        )
401: (8)                    self.parent = parent
402: (8)                    self._mode = mode
403: (8)                    self.set_bbox_to_anchor(bbox_to_anchor, bbox_transform)
404: (8)                    self._shadow_props = {'ox': 2, 'oy': -2}  # default location offsets
405: (8)                    if isinstance(self.shadow, dict):
406: (12)                       self._shadow_props.update(self.shadow)
407: (12)                       self.shadow = True
408: (8)                    elif self.shadow in (0, 1, True, False):
409: (12)                       self.shadow = bool(self.shadow)
410: (8)                    else:
411: (12)                       raise ValueError(
412: (16)                           'Legend shadow must be a dict or bool, not '
413: (16)                           f'{self.shadow!r} of type {type(self.shadow)}.'
414: (12)                       )
415: (8)                    facecolor = mpl._val_or_rc(facecolor, "legend.facecolor")
416: (8)                    if facecolor == 'inherit':
417: (12)                       facecolor = mpl.rcParams["axes.facecolor"]
418: (8)                    edgecolor = mpl._val_or_rc(edgecolor, "legend.edgecolor")
419: (8)                    if edgecolor == 'inherit':
420: (12)                       edgecolor = mpl.rcParams["axes.edgecolor"]
421: (8)                    fancybox = mpl._val_or_rc(fancybox, "legend.fancybox")
422: (8)                    self.legendPatch = FancyBboxPatch(
423: (12)                       xy=(0, 0), width=1, height=1,
424: (12)                       facecolor=facecolor, edgecolor=edgecolor,
425: (12)                       alpha=(framealpha if framealpha is not None
426: (19)                              else 1 if shadow
427: (19)                              else mpl.rcParams["legend.framealpha"]),
428: (12)                       boxstyle=("round,pad=0,rounding_size=0.2" if fancybox
429: (22)                                 else "square,pad=0"),
430: (12)                       mutation_scale=self._fontsize,
431: (12)                       snap=True,
432: (12)                       visible=mpl._val_or_rc(frameon, "legend.frameon")
433: (8)                    )
434: (8)                    self._set_artist_props(self.legendPatch)
435: (8)                    _api.check_in_list(["center", "left", "right"], alignment=alignment)
436: (8)                    self._alignment = alignment
```

```
437: (8)                    self._init_legend_box(handles, labels, markerfirst)
438: (8)                    self.set_loc(loc)
439: (8)                    if title_fontsize is not None and title_fontproperties is not None:
440: (12)                       raise ValueError(
441: (16)                           "title_fontsize and title_fontproperties can't be specified "
442: (16)                           "at the same time. Only use one of them. ")
443: (8)                    title_prop_fp = FontProperties._from_any(title_fontproperties)
444: (8)                    if isinstance(title_fontproperties, dict):
445: (12)                       if "size" not in title_fontproperties:
446: (16)                           title_fontsize = mpl.rcParams["legend.title_fontsize"]
447: (16)                           title_prop_fp.set_size(title_fontsize)
448: (8)                    elif title_fontsize is not None:
449: (12)                       title_prop_fp.set_size(title_fontsize)
450: (8)                    elif not isinstance(title_fontproperties, FontProperties):
451: (12)                       title_fontsize = mpl.rcParams["legend.title_fontsize"]
452: (12)                       title_prop_fp.set_size(title_fontsize)
453: (8)                    self.set_title(title, prop=title_prop_fp)
454: (8)                    self._draggable = None
455: (8)                    self.set_draggable(state=draggable)
456: (8)                    color_getters = {  # getter function depends on line or patch
457: (12)                       'linecolor':       ['get_color',           'get_facecolor'],
458: (12)                       'markerfacecolor': ['get_markerfacecolor', 'get_facecolor'],
459: (12)                       'mfc':             ['get_markerfacecolor', 'get_facecolor'],
460: (12)                       'markeredgecolor': ['get_markeredgecolor', 'get_edgecolor'],
461: (12)                       'mec':             ['get_markeredgecolor', 'get_edgecolor'],
462: (8)                    }
463: (8)                    labelcolor = mpl._val_or_rc(labelcolor, 'legend.labelcolor')
464: (8)                    if labelcolor is None:
465: (12)                       labelcolor = mpl.rcParams['text.color']
466: (8)                    if isinstance(labelcolor, str) and labelcolor in color_getters:
467: (12)                       getter_names = color_getters[labelcolor]
468: (12)                       for handle, text in zip(self.legend_handles, self.texts):
469: (16)                           try:
470: (20)                               if handle.get_array() is not None:
471: (24)                                   continue
472: (16)                           except AttributeError:
473: (20)                               pass
474: (16)                           for getter_name in getter_names:
475: (20)                               try:
476: (24)                                   color = getattr(handle, getter_name)()
477: (24)                                   if isinstance(color, np.ndarray):
478: (28)                                       if (
479: (36)                                               color.shape[0] == 1
480: (36)                                               or np.isclose(color, color[0]).all()
481: (28)                                       ):
482: (32)                                           text.set_color(color[0])
483: (28)                                       else:
484: (32)                                           pass
485: (24)                                   else:
486: (28)                                       text.set_color(color)
487: (24)                                   break
488: (20)                               except AttributeError:
489: (24)                                   pass
490: (8)                    elif cbook._str_equal(labelcolor, 'none'):
491: (12)                       for text in self.texts:
492: (16)                           text.set_color(labelcolor)
493: (8)                    elif np.iterable(labelcolor):
494: (12)                       for text, color in zip(self.texts,
495: (35)                                              itertools.cycle(
496: (39)                                                  colors.to_rgba_array(labelcolor))):
497: (16)                           text.set_color(color)
498: (8)                    else:
499: (12)                       raise ValueError(f"Invalid labelcolor: {labelcolor!r}")
500: (4)                def _set_artist_props(self, a):
501: (8)                    """
502: (8)                    Set the boilerplate props for artists added to Axes.
503: (8)                    """
504: (8)                    a.set_figure(self.figure)
505: (8)                    if self.isaxes:
```

```
(12)                        a.axes = self.axes
(8)                     a.set_transform(self.get_transform())
(4)                 @_docstring.dedent_interpd
(4)                 def set_loc(self, loc=None):
(8)                     """
(8)                     Set the location of the legend.
(8)                     .. versionadded:: 3.8
(8)                     Parameters
(8)                     ----------
(8)                     %(_legend_kw_set_loc_doc)s
(8)                     """
(8)                     loc0 = loc
(8)                     self._loc_used_default = loc is None
(8)                     if loc is None:
(12)                        loc = mpl.rcParams["legend.loc"]
(12)                        if not self.isaxes and loc in [0, 'best']:
(16)                            loc = 'upper right'
(8)                     type_err_message = ("loc must be string, coordinate tuple, or"
(28)                                        f" an integer 0-10, not {loc!r}")
(8)                     self._outside_loc = None
(8)                     if isinstance(loc, str):
(12)                        if loc.split()[0] == 'outside':
(16)                            loc = loc.split('outside ')[1]
(16)                            self._outside_loc = loc.replace('center ', '')
(16)                            self._outside_loc = self._outside_loc.split()[0]
(16)                            locs = loc.split()
(16)                            if len(locs) > 1 and locs[0] in ('right', 'left'):
(20)                                if locs[0] != 'center':
(24)                                    locs = locs[::-1]
(20)                                loc = locs[0] + ' ' + locs[1]
(12)                        loc = _api.check_getitem(self.codes, loc=loc)
(8)                     elif np.iterable(loc):
(12)                        loc = tuple(loc)
(12)                        if len(loc) != 2 or not all(isinstance(e, numbers.Real) for e in
loc):
(16)                            raise ValueError(type_err_message)
(8)                     elif isinstance(loc, int):
(12)                        if loc < 0 or loc > 10:
(16)                            raise ValueError(type_err_message)
(8)                     else:
(12)                        raise ValueError(type_err_message)
(8)                     if self.isaxes and self._outside_loc:
(12)                        raise ValueError(
(16)                            f"'outside' option for loc='{loc0}' keyword argument only "
(16)                            "works for figure legends")
(8)                     if not self.isaxes and loc == 0:
(12)                        raise ValueError(
(16)                            "Automatic legend placement (loc='best') not implemented for "
(16)                            "figure legend")
(8)                     tmp = self._loc_used_default
(8)                     self._set_loc(loc)
(8)                     self._loc_used_default = tmp  # ignore changes done by _set_loc
(4)                 def _set_loc(self, loc):
(8)                     self._loc_used_default = False
(8)                     self._loc_real = loc
(8)                     self.stale = True
(8)                     self._legend_box.set_offset(self._findoffset)
(4)                 def set_ncols(self, ncols):
(8)                     """Set the number of columns."""
(8)                     self._ncols = ncols
(4)                 def _get_loc(self):
(8)                     return self._loc_real
(4)                 _loc = property(_get_loc, _set_loc)
(4)                 def _findoffset(self, width, height, xdescent, ydescent, renderer):
(8)                     """Helper function to locate the legend."""
(8)                     if self._loc == 0:  # "best".
(12)                        x, y = self._find_best_position(width, height, renderer)
(8)                     elif self._loc in Legend.codes.values():  # Fixed location.
(12)                        bbox = Bbox.from_bounds(0, 0, width, height)
```

```
(12)                        x, y = self._get_anchored_bbox(self._loc, bbox,
(43)                                                       self.get_bbox_to_anchor(),
(43)                                                       renderer)
(8)                     else:  # Axes or figure coordinates.
(12)                        fx, fy = self._loc
(12)                        bbox = self.get_bbox_to_anchor()
(12)                        x, y = bbox.x0 + bbox.width * fx, bbox.y0 + bbox.height * fy
(8)                     return x + xdescent, y + ydescent
(4)                 @allow_rasterization
(4)                 def draw(self, renderer):
(8)                     if not self.get_visible():
(12)                        return
(8)                     renderer.open_group('legend', gid=self.get_gid())
(8)                     fontsize = renderer.points_to_pixels(self._fontsize)
(8)                     if self._mode in ["expand"]:
(12)                        pad = 2 * (self.borderaxespad + self.borderpad) * fontsize
(12)                        self._legend_box.set_width(self.get_bbox_to_anchor().width - pad)
(8)                     bbox = self._legend_box.get_window_extent(renderer)
(8)                     self.legendPatch.set_bounds(bbox.bounds)
(8)                     self.legendPatch.set_mutation_scale(fontsize)
(8)                     if self.shadow:
(12)                        Shadow(self.legendPatch, **self._shadow_props).draw(renderer)
(8)                     self.legendPatch.draw(renderer)
(8)                     self._legend_box.draw(renderer)
(8)                     renderer.close_group('legend')
(8)                     self.stale = False
(4)                 _default_handler_map = {
(8)                     StemContainer: legend_handler.HandlerStem(),
(8)                     ErrorbarContainer: legend_handler.HandlerErrorbar(),
(8)                     Line2D: legend_handler.HandlerLine2D(),
(8)                     Patch: legend_handler.HandlerPatch(),
(8)                     StepPatch: legend_handler.HandlerStepPatch(),
(8)                     LineCollection: legend_handler.HandlerLineCollection(),
(8)                     RegularPolyCollection: legend_handler.HandlerRegularPolyCollection(),
(8)                     CircleCollection: legend_handler.HandlerCircleCollection(),
(8)                     BarContainer: legend_handler.HandlerPatch(
(12)                        update_func=legend_handler.update_from_first_child),
(8)                     tuple: legend_handler.HandlerTuple(),
(8)                     PathCollection: legend_handler.HandlerPathCollection(),
(8)                     PolyCollection: legend_handler.HandlerPolyCollection()
(8)                     }
(4)                 @classmethod
(4)                 def get_default_handler_map(cls):
(8)                     """Return the global default handler map, shared by all legends."""
(8)                     return cls._default_handler_map
(4)                 @classmethod
(4)                 def set_default_handler_map(cls, handler_map):
(8)                     """Set the global default handler map, shared by all legends."""
(8)                     cls._default_handler_map = handler_map
(4)                 @classmethod
(4)                 def update_default_handler_map(cls, handler_map):
(8)                     """Update the global default handler map, shared by all legends."""
(8)                     cls._default_handler_map.update(handler_map)
(4)                 def get_legend_handler_map(self):
(8)                     """Return this legend instance's handler map."""
(8)                     default_handler_map = self.get_default_handler_map()
(8)                     return ({**default_handler_map, **self._custom_handler_map}
(16)                            if self._custom_handler_map else default_handler_map)
(4)                 @staticmethod
(4)                 def get_legend_handler(legend_handler_map, orig_handle):
(8)                     """
(8)                     Return a legend handler from *legend_handler_map* that
(8)                     corresponds to *orig_handler*.
(8)                     *legend_handler_map* should be a dictionary object (that is
(8)                     returned by the get_legend_handler_map method).
(8)                     It first checks if the *orig_handle* itself is a key in the
(8)                     *legend_handler_map* and return the associated value.
(8)                     Otherwise, it checks for each of the classes in its
(8)                     method-resolution-order. If no matching key is found, it
```

```
(8)                 returns ``None``.
(8)                 """
(8)                 try:
(12)                    return legend_handler_map[orig_handle]
(8)                 except (TypeError, KeyError):  # TypeError if unhashable.
(12)                    pass
(8)                 for handle_type in type(orig_handle).mro():
(12)                    try:
(16)                        return legend_handler_map[handle_type]
(12)                    except KeyError:
(16)                        pass
(8)                 return None
(4)             def _init_legend_box(self, handles, labels, markerfirst=True):
(8)                 """
(8)                 Initialize the legend_box. The legend_box is an instance of
(8)                 the OffsetBox, which is packed with legend handles and
(8)                 texts. Once packed, their location is calculated during the
(8)                 drawing time.
(8)                 """
(8)                 fontsize = self._fontsize
(8)                 text_list = []  # the list of text instances
(8)                 handle_list = []  # the list of handle instances
(8)                 handles_and_labels = []
(8)                 descent = 0.35 * fontsize * (self.handleheight - 0.7)  # heuristic.
(8)                 height = fontsize * self.handleheight - descent
(8)                 legend_handler_map = self.get_legend_handler_map()
(8)                 for orig_handle, label in zip(handles, labels):
(12)                    handler = self.get_legend_handler(legend_handler_map, orig_handle)
(12)                    if handler is None:
(16)                        _api.warn_external(
(29)                                    "Legend does not support handles for "
(29)                                    f"{type(orig_handle).__name__} "
(29)                                    "instances.\nA proxy artist may be used "
(29)                                    "instead.\nSee: https://matplotlib.org/"
(29)                                    "stable/users/explain/axes/legend_guide.html"
(29)                                    "#controlling-the-legend-entries")
(16)                        handle_list.append(None)
(12)                    else:
(16)                        textbox = TextArea(label, multilinebaseline=True,
(35)                                           textprops=dict(
(39)                                               verticalalignment='baseline',
(39)                                               horizontalalignment='left',
(39)                                               fontproperties=self.prop))
(16)                        handlebox = DrawingArea(width=self.handlelength * fontsize,
(40)                                                height=height,
(40)                                                xdescent=0., ydescent=descent)
(16)                        text_list.append(textbox._text)
(16)                        handle_list.append(handler.legend_artist(self, orig_handle,
(57)                                                                 fontsize, handlebox))
(16)                        handles_and_labels.append((handlebox, textbox))
(8)                 columnbox = []
(8)                 for handles_and_labels_column in filter(
(16)                        len, np.array_split(handles_and_labels, self._ncols)):
(12)                    itemboxes = [HPacker(pad=0,
(33)                                         sep=self.handletextpad * fontsize,
(33)                                         children=[h, t] if markerfirst else [t, h],
(33)                                         align="baseline")
(25)                                 for h, t in handles_and_labels_column]
(12)                    alignment = "baseline" if markerfirst else "right"
(12)                    columnbox.append(VPacker(pad=0,
(37)                                             sep=self.labelspacing * fontsize,
(37)                                             align=alignment,
(37)                                             children=itemboxes))
(8)                 mode = "expand" if self._mode == "expand" else "fixed"
(8)                 sep = self.columnspacing * fontsize
(8)                 self._legend_handle_box = HPacker(pad=0,
(42)                                                  sep=sep, align="baseline",
(42)                                                  mode=mode,
(42)                                                  children=columnbox)
```

```
712: (8)                    self._legend_title_box = TextArea("")
713: (8)                    self._legend_box = VPacker(pad=self.borderpad * fontsize,
714: (35)                                               sep=self.labelspacing * fontsize,
715: (35)                                               align=self._alignment,
716: (35)                                               children=[self._legend_title_box,
717: (45)                                                         self._legend_handle_box])
718: (8)                    self._legend_box.set_figure(self.figure)
719: (8)                    self._legend_box.axes = self.axes
720: (8)                    self.texts = text_list
721: (8)                    self.legend_handles = handle_list
722: (4)                def _auto_legend_data(self):
723: (8)                    """
724: (8)                    Return display coordinates for hit testing for "best" positioning.
725: (8)                    Returns
726: (8)                    -------
727: (8)                    bboxes
728: (12)                       List of bounding boxes of all patches.
729: (8)                    lines
730: (12)                       List of `.Path` corresponding to each line.
731: (8)                    offsets
732: (12)                       List of (x, y) offsets of all collection.
733: (8)                    """
734: (8)                    assert self.isaxes  # always holds, as this is only called internally
735: (8)                    bboxes = []
736: (8)                    lines = []
737: (8)                    offsets = []
738: (8)                    for artist in self.parent._children:
739: (12)                       if isinstance(artist, Line2D):
740: (16)                           lines.append(
741: (20)                               artist.get_transform().transform_path(artist.get_path()))
742: (12)                       elif isinstance(artist, Rectangle):
743: (16)                           bboxes.append(
744: (20)
artist.get_bbox().transformed(artist.get_data_transform()))
745: (12)                       elif isinstance(artist, Patch):
746: (16)                           lines.append(
747: (20)                               artist.get_transform().transform_path(artist.get_path()))
748: (12)                       elif isinstance(artist, PolyCollection):
749: (16)                           lines.extend(artist.get_transform().transform_path(path)
750: (29)                                        for path in artist.get_paths())
751: (12)                       elif isinstance(artist, Collection):
752: (16)                           transform, transOffset, hoffsets, _ = artist._prepare_points()
753: (16)                           if len(hoffsets):
754: (20)                               offsets.extend(transOffset.transform(hoffsets))
755: (12)                       elif isinstance(artist, Text):
756: (16)                           bboxes.append(artist.get_window_extent())
757: (8)                    return bboxes, lines, offsets
758: (4)                def get_children(self):
759: (8)                    return [self._legend_box, self.get_frame()]
760: (4)                def get_frame(self):
761: (8)                    """Return the `~.patches.Rectangle` used to frame the legend."""
762: (8)                    return self.legendPatch
763: (4)                def get_lines(self):
764: (8)                    r"""Return the list of `~.lines.Line2D`\s in the legend."""
765: (8)                    return [h for h in self.legend_handles if isinstance(h, Line2D)]
766: (4)                def get_patches(self):
767: (8)                    r"""Return the list of `~.patches.Patch`\s in the legend."""
768: (8)                    return silent_list('Patch',
769: (27)                                      [h for h in self.legend_handles
770: (28)                                       if isinstance(h, Patch)])
771: (4)                def get_texts(self):
772: (8)                    r"""Return the list of `~.text.Text`\s in the legend."""
773: (8)                    return silent_list('Text', self.texts)
774: (4)                def set_alignment(self, alignment):
775: (8)                    """
776: (8)                    Set the alignment of the legend title and the box of entries.
777: (8)                    The entries are aligned as a single block, so that markers always
778: (8)                    lined up.
779: (8)                    Parameters
```

```
(8)                     ----------
(8)                     alignment : {'center', 'left', 'right'}.
(8)                     """
(8)                     _api.check_in_list(["center", "left", "right"], alignment=alignment)
(8)                     self._alignment = alignment
(8)                     self._legend_box.align = alignment
(4)                 def get_alignment(self):
(8)                     """Get the alignment value of the legend box"""
(8)                     return self._legend_box.align
(4)                 def set_title(self, title, prop=None):
(8)                     """
(8)                     Set legend title and title style.
(8)                     Parameters
(8)                     ----------
(8)                     title : str
(12)                        The legend title.
(8)                     prop : `.font_manager.FontProperties` or `str` or `pathlib.Path`
(12)                        The font properties of the legend title.
(12)                        If a `str`, it is interpreted as a fontconfig pattern parsed by
(12)                        `.FontProperties`.  If a `pathlib.Path`, it is interpreted as the
(12)                        absolute path to a font file.
(8)                     """
(8)                     self._legend_title_box._text.set_text(title)
(8)                     if title:
(12)                        self._legend_title_box._text.set_visible(True)
(12)                        self._legend_title_box.set_visible(True)
(8)                     else:
(12)                        self._legend_title_box._text.set_visible(False)
(12)                        self._legend_title_box.set_visible(False)
(8)                     if prop is not None:
(12)                        self._legend_title_box._text.set_fontproperties(prop)
(8)                     self.stale = True
(4)                 def get_title(self):
(8)                     """Return the `.Text` instance for the legend title."""
(8)                     return self._legend_title_box._text
(4)                 def get_window_extent(self, renderer=None):
(8)                     if renderer is None:
(12)                        renderer = self.figure._get_renderer()
(8)                     return self._legend_box.get_window_extent(renderer=renderer)
(4)                 def get_tightbbox(self, renderer=None):
(8)                     return self._legend_box.get_window_extent(renderer)
(4)                 def get_frame_on(self):
(8)                     """Get whether the legend box patch is drawn."""
(8)                     return self.legendPatch.get_visible()
(4)                 def set_frame_on(self, b):
(8)                     """
(8)                     Set whether the legend box patch is drawn.
(8)                     Parameters
(8)                     ----------
(8)                     b : bool
(8)                     """
(8)                     self.legendPatch.set_visible(b)
(8)                     self.stale = True
(4)                 draw_frame = set_frame_on  # Backcompat alias.
(4)                 def get_bbox_to_anchor(self):
(8)                     """Return the bbox that the legend will be anchored to."""
(8)                     if self._bbox_to_anchor is None:
(12)                        return self.parent.bbox
(8)                     else:
(12)                        return self._bbox_to_anchor
(4)                 def set_bbox_to_anchor(self, bbox, transform=None):
(8)                     """
(8)                     Set the bbox that the legend will be anchored to.
(8)                     Parameters
(8)                     ----------
(8)                     bbox : `~matplotlib.transforms.BboxBase` or tuple
(12)                        The bounding box can be specified in the following ways:
(12)                        - A `.BboxBase` instance
(12)                        - A tuple of ``(left, bottom, width, height)`` in the given
```

```
849: (14)                         transform (normalized axes coordinate if None)
850: (12)                       - A tuple of ``(left, bottom)`` where the width and height will be
851: (14)                         assumed to be zero.
852: (12)                       - *None*, to remove the bbox anchoring, and use the parent bbox.
853: (8)                    transform : `~matplotlib.transforms.Transform`, optional
854: (12)                       A transform to apply to the bounding box. If not specified, this
855: (12)                       will use a transform to the bounding box of the parent.
856: (8)                    """
857: (8)                    if bbox is None:
858: (12)                       self._bbox_to_anchor = None
859: (12)                       return
860: (8)                    elif isinstance(bbox, BboxBase):
861: (12)                       self._bbox_to_anchor = bbox
862: (8)                    else:
863: (12)                       try:
864: (16)                           l = len(bbox)
865: (12)                       except TypeError as err:
866: (16)                           raise ValueError(f"Invalid bbox: {bbox}") from err
867: (12)                       if l == 2:
868: (16)                           bbox = [bbox[0], bbox[1], 0, 0]
869: (12)                       self._bbox_to_anchor = Bbox.from_bounds(*bbox)
870: (8)                    if transform is None:
871: (12)                       transform = BboxTransformTo(self.parent.bbox)
872: (8)                    self._bbox_to_anchor = TransformedBbox(self._bbox_to_anchor,
873: (47)                                                          transform)
874: (8)                    self.stale = True
875: (4)                def _get_anchored_bbox(self, loc, bbox, parentbbox, renderer):
876: (8)                    """
877: (8)                    Place the *bbox* inside the *parentbbox* according to a given
878: (8)                    location code. Return the (x, y) coordinate of the bbox.
879: (8)                    Parameters
880: (8)                    ----------
881: (8)                    loc : int
882: (12)                       A location code in range(1, 11). This corresponds to the possible
883: (12)                       values for ``self._loc``, excluding "best".
884: (8)                    bbox : `~matplotlib.transforms.Bbox`
885: (12)                       bbox to be placed, in display coordinates.
886: (8)                    parentbbox : `~matplotlib.transforms.Bbox`
887: (12)                       A parent box which will contain the bbox, in display coordinates.
888: (8)                    """
889: (8)                    return offsetbox._get_anchored_bbox(
890: (12)                       loc, bbox, parentbbox,
891: (12)                       self.borderaxespad * renderer.points_to_pixels(self._fontsize))
892: (4)                def _find_best_position(self, width, height, renderer):
893: (8)                    """Determine the best location to place the legend."""
894: (8)                    assert self.isaxes  # always holds, as this is only called internally
895: (8)                    start_time = time.perf_counter()
896: (8)                    bboxes, lines, offsets = self._auto_legend_data()
897: (8)                    bbox = Bbox.from_bounds(0, 0, width, height)
898: (8)                    candidates = []
899: (8)                    for idx in range(1, len(self.codes)):
900: (12)                       l, b = self._get_anchored_bbox(idx, bbox,
901: (43)                                                      self.get_bbox_to_anchor(),
902: (43)                                                      renderer)
903: (12)                       legendBox = Bbox.from_bounds(l, b, width, height)
904: (12)                       badness = (sum(legendBox.count_contains(line.vertices)
905: (27)                                      for line in lines)
906: (23)                                  + legendBox.count_contains(offsets)
907: (23)                                  + legendBox.count_overlaps(bboxes)
908: (23)                                  + sum(line.intersects_bbox(legendBox, filled=False)
909: (29)                                        for line in lines))
910: (12)                       candidates.append((badness, idx, (l, b)))
911: (12)                       if badness == 0:
912: (16)                           break
913: (8)                    _, _, (l, b) = min(candidates)
914: (8)                    if self._loc_used_default and time.perf_counter() - start_time > 1:
915: (12)                       _api.warn_external(
916: (16)                           'Creating legend with loc="best" can be slow with large '
917: (16)                           'amounts of data.')
```

```
(8)                 return l, b
(4)             @_api.rename_parameter("3.8", "event", "mouseevent")
(4)             def contains(self, mouseevent):
(8)                 return self.legendPatch.contains(mouseevent)
(4)             def set_draggable(self, state, use_blit=False, update='loc'):
(8)                 """
(8)                 Enable or disable mouse dragging support of the legend.
(8)                 Parameters
(8)                 ----------
(8)                 state : bool
(12)                    Whether mouse dragging is enabled.
(8)                 use_blit : bool, optional
(12)                    Use blitting for faster image composition. For details see
(12)                    :ref:`func-animation`.
(8)                 update : {'loc', 'bbox'}, optional
(12)                    The legend parameter to be changed when dragged:
(12)                    - 'loc': update the *loc* parameter of the legend
(12)                    - 'bbox': update the *bbox_to_anchor* parameter of the legend
(8)                 Returns
(8)                 -------
(8)                 `.DraggableLegend` or *None*
(12)                    If *state* is ``True`` this returns the `.DraggableLegend` helper
(12)                    instance. Otherwise this returns *None*.
(8)                 """
(8)                 if state:
(12)                    if self._draggable is None:
(16)                        self._draggable = DraggableLegend(self,
(50)                                                          use_blit,
(50)                                                          update=update)
(8)                 else:
(12)                    if self._draggable is not None:
(16)                        self._draggable.disconnect()
(12)                    self._draggable = None
(8)                 return self._draggable
(4)             def get_draggable(self):
(8)                 """Return ``True`` if the legend is draggable, ``False`` otherwise."""
(8)                 return self._draggable is not None
(0)         def _get_legend_handles(axs, legend_handler_map=None):
(4)             """Yield artists that can be used as handles in a legend."""
(4)             handles_original = []
(4)             for ax in axs:
(8)                 handles_original += [
(12)                    *(a for a in ax._children
(14)                      if isinstance(a, (Line2D, Patch, Collection, Text))),
(12)                    *ax.containers]
(8)                 if hasattr(ax, 'parasites'):
(12)                    for axx in ax.parasites:
(16)                        handles_original += [
(20)                            *(a for a in axx._children
(22)                              if isinstance(a, (Line2D, Patch, Collection, Text))),
(20)                            *axx.containers]
(4)             handler_map = {**Legend.get_default_handler_map(),
(19)                           **(legend_handler_map or {})}
(4)             has_handler = Legend.get_legend_handler
(4)             for handle in handles_original:
(8)                 label = handle.get_label()
(8)                 if label != '_nolegend_' and has_handler(handler_map, handle):
(12)                    yield handle
(8)                 elif (label and not label.startswith('_') and
(16)                        not has_handler(handler_map, handle)):
(12)                    _api.warn_external(
(29)                                       "Legend does not support handles for "
(29)                                       f"{type(handle).__name__} "
(29)                                       "instances.\nSee: https://matplotlib.org/stable/"
(29)                                       "tutorials/intermediate/legend_guide.html"
(29)                                       "#implementing-a-custom-legend-handler")
(12)                    continue
(0)         def _get_legend_handles_labels(axs, legend_handler_map=None):
(4)             """Return handles and labels for legend."""
```

```
    handles = []
    labels = []
    for handle in _get_legend_handles(axs, legend_handler_map):
        label = handle.get_label()
        if label and not label.startswith('_'):
            handles.append(handle)
            labels.append(label)
    return handles, labels
def _parse_legend_args(axs, *args, handles=None, labels=None, **kwargs):
    """
    Get the handles and labels from the calls to either ``figure.legend``
    or ``axes.legend``.
    The parser is a bit involved because we support::
        legend()
        legend(labels)
        legend(handles, labels)
        legend(labels=labels)
        legend(handles=handles)
        legend(handles=handles, labels=labels)
    The behavior for a mixture of positional and keyword handles and labels
    is undefined and issues a warning; it will be an error in the future.
    Parameters
    ----------
    axs : list of `.Axes`
        If handles are not given explicitly, the artists in these Axes are
        used as handles.
    *args : tuple
        Positional parameters passed to ``legend()``.
    handles
        The value of the keyword argument ``legend(handles=...)``, or *None*
        if that keyword argument was not used.
    labels
        The value of the keyword argument ``legend(labels=...)``, or *None*
        if that keyword argument was not used.
    **kwargs
        All other keyword arguments passed to ``legend()``.
    Returns
    -------
    handles : list of (`.Artist` or tuple of `.Artist`)
        The legend handles.
    labels : list of str
        The legend labels.
    kwargs : dict
        *kwargs* with keywords handles and labels removed.
    """
    log = logging.getLogger(__name__)
    handlers = kwargs.get('handler_map')
    if (handles is not None or labels is not None) and args:
        _api.warn_deprecated("3.9", message=(
            "You have mixed positional and keyword arguments, some input may "
            "be discarded.  This is deprecated since %(since)s and will "
            "become an error %(removal)s."))
    if (hasattr(handles, "__len__") and
            hasattr(labels, "__len__") and
            len(handles) != len(labels)):
        _api.warn_external(f"Mismatched number of handles and labels: "
                           f"len(handles) = {len(handles)} "
                           f"len(labels) = {len(labels)}")
    if handles and labels:
        handles, labels = zip(*zip(handles, labels))
    elif handles is not None and labels is None:
        labels = [handle.get_label() for handle in handles]
    elif labels is not None and handles is None:
        handles = [handle for handle, label
                   in zip(_get_legend_handles(axs, handlers), labels)]
    elif len(args) == 0:  # 0 args: automatically detect labels and handles.
        handles, labels = _get_legend_handles_labels(axs, handlers)
        if not handles:
            _api.warn_external(
```

```
1056: (16)                          "No artists with labels found to put in legend.  Note that "
1057: (16)                          "artists whose label start with an underscore are ignored "
1058: (16)                          "when legend() is called with no argument.")
1059: (4)               elif len(args) == 1:  # 1 arg: user defined labels, automatic handle
detection.
1060: (8)                   labels, = args
1061: (8)                   if any(isinstance(l, Artist) for l in labels):
1062: (12)                      raise TypeError("A single argument passed to legend() must be a "
1063: (28)                                      "list of labels, but found an Artist in there.")
1064: (8)                   handles = [handle for handle, label
1065: (19)                              in zip(_get_legend_handles(axs, handlers), labels)]
1066: (4)               elif len(args) == 2:  # 2 args: user defined handles and labels.
1067: (8)                   handles, labels = args[:2]
1068: (4)               else:
1069: (8)                   raise _api.nargs_error('legend', '0-2', len(args))
1070: (4)               return handles, labels, kwargs
```

File 46 - lines.py:

```
1: (0)              """
2: (0)              2D lines with support for a variety of line styles, markers, colors, etc.
3: (0)              """
4: (0)              import copy
5: (0)              from numbers import Integral, Number, Real
6: (0)              import logging
7: (0)              import numpy as np
8: (0)              import matplotlib as mpl
9: (0)              from . import _api, cbook, colors as mcolors, _docstring
10: (0)             from .artist import Artist, allow_rasterization
11: (0)             from .cbook import (
12: (4)                 _to_unmasked_float_array, ls_mapper, ls_mapper_r, STEP_LOOKUP_MAP)
13: (0)             from .markers import MarkerStyle
14: (0)             from .path import Path
15: (0)             from .transforms import Bbox, BboxTransformTo, TransformedPath
16: (0)             from ._enums import JoinStyle, CapStyle
17: (0)             from . import _path
18: (0)             from .markers import (  # noqa
19: (4)                 CARETLEFT, CARETRIGHT, CARETUP, CARETDOWN,
20: (4)                 CARETLEFTBASE, CARETRIGHTBASE, CARETUPBASE, CARETDOWNBASE,
21: (4)                 TICKLEFT, TICKRIGHT, TICKUP, TICKDOWN)
22: (0)             _log = logging.getLogger(__name__)
23: (0)             def _get_dash_pattern(style):
24: (4)                 """Convert linestyle to dash pattern."""
25: (4)                 if isinstance(style, str):
26: (8)                     style = ls_mapper.get(style, style)
27: (4)                 if style in ['solid', 'None']:
28: (8)                     offset = 0
29: (8)                     dashes = None
30: (4)                 elif style in ['dashed', 'dashdot', 'dotted']:
31: (8)                     offset = 0
32: (8)                     dashes = tuple(mpl.rcParams[f'lines.{style}_pattern'])
33: (4)                 elif isinstance(style, tuple):
34: (8)                     offset, dashes = style
35: (8)                     if offset is None:
36: (12)                        raise ValueError(f'Unrecognized linestyle: {style!r}')
37: (4)                 else:
38: (8)                     raise ValueError(f'Unrecognized linestyle: {style!r}')
39: (4)                 if dashes is not None:
40: (8)                     dsum = sum(dashes)
41: (8)                     if dsum:
42: (12)                        offset %= dsum
43: (4)                 return offset, dashes
44: (0)             def _get_inverse_dash_pattern(offset, dashes):
45: (4)                 """Return the inverse of the given dash pattern, for filling the gaps."""
46: (4)                 gaps = dashes[-1:] + dashes[:-1]
47: (4)                 offset_gaps = offset + dashes[-1]
48: (4)                 return offset_gaps, gaps
```

```
(0)             def _scale_dashes(offset, dashes, lw):
(4)                 if not mpl.rcParams['lines.scale_dashes']:
(8)                     return offset, dashes
(4)                 scaled_offset = offset * lw
(4)                 scaled_dashes = ([x * lw if x is not None else None for x in dashes]
(21)                                 if dashes is not None else None)
(4)                 return scaled_offset, scaled_dashes
(0)             def segment_hits(cx, cy, x, y, radius):
(4)                 """
(4)                 Return the indices of the segments in the polyline with coordinates (*cx*,
(4)                 *cy*) that are within a distance *radius* of the point (*x*, *y*).
(4)                 """
(4)                 if len(x) <= 1:
(8)                     res, = np.nonzero((cx - x) ** 2 + (cy - y) ** 2 <= radius ** 2)
(8)                     return res
(4)                 xr, yr = x[:-1], y[:-1]
(4)                 dx, dy = x[1:] - xr, y[1:] - yr
(4)                 Lnorm_sq = dx ** 2 + dy ** 2  # Possibly want to eliminate Lnorm==0
(4)                 u = ((cx - xr) * dx + (cy - yr) * dy) / Lnorm_sq
(4)                 candidates = (u >= 0) & (u <= 1)
(4)                 point_hits = (cx - x) ** 2 + (cy - y) ** 2 <= radius ** 2
(4)                 candidates = candidates & ~(point_hits[:-1] | point_hits[1:])
(4)                 px, py = xr + u * dx, yr + u * dy
(4)                 line_hits = (cx - px) ** 2 + (cy - py) ** 2 <= radius ** 2
(4)                 line_hits = line_hits & candidates
(4)                 points, = point_hits.ravel().nonzero()
(4)                 lines, = line_hits.ravel().nonzero()
(4)                 return np.concatenate((points, lines))
(0)             def _mark_every_path(markevery, tpath, affine, ax):
(4)                 """
(4)                 Helper function that sorts out how to deal the input
(4)                 `markevery` and returns the points where markers should be drawn.
(4)                 Takes in the `markevery` value and the line path and returns the
(4)                 sub-sampled path.
(4)                 """
(4)                 codes, verts = tpath.codes, tpath.vertices
(4)                 def _slice_or_none(in_v, slc):
(8)                     """Helper function to cope with `codes` being an ndarray or `None`."""
(8)                     if in_v is None:
(12)                        return None
(8)                     return in_v[slc]
(4)                 if isinstance(markevery, Integral):
(8)                     markevery = (0, markevery)
(4)                 elif isinstance(markevery, Real):
(8)                     markevery = (0.0, markevery)
(4)                 if isinstance(markevery, tuple):
(8)                     if len(markevery) != 2:
(12)                        raise ValueError('`markevery` is a tuple but its len is not 2; '
(29)                                         f'markevery={markevery}')
(8)                     start, step = markevery
(8)                     if isinstance(step, Integral):
(12)                        if not isinstance(start, Integral):
(16)                            raise ValueError(
(20)                                '`markevery` is a tuple with len 2 and second element is '
(20)                                'an int, but the first element is not an int; '
(20)                                f'markevery={markevery}')
(12)                        return Path(verts[slice(start, None, step)],
(24)                                    _slice_or_none(codes, slice(start, None, step)))
(8)                     elif isinstance(step, Real):
(12)                        if not isinstance(start, Real):
(16)                            raise ValueError(
(20)                                '`markevery` is a tuple with len 2 and second element is '
(20)                                'a float, but the first element is not a float or an int;
'
(20)                                f'markevery={markevery}')
(12)                        if ax is None:
(16)                            raise ValueError(
(20)                                "markevery is specified relative to the Axes size, but "
(20)                                "the line does not have a Axes as parent")
```

```
(12)                        fin = np.isfinite(verts).all(axis=1)
(12)                        fverts = verts[fin]
(12)                        disp_coords = affine.transform(fverts)
(12)                        delta = np.empty((len(disp_coords), 2))
(12)                        delta[0, :] = 0
(12)                        delta[1:, :] = disp_coords[1:, :] - disp_coords[:-1, :]
(12)                        delta = np.hypot(*delta.T).cumsum()
(12)                        (x0, y0), (x1, y1) = ax.transAxes.transform([[0, 0], [1, 1]])
(12)                        scale = np.hypot(x1 - x0, y1 - y0)
(12)                        marker_delta = np.arange(start * scale, delta[-1], step * scale)
(12)                        inds = np.abs(delta[np.newaxis, :] - marker_delta[:, np.newaxis])
(12)                        inds = inds.argmin(axis=1)
(12)                        inds = np.unique(inds)
(12)                        return Path(fverts[inds], _slice_or_none(codes, inds))
(8)                     else:
(12)                        raise ValueError(
(16)                            f"markevery={markevery!r} is a tuple with len 2, but its "
(16)                            f"second element is not an int or a float")
(4)                 elif isinstance(markevery, slice):
(8)                     return Path(verts[markevery], _slice_or_none(codes, markevery))
(4)                 elif np.iterable(markevery):
(8)                     try:
(12)                        return Path(verts[markevery], _slice_or_none(codes, markevery))
(8)                     except (ValueError, IndexError) as err:
(12)                        raise ValueError(
(16)                            f"markevery={markevery!r} is iterable but not a valid numpy "
(16)                            f"fancy index") from err
(4)                 else:
(8)                     raise ValueError(f"markevery={markevery!r} is not a recognized value")
(0)             @_docstring.interpd
(0)             @_api.define_aliases({
(4)                 "antialiased": ["aa"],
(4)                 "color": ["c"],
(4)                 "drawstyle": ["ds"],
(4)                 "linestyle": ["ls"],
(4)                 "linewidth": ["lw"],
(4)                 "markeredgecolor": ["mec"],
(4)                 "markeredgewidth": ["mew"],
(4)                 "markerfacecolor": ["mfc"],
(4)                 "markerfacecoloralt": ["mfcalt"],
(4)                 "markersize": ["ms"],
(0)             })
(0)             class Line2D(Artist):
(4)                 """
(4)                 A line - the line can have both a solid linestyle connecting all
(4)                 the vertices, and a marker at each vertex.  Additionally, the
(4)                 drawing of the solid line is influenced by the drawstyle, e.g., one
(4)                 can create "stepped" lines in various styles.
(4)                 """
(4)                 lineStyles = _lineStyles = {  # hidden names deprecated
(8)                     '-':    '_draw_solid',
(8)                     '--':   '_draw_dashed',
(8)                     '-.':   '_draw_dash_dot',
(8)                     ':':    '_draw_dotted',
(8)                     'None': '_draw_nothing',
(8)                     ' ':    '_draw_nothing',
(8)                     '':     '_draw_nothing',
(4)                 }
(4)                 _drawStyles_l = {
(8)                     'default':    '_draw_lines',
(8)                     'steps-mid':  '_draw_steps_mid',
(8)                     'steps-pre':  '_draw_steps_pre',
(8)                     'steps-post': '_draw_steps_post',
(4)                 }
(4)                 _drawStyles_s = {
(8)                     'steps': '_draw_steps_pre',
(4)                 }
(4)                 drawStyles = {**_drawStyles_l, **_drawStyles_s}
(4)                 drawStyleKeys = [*_drawStyles_l, *_drawStyles_s]
```

```
(4)     markers = MarkerStyle.markers
(4)     filled_markers = MarkerStyle.filled_markers
(4)     fillStyles = MarkerStyle.fillstyles
(4)     zorder = 2
(4)     _subslice_optim_min_size = 1000
(4)     def __str__(self):
(8)         if self._label != "":
(12)            return f"Line2D({self._label})"
(8)         elif self._x is None:
(12)            return "Line2D()"
(8)         elif len(self._x) > 3:
(12)            return "Line2D(({:g},{:g}),({:g},{:g}),...,({:g},{:g}))".format(
(16)                self._x[0], self._y[0],
(16)                self._x[1], self._y[1],
(16)                self._x[-1], self._y[-1])
(8)         else:
(12)            return "Line2D(%s)" % ",".join(
(16)                map("({:g},{:g})".format, self._x, self._y))
(4)     def __init__(self, xdata, ydata, *,
(17)                 linewidth=None,  # all Nones default to rc
(17)                 linestyle=None,
(17)                 color=None,
(17)                 gapcolor=None,
(17)                 marker=None,
(17)                 markersize=None,
(17)                 markeredgewidth=None,
(17)                 markeredgecolor=None,
(17)                 markerfacecolor=None,
(17)                 markerfacecoloralt='none',
(17)                 fillstyle=None,
(17)                 antialiased=None,
(17)                 dash_capstyle=None,
(17)                 solid_capstyle=None,
(17)                 dash_joinstyle=None,
(17)                 solid_joinstyle=None,
(17)                 pickradius=5,
(17)                 drawstyle=None,
(17)                 markevery=None,
(17)                 **kwargs
(17)                 ):
(8)         """
(8)         Create a `.Line2D` instance with *x* and *y* data in sequences of
(8)         *xdata*, *ydata*.
(8)         Additional keyword arguments are `.Line2D` properties:
(8)         %(Line2D:kwdoc)s
(8)         See :meth:`set_linestyle` for a description of the line styles,
(8)         :meth:`set_marker` for a description of the markers, and
(8)         :meth:`set_drawstyle` for a description of the draw styles.
(8)         """
(8)         super().__init__()
(8)         if not np.iterable(xdata):
(12)            raise RuntimeError('xdata must be a sequence')
(8)         if not np.iterable(ydata):
(12)            raise RuntimeError('ydata must be a sequence')
(8)         if linewidth is None:
(12)            linewidth = mpl.rcParams['lines.linewidth']
(8)         if linestyle is None:
(12)            linestyle = mpl.rcParams['lines.linestyle']
(8)         if marker is None:
(12)            marker = mpl.rcParams['lines.marker']
(8)         if color is None:
(12)            color = mpl.rcParams['lines.color']
(8)         if markersize is None:
(12)            markersize = mpl.rcParams['lines.markersize']
(8)         if antialiased is None:
(12)            antialiased = mpl.rcParams['lines.antialiased']
(8)         if dash_capstyle is None:
(12)            dash_capstyle = mpl.rcParams['lines.dash_capstyle']
(8)         if dash_joinstyle is None:
```

```
(12)                        dash_joinstyle = mpl.rcParams['lines.dash_joinstyle']
(8)                     if solid_capstyle is None:
(12)                        solid_capstyle = mpl.rcParams['lines.solid_capstyle']
(8)                     if solid_joinstyle is None:
(12)                        solid_joinstyle = mpl.rcParams['lines.solid_joinstyle']
(8)                     if drawstyle is None:
(12)                        drawstyle = 'default'
(8)                     self._dashcapstyle = None
(8)                     self._dashjoinstyle = None
(8)                     self._solidjoinstyle = None
(8)                     self._solidcapstyle = None
(8)                     self.set_dash_capstyle(dash_capstyle)
(8)                     self.set_dash_joinstyle(dash_joinstyle)
(8)                     self.set_solid_capstyle(solid_capstyle)
(8)                     self.set_solid_joinstyle(solid_joinstyle)
(8)                     self._linestyles = None
(8)                     self._drawstyle = None
(8)                     self._linewidth = linewidth
(8)                     self._unscaled_dash_pattern = (0, None)  # offset, dash
(8)                     self._dash_pattern = (0, None)  # offset, dash (scaled by linewidth)
(8)                     self.set_linewidth(linewidth)
(8)                     self.set_linestyle(linestyle)
(8)                     self.set_drawstyle(drawstyle)
(8)                     self._color = None
(8)                     self.set_color(color)
(8)                     if marker is None:
(12)                        marker = 'none'  # Default.
(8)                     if not isinstance(marker, MarkerStyle):
(12)                        self._marker = MarkerStyle(marker, fillstyle)
(8)                     else:
(12)                        self._marker = marker
(8)                     self._gapcolor = None
(8)                     self.set_gapcolor(gapcolor)
(8)                     self._markevery = None
(8)                     self._markersize = None
(8)                     self._antialiased = None
(8)                     self.set_markevery(markevery)
(8)                     self.set_antialiased(antialiased)
(8)                     self.set_markersize(markersize)
(8)                     self._markeredgecolor = None
(8)                     self._markeredgewidth = None
(8)                     self._markerfacecolor = None
(8)                     self._markerfacecoloralt = None
(8)                     self.set_markerfacecolor(markerfacecolor)  # Normalizes None to rc.
(8)                     self.set_markerfacecoloralt(markerfacecoloralt)
(8)                     self.set_markeredgecolor(markeredgecolor)  # Normalizes None to rc.
(8)                     self.set_markeredgewidth(markeredgewidth)
(8)                     self._internal_update(kwargs)
(8)                     self.pickradius = pickradius
(8)                     self.ind_offset = 0
(8)                     if (isinstance(self._picker, Number) and
(16)                            not isinstance(self._picker, bool)):
(12)                        self._pickradius = self._picker
(8)                     self._xorig = np.asarray([])
(8)                     self._yorig = np.asarray([])
(8)                     self._invalidx = True
(8)                     self._invalidy = True
(8)                     self._x = None
(8)                     self._y = None
(8)                     self._xy = None
(8)                     self._path = None
(8)                     self._transformed_path = None
(8)                     self._subslice = False
(8)                     self._x_filled = None  # used in subslicing; only x is needed
(8)                     self.set_data(xdata, ydata)
(4)                 def contains(self, mouseevent):
(8)                     """
(8)                     Test whether *mouseevent* occurred on the line.
(8)                     An event is deemed to have occurred "on" the line if it is less
```

```
(8)                 than ``self.pickradius`` (default: 5 points) away from it.  Use
(8)                 `~.Line2D.get_pickradius` or `~.Line2D.set_pickradius` to get or set
(8)                 the pick radius.
(8)                 Parameters
(8)                 ----------
(8)                 mouseevent : `~matplotlib.backend_bases.MouseEvent`
(8)                 Returns
(8)                 -------
(8)                 contains : bool
(12)                    Whether any values are within the radius.
(8)                 details : dict
(12)                    A dictionary ``{'ind': pointlist}``, where *pointlist* is a
(12)                    list of points of the line that are within the pickradius around
(12)                    the event position.
(12)                    TODO: sort returned indices by distance
(8)                 """
(8)                 if self._different_canvas(mouseevent):
(12)                    return False, {}
(8)                 if self._invalidy or self._invalidx:
(12)                    self.recache()
(8)                 if len(self._xy) == 0:
(12)                    return False, {}
(8)                 transformed_path = self._get_transformed_path()
(8)                 path, affine = transformed_path.get_transformed_path_and_affine()
(8)                 path = affine.transform_path(path)
(8)                 xy = path.vertices
(8)                 xt = xy[:, 0]
(8)                 yt = xy[:, 1]
(8)                 if self.figure is None:
(12)                    _log.warning('no figure set when check if mouse is on line')
(12)                    pixels = self._pickradius
(8)                 else:
(12)                    pixels = self.figure.dpi / 72. * self._pickradius
(8)                 with np.errstate(all='ignore'):
(12)                    if self._linestyle in ['None', None]:
(16)                        ind, = np.nonzero(
(20)                            (xt - mouseevent.x) ** 2 + (yt - mouseevent.y) ** 2
(20)                            <= pixels ** 2)
(12)                    else:
(16)                        ind = segment_hits(mouseevent.x, mouseevent.y, xt, yt, pixels)
(16)                        if self._drawstyle.startswith("steps"):
(20)                            ind //= 2
(8)                 ind += self.ind_offset
(8)                 return len(ind) > 0, dict(ind=ind)
(4)             def get_pickradius(self):
(8)                 """
(8)                 Return the pick radius used for containment tests.
(8)                 See `.contains` for more details.
(8)                 """
(8)                 return self._pickradius
(4)             def set_pickradius(self, pickradius):
(8)                 """
(8)                 Set the pick radius used for containment tests.
(8)                 See `.contains` for more details.
(8)                 Parameters
(8)                 ----------
(8)                 pickradius : float
(12)                    Pick radius, in points.
(8)                 """
(8)                 if not isinstance(pickradius, Real) or pickradius < 0:
(12)                    raise ValueError("pick radius should be a distance")
(8)                 self._pickradius = pickradius
(4)             pickradius = property(get_pickradius, set_pickradius)
(4)             def get_fillstyle(self):
(8)                 """
(8)                 Return the marker fill style.
(8)                 See also `~.Line2D.set_fillstyle`.
(8)                 """
(8)                 return self._marker.get_fillstyle()
```

```
    def set_fillstyle(self, fs):
        """
        Set the marker fill style.
        Parameters
        ----------
        fs : {'full', 'left', 'right', 'bottom', 'top', 'none'}
            Possible values:
            - 'full': Fill the whole marker with the *markerfacecolor*.
            - 'left', 'right', 'bottom', 'top': Fill the marker half at
              the given side with the *markerfacecolor*. The other
              half of the marker is filled with *markerfacecoloralt*.
            - 'none': No filling.
            For examples see :ref:`marker_fill_styles`.
        """
        self.set_marker(MarkerStyle(self._marker.get_marker(), fs))
        self.stale = True
    def set_markevery(self, every):
        """
        Set the markevery property to subsample the plot when using markers.
        e.g., if ``every=5``, every 5-th marker will be plotted.
        Parameters
        ----------
        every : None or int or (int, int) or slice or list[int] or float or \
(float, float) or list[bool]
            Which markers to plot.
            - ``every=None``: every point will be plotted.
            - ``every=N``: every N-th marker will be plotted starting with
              marker 0.
            - ``every=(start, N)``: every N-th marker, starting at index
              *start*, will be plotted.
            - ``every=slice(start, end, N)``: every N-th marker, starting at
              index *start*, up to but not including index *end*, will be
              plotted.
            - ``every=[i, j, m, ...]``: only markers at the given indices
              will be plotted.
            - ``every=[True, False, True, ...]``: only positions that are True
              will be plotted. The list must have the same length as the data
              points.
            - ``every=0.1``, (i.e. a float): markers will be spaced at
              approximately equal visual distances along the line; the
distance
              along the line between markers is determined by multiplying the
              display-coordinate distance of the Axes bounding-box diagonal
              by the value of *every*.
            - ``every=(0.5, 0.1)`` (i.e. a length-2 tuple of float): similar
              to ``every=0.1`` but the first marker will be offset along the
              line by 0.5 multiplied by the
              display-coordinate-diagonal-distance along the line.
            For examples see
            :doc:`/gallery/lines_bars_and_markers/markevery_demo`.
        Notes
        -----
        Setting *markevery* will still only draw markers at actual data
points.
        While the float argument form aims for uniform visual spacing, it has
        to coerce from the ideal spacing to the nearest available data point.
        Depending on the number and distribution of data points, the result
        may still not look evenly spaced.
        When using a start offset to specify the first marker, the offset will
        be from the first data point which may be different from the first
        the visible data point if the plot is zoomed in.
        If zooming in on a plot when using float arguments then the actual
        data points that have markers will change because the distance between
        markers is always determined from the display-coordinates
        axes-bounding-box-diagonal regardless of the actual axes data limits.
        """
        self._markevery = every
        self.stale = True
    def get_markevery(self):
```

```
(8)                 """
(8)                 Return the markevery setting for marker subsampling.
(8)                 See also `~.Line2D.set_markevery`.
(8)                 """
(8)                 return self._markevery
(4)             def set_picker(self, p):
(8)                 """
(8)                 Set the event picker details for the line.
(8)                 Parameters
(8)                 ----------
(8)                 p : float or callable[[Artist, Event], tuple[bool, dict]]
(12)                    If a float, it is used as the pick radius in points.
(8)                 """
(8)                 if not callable(p):
(12)                    self.set_pickradius(p)
(8)                 self._picker = p
(4)             def get_bbox(self):
(8)                 """Get the bounding box of this line."""
(8)                 bbox = Bbox([[0, 0], [0, 0]])
(8)                 bbox.update_from_data_xy(self.get_xydata())
(8)                 return bbox
(4)             def get_window_extent(self, renderer=None):
(8)                 bbox = Bbox([[0, 0], [0, 0]])
(8)                 trans_data_to_xy = self.get_transform().transform
(8)                 bbox.update_from_data_xy(trans_data_to_xy(self.get_xydata()),
(33)                                         ignore=True)
(8)                 if self._marker:
(12)                    ms = (self._markersize / 72.0 * self.figure.dpi) * 0.5
(12)                    bbox = bbox.padded(ms)
(8)                 return bbox
(4)             def set_data(self, *args):
(8)                 """
(8)                 Set the x and y data.
(8)                 Parameters
(8)                 ----------
(8)                 *args : (2, N) array or two 1D arrays
(8)                 See Also
(8)                 --------
(8)                 set_xdata
(8)                 set_ydata
(8)                 """
(8)                 if len(args) == 1:
(12)                    (x, y), = args
(8)                 else:
(12)                    x, y = args
(8)                 self.set_xdata(x)
(8)                 self.set_ydata(y)
(4)             def recache_always(self):
(8)                 self.recache(always=True)
(4)             def recache(self, always=False):
(8)                 if always or self._invalidx:
(12)                    xconv = self.convert_xunits(self._xorig)
(12)                    x = _to_unmasked_float_array(xconv).ravel()
(8)                 else:
(12)                    x = self._x
(8)                 if always or self._invalidy:
(12)                    yconv = self.convert_yunits(self._yorig)
(12)                    y = _to_unmasked_float_array(yconv).ravel()
(8)                 else:
(12)                    y = self._y
(8)                 self._xy = np.column_stack(np.broadcast_arrays(x, y)).astype(float)
(8)                 self._x, self._y = self._xy.T  # views
(8)                 self._subslice = False
(8)                 if (self.axes
(16)                        and len(x) > self._subslice_optim_min_size
(16)                        and _path.is_sorted_and_has_non_nan(x)
(16)                        and self.axes.name == 'rectilinear'
(16)                        and self.axes.get_xscale() == 'linear'
(16)                        and self._markevery is None
```

```
(16)                            and self.get_clip_on()
(16)                            and self.get_transform() == self.axes.transData):
(12)                        self._subslice = True
(12)                        nanmask = np.isnan(x)
(12)                        if nanmask.any():
(16)                            self._x_filled = self._x.copy()
(16)                            indices = np.arange(len(x))
(16)                            self._x_filled[nanmask] = np.interp(
(20)                                indices[nanmask], indices[~nanmask], self._x[~nanmask])
(12)                        else:
(16)                            self._x_filled = self._x
(8)                     if self._path is not None:
(12)                        interpolation_steps = self._path._interpolation_steps
(8)                     else:
(12)                        interpolation_steps = 1
(8)                     xy = STEP_LOOKUP_MAP[self._drawstyle](*self._xy.T)
(8)                     self._path = Path(np.asarray(xy).T,
(26)                                      _interpolation_steps=interpolation_steps)
(8)                     self._transformed_path = None
(8)                     self._invalidx = False
(8)                     self._invalidy = False
(4)                 def _transform_path(self, subslice=None):
(8)                     """
(8)                     Put a TransformedPath instance at self._transformed_path;
(8)                     all invalidation of the transform is then handled by the
(8)                     TransformedPath instance.
(8)                     """
(8)                     if subslice is not None:
(12)                        xy = STEP_LOOKUP_MAP[self._drawstyle](*self._xy[subslice, :].T)
(12)                        _path = Path(np.asarray(xy).T,
(25)                                     _interpolation_steps=self._path._interpolation_steps)
(8)                     else:
(12)                        _path = self._path
(8)                     self._transformed_path = TransformedPath(_path, self.get_transform())
(4)                 def _get_transformed_path(self):
(8)                     """Return this line's `~matplotlib.transforms.TransformedPath`."""
(8)                     if self._transformed_path is None:
(12)                        self._transform_path()
(8)                     return self._transformed_path
(4)                 def set_transform(self, t):
(8)                     self._invalidx = True
(8)                     self._invalidy = True
(8)                     super().set_transform(t)
(4)                 @allow_rasterization
(4)                 def draw(self, renderer):
(8)                     if not self.get_visible():
(12)                        return
(8)                     if self._invalidy or self._invalidx:
(12)                        self.recache()
(8)                     self.ind_offset = 0  # Needed for contains() method.
(8)                     if self._subslice and self.axes:
(12)                        x0, x1 = self.axes.get_xbound()
(12)                        i0 = self._x_filled.searchsorted(x0, 'left')
(12)                        i1 = self._x_filled.searchsorted(x1, 'right')
(12)                        subslice = slice(max(i0 - 1, 0), i1 + 1)
(12)                        self.ind_offset = subslice.start
(12)                        self._transform_path(subslice)
(8)                     else:
(12)                        subslice = None
(8)                     if self.get_path_effects():
(12)                        from matplotlib.patheffects import PathEffectRenderer
(12)                        renderer = PathEffectRenderer(self.get_path_effects(), renderer)
(8)                     renderer.open_group('line2d', self.get_gid())
(8)                     if self._lineStyles[self._linestyle] != '_draw_nothing':
(12)                        tpath, affine = (self._get_transformed_path()
(29)                                         .get_transformed_path_and_affine())
(12)                        if len(tpath.vertices):
(16)                            gc = renderer.new_gc()
(16)                            self._set_gc_clip(gc)
```

```
598: (16)                            gc.set_url(self.get_url())
599: (16)                            gc.set_antialiased(self._antialiased)
600: (16)                            gc.set_linewidth(self._linewidth)
601: (16)                            if self.is_dashed():
602: (20)                                cap = self._dashcapstyle
603: (20)                                join = self._dashjoinstyle
604: (16)                            else:
605: (20)                                cap = self._solidcapstyle
606: (20)                                join = self._solidjoinstyle
607: (16)                            gc.set_joinstyle(join)
608: (16)                            gc.set_capstyle(cap)
609: (16)                            gc.set_snap(self.get_snap())
610: (16)                            if self.get_sketch_params() is not None:
611: (20)                                gc.set_sketch_params(*self.get_sketch_params())
612: (16)                            if self.is_dashed() and self._gapcolor is not None:
613: (20)                                lc_rgba = mcolors.to_rgba(self._gapcolor, self._alpha)
614: (20)                                gc.set_foreground(lc_rgba, isRGBA=True)
615: (20)                                offset_gaps, gaps = _get_inverse_dash_pattern(
616: (24)                                    *self._dash_pattern)
617: (20)                                gc.set_dashes(offset_gaps, gaps)
618: (20)                                renderer.draw_path(gc, tpath, affine.frozen())
619: (16)                            lc_rgba = mcolors.to_rgba(self._color, self._alpha)
620: (16)                            gc.set_foreground(lc_rgba, isRGBA=True)
621: (16)                            gc.set_dashes(*self._dash_pattern)
622: (16)                            renderer.draw_path(gc, tpath, affine.frozen())
623: (16)                            gc.restore()
624: (8)                    if self._marker and self._markersize > 0:
625: (12)                        gc = renderer.new_gc()
626: (12)                        self._set_gc_clip(gc)
627: (12)                        gc.set_url(self.get_url())
628: (12)                        gc.set_linewidth(self._markeredgewidth)
629: (12)                        gc.set_antialiased(self._antialiased)
630: (12)                        ec_rgba = mcolors.to_rgba(
631: (16)                            self.get_markeredgecolor(), self._alpha)
632: (12)                        fc_rgba = mcolors.to_rgba(
633: (16)                            self._get_markerfacecolor(), self._alpha)
634: (12)                        fcalt_rgba = mcolors.to_rgba(
635: (16)                            self._get_markerfacecolor(alt=True), self._alpha)
636: (12)                        if (cbook._str_equal(self._markeredgecolor, "auto")
637: (20)                                and not cbook._str_lower_equal(
638: (24)                                    self.get_markerfacecolor(), "none")):
639: (16)                            ec_rgba = ec_rgba[:3] + (fc_rgba[3],)
640: (12)                        gc.set_foreground(ec_rgba, isRGBA=True)
641: (12)                        if self.get_sketch_params() is not None:
642: (16)                            scale, length, randomness = self.get_sketch_params()
643: (16)                            gc.set_sketch_params(scale/2, length/2, 2*randomness)
644: (12)                        marker = self._marker
645: (12)                        if self.get_drawstyle() != "default":
646: (16)                            with cbook._setattr_cm(
647: (24)                                    self, _drawstyle="default", _transformed_path=None):
648: (20)                                self.recache()
649: (20)                                self._transform_path(subslice)
650: (20)                                tpath, affine = (self._get_transformed_path()
651: (37)                                                 .get_transformed_points_and_affine())
652: (12)                        else:
653: (16)                            tpath, affine = (self._get_transformed_path()
654: (33)                                             .get_transformed_points_and_affine())
655: (12)                        if len(tpath.vertices):
656: (16)                            markevery = self.get_markevery()
657: (16)                            if markevery is not None:
658: (20)                                subsampled = _mark_every_path(
659: (24)                                    markevery, tpath, affine, self.axes)
660: (16)                            else:
661: (20)                                subsampled = tpath
662: (16)                            snap = marker.get_snap_threshold()
663: (16)                            if isinstance(snap, Real):
664: (20)                                snap = renderer.points_to_pixels(self._markersize) >= snap
665: (16)                            gc.set_snap(snap)
666: (16)                            gc.set_joinstyle(marker.get_joinstyle())
```

```
(16)                            gc.set_capstyle(marker.get_capstyle())
(16)                            marker_path = marker.get_path()
(16)                            marker_trans = marker.get_transform()
(16)                            w = renderer.points_to_pixels(self._markersize)
(16)                            if cbook._str_equal(marker.get_marker(), ","):
(20)                                gc.set_linewidth(0)
(16)                            else:
(20)                                marker_trans = marker_trans.scale(w)
(16)                            renderer.draw_markers(gc, marker_path, marker_trans,
(38)                                                  subsampled, affine.frozen(),
(38)                                                  fc_rgba)
(16)                            alt_marker_path = marker.get_alt_path()
(16)                            if alt_marker_path:
(20)                                alt_marker_trans = marker.get_alt_transform()
(20)                                alt_marker_trans = alt_marker_trans.scale(w)
(20)                                renderer.draw_markers(
(28)                                        gc, alt_marker_path, alt_marker_trans, subsampled,
(28)                                        affine.frozen(), fcalt_rgba)
(12)                        gc.restore()
(8)                     renderer.close_group('line2d')
(8)                     self.stale = False
(4)                 def get_antialiased(self):
(8)                     """Return whether antialiased rendering is used."""
(8)                     return self._antialiased
(4)                 def get_color(self):
(8)                     """
(8)                     Return the line color.
(8)                     See also `~.Line2D.set_color`.
(8)                     """
(8)                     return self._color
(4)                 def get_drawstyle(self):
(8)                     """
(8)                     Return the drawstyle.
(8)                     See also `~.Line2D.set_drawstyle`.
(8)                     """
(8)                     return self._drawstyle
(4)                 def get_gapcolor(self):
(8)                     """
(8)                     Return the line gapcolor.
(8)                     See also `~.Line2D.set_gapcolor`.
(8)                     """
(8)                     return self._gapcolor
(4)                 def get_linestyle(self):
(8)                     """
(8)                     Return the linestyle.
(8)                     See also `~.Line2D.set_linestyle`.
(8)                     """
(8)                     return self._linestyle
(4)                 def get_linewidth(self):
(8)                     """
(8)                     Return the linewidth in points.
(8)                     See also `~.Line2D.set_linewidth`.
(8)                     """
(8)                     return self._linewidth
(4)                 def get_marker(self):
(8)                     """
(8)                     Return the line marker.
(8)                     See also `~.Line2D.set_marker`.
(8)                     """
(8)                     return self._marker.get_marker()
(4)                 def get_markeredgecolor(self):
(8)                     """
(8)                     Return the marker edge color.
(8)                     See also `~.Line2D.set_markeredgecolor`.
(8)                     """
(8)                     mec = self._markeredgecolor
(8)                     if cbook._str_equal(mec, 'auto'):
(12)                        if mpl.rcParams['_internal.classic_mode']:
(16)                            if self._marker.get_marker() in ('.', ','):
```

```
(20)                                return self._color
(16)                            if (self._marker.is_filled()
(24)                                    and self._marker.get_fillstyle() != 'none'):
(20)                                return 'k'  # Bad hard-wired default...
(12)                        return self._color
(8)                     else:
(12)                        return mec
(4)                 def get_markeredgewidth(self):
(8)                     """
(8)                     Return the marker edge width in points.
(8)                     See also `~.Line2D.set_markeredgewidth`.
(8)                     """
(8)                     return self._markeredgewidth
(4)                 def _get_markerfacecolor(self, alt=False):
(8)                     if self._marker.get_fillstyle() == 'none':
(12)                        return 'none'
(8)                     fc = self._markerfacecoloralt if alt else self._markerfacecolor
(8)                     if cbook._str_lower_equal(fc, 'auto'):
(12)                        return self._color
(8)                     else:
(12)                        return fc
(4)                 def get_markerfacecolor(self):
(8)                     """
(8)                     Return the marker face color.
(8)                     See also `~.Line2D.set_markerfacecolor`.
(8)                     """
(8)                     return self._get_markerfacecolor(alt=False)
(4)                 def get_markerfacecoloralt(self):
(8)                     """
(8)                     Return the alternate marker face color.
(8)                     See also `~.Line2D.set_markerfacecoloralt`.
(8)                     """
(8)                     return self._get_markerfacecolor(alt=True)
(4)                 def get_markersize(self):
(8)                     """
(8)                     Return the marker size in points.
(8)                     See also `~.Line2D.set_markersize`.
(8)                     """
(8)                     return self._markersize
(4)                 def get_data(self, orig=True):
(8)                     """
(8)                     Return the line data as an ``(xdata, ydata)`` pair.
(8)                     If *orig* is *True*, return the original data.
(8)                     """
(8)                     return self.get_xdata(orig=orig), self.get_ydata(orig=orig)
(4)                 def get_xdata(self, orig=True):
(8)                     """
(8)                     Return the xdata.
(8)                     If *orig* is *True*, return the original data, else the
(8)                     processed data.
(8)                     """
(8)                     if orig:
(12)                        return self._xorig
(8)                     if self._invalidx:
(12)                        self.recache()
(8)                     return self._x
(4)                 def get_ydata(self, orig=True):
(8)                     """
(8)                     Return the ydata.
(8)                     If *orig* is *True*, return the original data, else the
(8)                     processed data.
(8)                     """
(8)                     if orig:
(12)                        return self._yorig
(8)                     if self._invalidy:
(12)                        self.recache()
(8)                     return self._y
(4)                 def get_path(self):
(8)                     """Return the `~matplotlib.path.Path` associated with this line."""
```

```
        if self._invalidy or self._invalidx:
            self.recache()
        return self._path
    def get_xydata(self):
        """Return the *xy* data as a (N, 2) array."""
        if self._invalidy or self._invalidx:
            self.recache()
        return self._xy
    def set_antialiased(self, b):
        """
        Set whether to use antialiased rendering.
        Parameters
        ----------
        b : bool
        """
        if self._antialiased != b:
            self.stale = True
        self._antialiased = b
    def set_color(self, color):
        """
        Set the color of the line.
        Parameters
        ----------
        color : :mpltype:`color`
        """
        mcolors._check_color_like(color=color)
        self._color = color
        self.stale = True
    def set_drawstyle(self, drawstyle):
        """
        Set the drawstyle of the plot.
        The drawstyle determines how the points are connected.
        Parameters
        ----------
        drawstyle : {'default', 'steps', 'steps-pre', 'steps-mid', \
'steps-post'}, default: 'default'
            For 'default', the points are connected with straight lines.
            The steps variants connect the points with step-like lines,
            i.e. horizontal lines with vertical steps. They differ in the
            location of the step:
            - 'steps-pre': The step is at the beginning of the line segment,
              i.e. the line will be at the y-value of point to the right.
            - 'steps-mid': The step is halfway between the points.
            - 'steps-post: The step is at the end of the line segment,
              i.e. the line will be at the y-value of the point to the left.
            - 'steps' is equal to 'steps-pre' and is maintained for
              backward-compatibility.
            For examples see :doc:`/gallery/lines_bars_and_markers/step_demo`.
        """
        if drawstyle is None:
            drawstyle = 'default'
        _api.check_in_list(self.drawStyles, drawstyle=drawstyle)
        if self._drawstyle != drawstyle:
            self.stale = True
            self._invalidx = True
        self._drawstyle = drawstyle
    def set_gapcolor(self, gapcolor):
        """
        Set a color to fill the gaps in the dashed line style.
        .. note::
            Striped lines are created by drawing two interleaved dashed lines.
            There can be overlaps between those two, which may result in
            artifacts when using transparency.
            This functionality is experimental and may change.
        Parameters
        ----------
        gapcolor : :mpltype:`color` or None
            The color with which to fill the gaps. If None, the gaps are
            unfilled.
```

```
(8)                     """
(8)                     if gapcolor is not None:
(12)                        mcolors._check_color_like(color=gapcolor)
(8)                     self._gapcolor = gapcolor
(8)                     self.stale = True
(4)                 def set_linewidth(self, w):
(8)                     """
(8)                     Set the line width in points.
(8)                     Parameters
(8)                     ----------
(8)                     w : float
(12)                        Line width, in points.
(8)                     """
(8)                     w = float(w)
(8)                     if self._linewidth != w:
(12)                        self.stale = True
(8)                     self._linewidth = w
(8)                     self._dash_pattern = _scale_dashes(*self._unscaled_dash_pattern, w)
(4)                 def set_linestyle(self, ls):
(8)                     """
(8)                     Set the linestyle of the line.
(8)                     Parameters
(8)                     ----------
(8)                     ls : {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
(12)                        Possible values:
(12)                        - A string:
(14)                          ========================================  =================
(14)                          linestyle                                 description
(14)                          ========================================  =================
(14)                          ``'-'`` or ``'solid'``                    solid line
(14)                          ``'--'`` or  ``'dashed'``                 dashed line
(14)                          ``'-.'`` or  ``'dashdot'``                dash-dotted line
(14)                          ``':'`` or ``'dotted'``                   dotted line
(14)                          ``'none'``, ``'None'``, ``' '``, or ``''``  draw nothing
(14)                          ========================================  =================
(12)                        - Alternatively a dash tuple of the following form can be
(14)                          provided::
(18)                              (offset, onoffseq)
(14)                          where ``onoffseq`` is an even length tuple of on and off ink
(14)                          in points. See also :meth:`set_dashes`.
(12)                        For examples see
:doc:`/gallery/lines_bars_and_markers/linestyles`.
(8)                     """
(8)                     if isinstance(ls, str):
(12)                        if ls in [' ', '', 'none']:
(16)                            ls = 'None'
(12)                        _api.check_in_list([*self._lineStyles, *ls_mapper_r], ls=ls)
(12)                        if ls not in self._lineStyles:
(16)                            ls = ls_mapper_r[ls]
(12)                        self._linestyle = ls
(8)                     else:
(12)                        self._linestyle = '--'
(8)                     self._unscaled_dash_pattern = _get_dash_pattern(ls)
(8)                     self._dash_pattern = _scale_dashes(
(12)                        *self._unscaled_dash_pattern, self._linewidth)
(8)                     self.stale = True
(4)                 @_docstring.interpd
(4)                 def set_marker(self, marker):
(8)                     """
(8)                     Set the line marker.
(8)                     Parameters
(8)                     ----------
(8)                     marker : marker style string, `~.path.Path` or `~.markers.MarkerStyle`
(12)                        See `~matplotlib.markers` for full description of possible
(12)                        arguments.
(8)                     """
(8)                     self._marker = MarkerStyle(marker, self._marker.get_fillstyle())
(8)                     self.stale = True
(4)                 def _set_markercolor(self, name, has_rcdefault, val):
```

```
(8)                     if val is None:
(12)                        val = mpl.rcParams[f"lines.{name}"] if has_rcdefault else "auto"
(8)                     attr = f"_{name}"
(8)                     current = getattr(self, attr)
(8)                     if current is None:
(12)                        self.stale = True
(8)                     else:
(12)                        neq = current != val
(12)                        if neq.any() if isinstance(neq, np.ndarray) else neq:
(16)                            self.stale = True
(8)                     setattr(self, attr, val)
(4)                 def set_markeredgecolor(self, ec):
(8)                     """
(8)                     Set the marker edge color.
(8)                     Parameters
(8)                     ----------
(8)                     ec : :mpltype:`color`
(8)                     """
(8)                     self._set_markercolor("markeredgecolor", True, ec)
(4)                 def set_markerfacecolor(self, fc):
(8)                     """
(8)                     Set the marker face color.
(8)                     Parameters
(8)                     ----------
(8)                     fc : :mpltype:`color`
(8)                     """
(8)                     self._set_markercolor("markerfacecolor", True, fc)
(4)                 def set_markerfacecoloralt(self, fc):
(8)                     """
(8)                     Set the alternate marker face color.
(8)                     Parameters
(8)                     ----------
(8)                     fc : :mpltype:`color`
(8)                     """
(8)                     self._set_markercolor("markerfacecoloralt", False, fc)
(4)                 def set_markeredgewidth(self, ew):
(8)                     """
(8)                     Set the marker edge width in points.
(8)                     Parameters
(8)                     ----------
(8)                     ew : float
(13)                         Marker edge width, in points.
(8)                     """
(8)                     if ew is None:
(12)                        ew = mpl.rcParams['lines.markeredgewidth']
(8)                     if self._markeredgewidth != ew:
(12)                        self.stale = True
(8)                     self._markeredgewidth = ew
(4)                 def set_markersize(self, sz):
(8)                     """
(8)                     Set the marker size in points.
(8)                     Parameters
(8)                     ----------
(8)                     sz : float
(13)                         Marker size, in points.
(8)                     """
(8)                     sz = float(sz)
(8)                     if self._markersize != sz:
(12)                        self.stale = True
(8)                     self._markersize = sz
(4)                 def set_xdata(self, x):
(8)                     """
(8)                     Set the data array for x.
(8)                     Parameters
(8)                     ----------
(8)                     x : 1D array
(8)                     See Also
(8)                     --------
(8)                     set_data
```

```
(8)                 set_ydata
(8)                 """
(8)                 if not np.iterable(x):
(12)                    raise RuntimeError('x must be a sequence')
(8)                 self._xorig = copy.copy(x)
(8)                 self._invalidx = True
(8)                 self.stale = True
(4)             def set_ydata(self, y):
(8)                 """
(8)                 Set the data array for y.
(8)                 Parameters
(8)                 ----------
(8)                 y : 1D array
(8)                 See Also
(8)                 --------
(8)                 set_data
(8)                 set_xdata
(8)                 """
(8)                 if not np.iterable(y):
(12)                    raise RuntimeError('y must be a sequence')
(8)                 self._yorig = copy.copy(y)
(8)                 self._invalidy = True
(8)                 self.stale = True
(4)             def set_dashes(self, seq):
(8)                 """
(8)                 Set the dash sequence.
(8)                 The dash sequence is a sequence of floats of even length describing
(8)                 the length of dashes and spaces in points.
(8)                 For example, (5, 2, 1, 2) describes a sequence of 5 point and 1 point
(8)                 dashes separated by 2 point spaces.
(8)                 See also `~.Line2D.set_gapcolor`, which allows those spaces to be
(8)                 filled with a color.
(8)                 Parameters
(8)                 ----------
(8)                 seq : sequence of floats (on/off ink in points) or (None, None)
(12)                    If *seq* is empty or ``(None, None)``, the linestyle will be set
(12)                    to solid.
(8)                 """
(8)                 if seq == (None, None) or len(seq) == 0:
(12)                    self.set_linestyle('-')
(8)                 else:
(12)                    self.set_linestyle((0, seq))
(4)             def update_from(self, other):
(8)                 """Copy properties from *other* to self."""
(8)                 super().update_from(other)
(8)                 self._linestyle = other._linestyle
(8)                 self._linewidth = other._linewidth
(8)                 self._color = other._color
(8)                 self._gapcolor = other._gapcolor
(8)                 self._markersize = other._markersize
(8)                 self._markerfacecolor = other._markerfacecolor
(8)                 self._markerfacecoloralt = other._markerfacecoloralt
(8)                 self._markeredgecolor = other._markeredgecolor
(8)                 self._markeredgewidth = other._markeredgewidth
(8)                 self._unscaled_dash_pattern = other._unscaled_dash_pattern
(8)                 self._dash_pattern = other._dash_pattern
(8)                 self._dashcapstyle = other._dashcapstyle
(8)                 self._dashjoinstyle = other._dashjoinstyle
(8)                 self._solidcapstyle = other._solidcapstyle
(8)                 self._solidjoinstyle = other._solidjoinstyle
(8)                 self._linestyle = other._linestyle
(8)                 self._marker = MarkerStyle(marker=other._marker)
(8)                 self._drawstyle = other._drawstyle
(4)             @_docstring.interpd
(4)             def set_dash_joinstyle(self, s):
(8)                 """
(8)                 How to join segments of the line if it `~Line2D.is_dashed`.
(8)                 The default joinstyle is :rc:`lines.dash_joinstyle`.
(8)                 Parameters
```

```
(8)                     ----------
(8)                     s : `.JoinStyle` or %(JoinStyle)s
(8)                     """
(8)                     js = JoinStyle(s)
(8)                     if self._dashjoinstyle != js:
(12)                        self.stale = True
(8)                     self._dashjoinstyle = js
(4)                 @_docstring.interpd
(4)                 def set_solid_joinstyle(self, s):
(8)                     """
(8)                     How to join segments if the line is solid (not `~Line2D.is_dashed`).
(8)                     The default joinstyle is :rc:`lines.solid_joinstyle`.
(8)                     Parameters
(8)                     ----------
(8)                     s : `.JoinStyle` or %(JoinStyle)s
(8)                     """
(8)                     js = JoinStyle(s)
(8)                     if self._solidjoinstyle != js:
(12)                        self.stale = True
(8)                     self._solidjoinstyle = js
(4)                 def get_dash_joinstyle(self):
(8)                     """
(8)                     Return the `.JoinStyle` for dashed lines.
(8)                     See also `~.Line2D.set_dash_joinstyle`.
(8)                     """
(8)                     return self._dashjoinstyle.name
(4)                 def get_solid_joinstyle(self):
(8)                     """
(8)                     Return the `.JoinStyle` for solid lines.
(8)                     See also `~.Line2D.set_solid_joinstyle`.
(8)                     """
(8)                     return self._solidjoinstyle.name
(4)                 @_docstring.interpd
(4)                 def set_dash_capstyle(self, s):
(8)                     """
(8)                     How to draw the end caps if the line is `~Line2D.is_dashed`.
(8)                     The default capstyle is :rc:`lines.dash_capstyle`.
(8)                     Parameters
(8)                     ----------
(8)                     s : `.CapStyle` or %(CapStyle)s
(8)                     """
(8)                     cs = CapStyle(s)
(8)                     if self._dashcapstyle != cs:
(12)                        self.stale = True
(8)                     self._dashcapstyle = cs
(4)                 @_docstring.interpd
(4)                 def set_solid_capstyle(self, s):
(8)                     """
(8)                     How to draw the end caps if the line is solid (not
`~Line2D.is_dashed`)
(8)                     The default capstyle is :rc:`lines.solid_capstyle`.
(8)                     Parameters
(8)                     ----------
(8)                     s : `.CapStyle` or %(CapStyle)s
(8)                     """
(8)                     cs = CapStyle(s)
(8)                     if self._solidcapstyle != cs:
(12)                        self.stale = True
(8)                     self._solidcapstyle = cs
(4)                 def get_dash_capstyle(self):
(8)                     """
(8)                     Return the `.CapStyle` for dashed lines.
(8)                     See also `~.Line2D.set_dash_capstyle`.
(8)                     """
(8)                     return self._dashcapstyle.name
(4)                 def get_solid_capstyle(self):
(8)                     """
(8)                     Return the `.CapStyle` for solid lines.
(8)                     See also `~.Line2D.set_solid_capstyle`.
```

```
1148: (8)                    """
1149: (8)                    return self._solidcapstyle.name
1150: (4)                def is_dashed(self):
1151: (8)                    """
1152: (8)                    Return whether line has a dashed linestyle.
1153: (8)                    A custom linestyle is assumed to be dashed, we do not inspect the
1154: (8)                    ``onoffseq`` directly.
1155: (8)                    See also `~.Line2D.set_linestyle`.
1156: (8)                    """
1157: (8)                    return self._linestyle in ('--', '-.', ':')
1158: (0)            class AxLine(Line2D):
1159: (4)                """
1160: (4)                A helper class that implements `~.Axes.axline`, by recomputing the artist
1161: (4)                transform at draw time.
1162: (4)                """
1163: (4)                def __init__(self, xy1, xy2, slope, **kwargs):
1164: (8)                    """
1165: (8)                    Parameters
1166: (8)                    ----------
1167: (8)                    xy1 : (float, float)
1168: (12)                       The first set of (x, y) coordinates for the line to pass through.
1169: (8)                    xy2 : (float, float) or None
1170: (12)                       The second set of (x, y) coordinates for the line to pass through.
1171: (12)                       Both *xy2* and *slope* must be passed, but one of them must be
None.
1172: (8)                    slope : float or None
1173: (12)                       The slope of the line. Both *xy2* and *slope* must be passed, but
one of
1174: (12)                       them must be None.
1175: (8)                    """
1176: (8)                    super().__init__([0, 1], [0, 1], **kwargs)
1177: (8)                    if (xy2 is None and slope is None or
1178: (16)                           xy2 is not None and slope is not None):
1179: (12)                       raise TypeError(
1180: (16)                           "Exactly one of 'xy2' and 'slope' must be given")
1181: (8)                    self._slope = slope
1182: (8)                    self._xy1 = xy1
1183: (8)                    self._xy2 = xy2
1184: (4)                def get_transform(self):
1185: (8)                    ax = self.axes
1186: (8)                    points_transform = self._transform - ax.transData + ax.transScale
1187: (8)                    if self._xy2 is not None:
1188: (12)                       (x1, y1), (x2, y2) = \
1189: (16)                           points_transform.transform([self._xy1, self._xy2])
1190: (12)                       dx = x2 - x1
1191: (12)                       dy = y2 - y1
1192: (12)                       if np.allclose(x1, x2):
1193: (16)                           if np.allclose(y1, y2):
1194: (20)                               raise ValueError(
1195: (24)                                   f"Cannot draw a line through two identical points "
1196: (24)                                   f"(x={(x1, x2)}, y={(y1, y2)})")
1197: (16)                           slope = np.inf
1198: (12)                       else:
1199: (16)                           slope = dy / dx
1200: (8)                    else:
1201: (12)                       x1, y1 = points_transform.transform(self._xy1)
1202: (12)                       slope = self._slope
1203: (8)                    (vxlo, vylo), (vxhi, vyhi) = ax.transScale.transform(ax.viewLim)
1204: (8)                    if np.isclose(slope, 0):
1205: (12)                       start = vxlo, y1
1206: (12)                       stop = vxhi, y1
1207: (8)                    elif np.isinf(slope):
1208: (12)                       start = x1, vylo
1209: (12)                       stop = x1, vyhi
1210: (8)                    else:
1211: (12)                       _, start, stop, _ = sorted([
1212: (16)                           (vxlo, y1 + (vxlo - x1) * slope),
1213: (16)                           (vxhi, y1 + (vxhi - x1) * slope),
1214: (16)                           (x1 + (vylo - y1) / slope, vylo),
```

```
(16)                        (x1 + (vyhi - y1) / slope, vyhi),
(12)                    ])
(8)                 return (BboxTransformTo(Bbox([start, stop]))
(16)                        + ax.transLimits + ax.transAxes)
(4)             def draw(self, renderer):
(8)                 self._transformed_path = None  # Force regen.
(8)                 super().draw(renderer)
(4)             def get_xy1(self):
(8)                 """
(8)                 Return the *xy1* value of the line.
(8)                 """
(8)                 return self._xy1
(4)             def get_xy2(self):
(8)                 """
(8)                 Return the *xy2* value of the line.
(8)                 """
(8)                 return self._xy2
(4)             def get_slope(self):
(8)                 """
(8)                 Return the *slope* value of the line.
(8)                 """
(8)                 return self._slope
(4)             def set_xy1(self, x, y):
(8)                 """
(8)                 Set the *xy1* value of the line.
(8)                 Parameters
(8)                 ----------
(8)                 x, y : float
(12)                    Points for the line to pass through.
(8)                 """
(8)                 self._xy1 = x, y
(4)             def set_xy2(self, x, y):
(8)                 """
(8)                 Set the *xy2* value of the line.
(8)                 Parameters
(8)                 ----------
(8)                 x, y : float
(12)                    Points for the line to pass through.
(8)                 """
(8)                 if self._slope is None:
(12)                    self._xy2 = x, y
(8)                 else:
(12)                    raise ValueError("Cannot set an 'xy2' value while 'slope' is set;"
(29)                                     " they differ but their functionalities overlap")
(4)             def set_slope(self, slope):
(8)                 """
(8)                 Set the *slope* value of the line.
(8)                 Parameters
(8)                 ----------
(8)                 slope : float
(12)                    The slope of the line.
(8)                 """
(8)                 if self._xy2 is None:
(12)                    self._slope = slope
(8)                 else:
(12)                    raise ValueError("Cannot set a 'slope' value while 'xy2' is set;"
(29)                                     " they differ but their functionalities overlap")
(0)         class VertexSelector:
(4)             """
(4)             Manage the callbacks to maintain a list of selected vertices for
`.Line2D`.
(4)             Derived classes should override the `process_selected` method to do
(4)             something with the picks.
(4)             Here is an example which highlights the selected verts with red circles::
(8)                 import numpy as np
(8)                 import matplotlib.pyplot as plt
(8)                 import matplotlib.lines as lines
(8)                 class HighlightSelected(lines.VertexSelector):
(12)                    def __init__(self, line, fmt='ro', **kwargs):
```

```
1283: (16)                          super().__init__(line)
1284: (16)                          self.markers, = self.axes.plot([], [], fmt, **kwargs)
1285: (12)                      def process_selected(self, ind, xs, ys):
1286: (16)                          self.markers.set_data(xs, ys)
1287: (16)                          self.canvas.draw()
1288: (8)                   fig, ax = plt.subplots()
1289: (8)                   x, y = np.random.rand(2, 30)
1290: (8)                   line, = ax.plot(x, y, 'bs-', picker=5)
1291: (8)                   selector = HighlightSelected(line)
1292: (8)                   plt.show()
1293: (4)               """
1294: (4)               def __init__(self, line):
1295: (8)                   """
1296: (8)                   Parameters
1297: (8)                   ----------
1298: (8)                   line : `~matplotlib.lines.Line2D`
1299: (12)                      The line must already have been added to an `~.axes.Axes` and must
1300: (12)                      have its picker property set.
1301: (8)                   """
1302: (8)                   if line.axes is None:
1303: (12)                      raise RuntimeError('You must first add the line to the Axes')
1304: (8)                   if line.get_picker() is None:
1305: (12)                      raise RuntimeError('You must first set the picker property '
1306: (31)                                         'of the line')
1307: (8)                   self.axes = line.axes
1308: (8)                   self.line = line
1309: (8)                   self.cid = self.canvas.callbacks._connect_picklable(
1310: (12)                      'pick_event', self.onpick)
1311: (8)                   self.ind = set()
1312: (4)               canvas = property(lambda self: self.axes.figure.canvas)
1313: (4)               def process_selected(self, ind, xs, ys):
1314: (8)                   """
1315: (8)                   Default "do nothing" implementation of the `process_selected` method.
1316: (8)                   Parameters
1317: (8)                   ----------
1318: (8)                   ind : list of int
1319: (12)                      The indices of the selected vertices.
1320: (8)                   xs, ys : array-like
1321: (12)                      The coordinates of the selected vertices.
1322: (8)                   """
1323: (8)                   pass
1324: (4)               def onpick(self, event):
1325: (8)                   """When the line is picked, update the set of selected indices."""
1326: (8)                   if event.artist is not self.line:
1327: (12)                      return
1328: (8)                   self.ind ^= set(event.ind)
1329: (8)                   ind = sorted(self.ind)
1330: (8)                   xdata, ydata = self.line.get_data()
1331: (8)                   self.process_selected(ind, xdata[ind], ydata[ind])
1332: (0)           lineStyles = Line2D._lineStyles
1333: (0)           lineMarkers = MarkerStyle.markers
1334: (0)           drawStyles = Line2D.drawStyles
1335: (0)           fillStyles = MarkerStyle.fillstyles
```

File 47 - markers.py:

```
1: (0)              r"""
2: (0)              Functions to handle markers; used by the marker functionality of
3: (0)              `~matplotlib.axes.Axes.plot`, `~matplotlib.axes.Axes.scatter`, and
4: (0)              `~matplotlib.axes.Axes.errorbar`.
5: (0)              All possible markers are defined here:
6: (0)              ============================== ======
========================================
7: (0)              marker                         symbol description
8: (0)              ============================== ======
========================================
9: (0)              ``"."``                        |m00|  point
```

```
 10: (0)             ``","``                        |m01|  pixel
 11: (0)             ``"o"``                        |m02|  circle
 12: (0)             ``"v"``                        |m03|  triangle_down
 13: (0)             ``"^"``                        |m04|  triangle_up
 14: (0)             ``"<"``                        |m05|  triangle_left
 15: (0)             ``">"``                        |m06|  triangle_right
 16: (0)             ``"1"``                        |m07|  tri_down
 17: (0)             ``"2"``                        |m08|  tri_up
 18: (0)             ``"3"``                        |m09|  tri_left
 19: (0)             ``"4"``                        |m10|  tri_right
 20: (0)             ``"8"``                        |m11|  octagon
 21: (0)             ``"s"``                        |m12|  square
 22: (0)             ``"p"``                        |m13|  pentagon
 23: (0)             ``"P"``                        |m23|  plus (filled)
 24: (0)             ``"*"``                        |m14|  star
 25: (0)             ``"h"``                        |m15|  hexagon1
 26: (0)             ``"H"``                        |m16|  hexagon2
 27: (0)             ``"+"``                        |m17|  plus
 28: (0)             ``"x"``                        |m18|  x
 29: (0)             ``"X"``                        |m24|  x (filled)
 30: (0)             ``"D"``                        |m19|  diamond
 31: (0)             ``"d"``                        |m20|  thin_diamond
 32: (0)             ``"|"``                        |m21|  vline
 33: (0)             ``"_"``                        |m22|  hline
                       _
 34: (0)             ``0`` (``TICKLEFT``)           |m25|  tickleft
 35: (0)             ``1`` (``TICKRIGHT``)          |m26|  tickright
 36: (0)             ``2`` (``TICKUP``)             |m27|  tickup
 37: (0)             ``3`` (``TICKDOWN``)           |m28|  tickdown
 38: (0)             ``4`` (``CARETLEFT``)          |m29|  caretleft
 39: (0)             ``5`` (``CARETRIGHT``)         |m30|  caretright
 40: (0)             ``6`` (``CARETUP``)            |m31|  caretup
 41: (0)             ``7`` (``CARETDOWN``)          |m32|  caretdown
 42: (0)             ``8`` (``CARETLEFTBASE``)      |m33|  caretleft (centered at base)
 43: (0)             ``9`` (``CARETRIGHTBASE``)     |m34|  caretright (centered at base)
 44: (0)             ``10`` (``CARETUPBASE``)       |m35|  caretup (centered at base)
 45: (0)             ``11`` (``CARETDOWNBASE``)     |m36|  caretdown (centered at base)
 46: (0)             ``"none"`` or ``"None"``              nothing
 47: (0)             ``" "`` or  ``""``                    nothing
 48: (0)             ``"$...$"``                    |m37|  Render the string using mathtext.
 49: (38)                                                  E.g ``"$f$"`` for marker showing the
 50: (38)                                                  letter ``f``.
 51: (0)             ``verts``                             A list of (x, y) pairs used for Path
 52: (38)                                                  vertices. The center of the marker is
 53: (38)                                                  located at (0, 0) and the size is
 54: (38)                                                  normalized, such that the created path
 55: (38)                                                  is encapsulated inside the unit cell.
 56: (0)             ``path``                              A `~matplotlib.path.Path` instance.
 57: (0)             ``(numsides, 0, angle)``              A regular polygon with ``numsides``
 58: (38)                                                  sides, rotated by ``angle``.
 59: (0)             ``(numsides, 1, angle)``              A star-like symbol with ``numsides``
 60: (38)                                                  sides, rotated by ``angle``.
 61: (0)             ``(numsides, 2, angle)``              An asterisk with ``numsides`` sides,
 62: (38)                                                  rotated by ``angle``.
 63: (0)             ============================= ======
 ==========================================
 64: (0)             Note that special symbols can be defined via the
 65: (0)             :ref:`STIX math font <mathtext>`,
 66: (0)             e.g. ``"$\u266B$"``. For an overview over the STIX font symbols refer to the
 67: (0)             `STIX font table <http://www.stixfonts.org/allGlyphs.html>`_.
 68: (0)             Also see the :doc:`/gallery/text_labels_and_annotations/stix_fonts_demo`.
 69: (0)             Integer numbers from ``0`` to ``11`` create lines and triangles. Those are
 70: (0)             equally accessible via capitalized variables, like ``CARETDOWNBASE``.
 71: (0)             Hence the following are equivalent::
 72: (4)                 plt.plot([1, 2, 3], marker=11)
 73: (4)                 plt.plot([1, 2, 3], marker=matplotlib.markers.CARETDOWNBASE)
 74: (0)             Markers join and cap styles can be customized by creating a new instance of
 75: (0)             MarkerStyle.
 76: (0)             A MarkerStyle can also have a custom `~matplotlib.transforms.Transform`
 77: (0)             allowing it to be arbitrarily rotated or offset.
```

```
(0)             Examples showing the use of markers:
(0)             * :doc:`/gallery/lines_bars_and_markers/marker_reference`
(0)             * :doc:`/gallery/lines_bars_and_markers/scatter_star_poly`
(0)             * :doc:`/gallery/lines_bars_and_markers/multivariate_marker_plot`
(0)             .. |m00| image:: /_static/markers/m00.png
(0)             .. |m01| image:: /_static/markers/m01.png
(0)             .. |m02| image:: /_static/markers/m02.png
(0)             .. |m03| image:: /_static/markers/m03.png
(0)             .. |m04| image:: /_static/markers/m04.png
(0)             .. |m05| image:: /_static/markers/m05.png
(0)             .. |m06| image:: /_static/markers/m06.png
(0)             .. |m07| image:: /_static/markers/m07.png
(0)             .. |m08| image:: /_static/markers/m08.png
(0)             .. |m09| image:: /_static/markers/m09.png
(0)             .. |m10| image:: /_static/markers/m10.png
(0)             .. |m11| image:: /_static/markers/m11.png
(0)             .. |m12| image:: /_static/markers/m12.png
(0)             .. |m13| image:: /_static/markers/m13.png
(0)             .. |m14| image:: /_static/markers/m14.png
(0)             .. |m15| image:: /_static/markers/m15.png
(0)             .. |m16| image:: /_static/markers/m16.png
(0)             .. |m17| image:: /_static/markers/m17.png
(0)             .. |m18| image:: /_static/markers/m18.png
(0)             .. |m19| image:: /_static/markers/m19.png
(0)             .. |m20| image:: /_static/markers/m20.png
(0)             .. |m21| image:: /_static/markers/m21.png
(0)             .. |m22| image:: /_static/markers/m22.png
(0)             .. |m23| image:: /_static/markers/m23.png
(0)             .. |m24| image:: /_static/markers/m24.png
(0)             .. |m25| image:: /_static/markers/m25.png
(0)             .. |m26| image:: /_static/markers/m26.png
(0)             .. |m27| image:: /_static/markers/m27.png
(0)             .. |m28| image:: /_static/markers/m28.png
(0)             .. |m29| image:: /_static/markers/m29.png
(0)             .. |m30| image:: /_static/markers/m30.png
(0)             .. |m31| image:: /_static/markers/m31.png
(0)             .. |m32| image:: /_static/markers/m32.png
(0)             .. |m33| image:: /_static/markers/m33.png
(0)             .. |m34| image:: /_static/markers/m34.png
(0)             .. |m35| image:: /_static/markers/m35.png
(0)             .. |m36| image:: /_static/markers/m36.png
(0)             .. |m37| image:: /_static/markers/m37.png
(0)             """
(0)             import copy
(0)             from collections.abc import Sized
(0)             import numpy as np
(0)             import matplotlib as mpl
(0)             from . import _api, cbook
(0)             from .path import Path
(0)             from .transforms import IdentityTransform, Affine2D
(0)             from ._enums import JoinStyle, CapStyle
(0)             (TICKLEFT, TICKRIGHT, TICKUP, TICKDOWN,
(1)              CARETLEFT, CARETRIGHT, CARETUP, CARETDOWN,
(1)              CARETLEFTBASE, CARETRIGHTBASE, CARETUPBASE, CARETDOWNBASE) = range(12)
(0)             _empty_path = Path(np.empty((0, 2)))
(0)             class MarkerStyle:
(4)                 """
(4)                 A class representing marker types.
(4)                 Instances are immutable. If you need to change anything, create a new
(4)                 instance.
(4)                 Attributes
(4)                 ----------
(4)                 markers : dict
(8)                     All known markers.
(4)                 filled_markers : tuple
(8)                     All known filled markers. This is a subset of *markers*.
(4)                 fillstyles : tuple
(8)                     The supported fillstyles.
(4)                 """
```

```
    markers = {
        '.': 'point',
        ',': 'pixel',
        'o': 'circle',
        'v': 'triangle_down',
        '^': 'triangle_up',
        '<': 'triangle_left',
        '>': 'triangle_right',
        '1': 'tri_down',
        '2': 'tri_up',
        '3': 'tri_left',
        '4': 'tri_right',
        '8': 'octagon',
        's': 'square',
        'p': 'pentagon',
        '*': 'star',
        'h': 'hexagon1',
        'H': 'hexagon2',
        '+': 'plus',
        'x': 'x',
        'D': 'diamond',
        'd': 'thin_diamond',
        '|': 'vline',
        '_': 'hline',
        'P': 'plus_filled',
        'X': 'x_filled',
        TICKLEFT: 'tickleft',
        TICKRIGHT: 'tickright',
        TICKUP: 'tickup',
        TICKDOWN: 'tickdown',
        CARETLEFT: 'caretleft',
        CARETRIGHT: 'caretright',
        CARETUP: 'caretup',
        CARETDOWN: 'caretdown',
        CARETLEFTBASE: 'caretleftbase',
        CARETRIGHTBASE: 'caretrightbase',
        CARETUPBASE: 'caretupbase',
        CARETDOWNBASE: 'caretdownbase',
        "None": 'nothing',
        "none": 'nothing',
        ' ': 'nothing',
        '': 'nothing'
    }
    filled_markers = (
        '.', 'o', 'v', '^', '<', '>', '8', 's', 'p', '*', 'h', 'H', 'D', 'd',
        'P', 'X')
    fillstyles = ('full', 'left', 'right', 'bottom', 'top', 'none')
    _half_fillstyles = ('left', 'right', 'bottom', 'top')
    def __init__(self, marker,
                 fillstyle=None, transform=None, capstyle=None,
joinstyle=None):
        """
        Parameters
        ----------
        marker : str, array-like, Path, MarkerStyle
            - Another instance of `MarkerStyle` copies the details of that
*marker*.
            - For other possible marker values, see the module docstring
              `matplotlib.markers`.
        fillstyle : str, default: :rc:`markers.fillstyle`
            One of 'full', 'left', 'right', 'bottom', 'top', 'none'.
        transform : `~matplotlib.transforms.Transform`, optional
            Transform that will be combined with the native transform of the
            marker.
        capstyle : `.CapStyle` or %(CapStyle)s, optional
            Cap style that will override the default cap style of the marker.
        joinstyle : `.JoinStyle` or %(JoinStyle)s, optional
            Join style that will override the default join style of the
marker.
```

```
(8)                     """
(8)                     self._marker_function = None
(8)                     self._user_transform = transform
(8)                     self._user_capstyle = CapStyle(capstyle) if capstyle is not None else
None
(8)                     self._user_joinstyle = JoinStyle(joinstyle) if joinstyle is not None
else None
(8)                     self._set_fillstyle(fillstyle)
(8)                     self._set_marker(marker)
(4)                 def _recache(self):
(8)                     if self._marker_function is None:
(12)                        return
(8)                     self._path = _empty_path
(8)                     self._transform = IdentityTransform()
(8)                     self._alt_path = None
(8)                     self._alt_transform = None
(8)                     self._snap_threshold = None
(8)                     self._joinstyle = JoinStyle.round
(8)                     self._capstyle = self._user_capstyle or CapStyle.butt
(8)                     self._filled = self._fillstyle != 'none'
(8)                     self._marker_function()
(4)                 def __bool__(self):
(8)                     return bool(len(self._path.vertices))
(4)                 def is_filled(self):
(8)                     return self._filled
(4)                 def get_fillstyle(self):
(8)                     return self._fillstyle
(4)                 def _set_fillstyle(self, fillstyle):
(8)                     """
(8)                     Set the fillstyle.
(8)                     Parameters
(8)                     ----------
(8)                     fillstyle : {'full', 'left', 'right', 'bottom', 'top', 'none'}
(12)                        The part of the marker surface that is colored with
(12)                        markerfacecolor.
(8)                     """
(8)                     if fillstyle is None:
(12)                        fillstyle = mpl.rcParams['markers.fillstyle']
(8)                     _api.check_in_list(self.fillstyles, fillstyle=fillstyle)
(8)                     self._fillstyle = fillstyle
(4)                 def get_joinstyle(self):
(8)                     return self._joinstyle.name
(4)                 def get_capstyle(self):
(8)                     return self._capstyle.name
(4)                 def get_marker(self):
(8)                     return self._marker
(4)                 def _set_marker(self, marker):
(8)                     """
(8)                     Set the marker.
(8)                     Parameters
(8)                     ----------
(8)                     marker : str, array-like, Path, MarkerStyle
(12)                        - Another instance of `MarkerStyle` copies the details of that
*marker*.
(12)                        - For other possible marker values see the module docstring
(14)                          `matplotlib.markers`.
(8)                     """
(8)                     if isinstance(marker, str) and cbook.is_math_text(marker):
(12)                        self._marker_function = self._set_mathtext_path
(8)                     elif isinstance(marker, (int, str)) and marker in self.markers:
(12)                        self._marker_function = getattr(self, '_set_' +
self.markers[marker])
(8)                     elif (isinstance(marker, np.ndarray) and marker.ndim == 2 and
(16)                            marker.shape[1] == 2):
(12)                        self._marker_function = self._set_vertices
(8)                     elif isinstance(marker, Path):
(12)                        self._marker_function = self._set_path_marker
(8)                     elif (isinstance(marker, Sized) and len(marker) in (2, 3) and
(16)                            marker[1] in (0, 1, 2)):
```

```
(12)                        self._marker_function = self._set_tuple_marker
(8)                     elif isinstance(marker, MarkerStyle):
(12)                        self.__dict__ = copy.deepcopy(marker.__dict__)
(8)                     else:
(12)                        try:
(16)                            Path(marker)
(16)                            self._marker_function = self._set_vertices
(12)                        except ValueError as err:
(16)                            raise ValueError(
(20)                                f'Unrecognized marker style {marker!r}') from err
(8)                     if not isinstance(marker, MarkerStyle):
(12)                        self._marker = marker
(12)                        self._recache()
(4)                 def get_path(self):
(8)                     """
(8)                     Return a `.Path` for the primary part of the marker.
(8)                     For unfilled markers this is the whole marker, for filled markers,
(8)                     this is the area to be drawn with *markerfacecolor*.
(8)                     """
(8)                     return self._path
(4)                 def get_transform(self):
(8)                     """
(8)                     Return the transform to be applied to the `.Path` from
(8)                     `MarkerStyle.get_path()`.
(8)                     """
(8)                     if self._user_transform is None:
(12)                        return self._transform.frozen()
(8)                     else:
(12)                        return (self._transform + self._user_transform).frozen()
(4)                 def get_alt_path(self):
(8)                     """
(8)                     Return a `.Path` for the alternate part of the marker.
(8)                     For unfilled markers, this is *None*; for filled markers, this is the
(8)                     area to be drawn with *markerfacecoloralt*.
(8)                     """
(8)                     return self._alt_path
(4)                 def get_alt_transform(self):
(8)                     """
(8)                     Return the transform to be applied to the `.Path` from
(8)                     `MarkerStyle.get_alt_path()`.
(8)                     """
(8)                     if self._user_transform is None:
(12)                        return self._alt_transform.frozen()
(8)                     else:
(12)                        return (self._alt_transform + self._user_transform).frozen()
(4)                 def get_snap_threshold(self):
(8)                     return self._snap_threshold
(4)                 def get_user_transform(self):
(8)                     """Return user supplied part of marker transform."""
(8)                     if self._user_transform is not None:
(12)                        return self._user_transform.frozen()
(4)                 def transformed(self, transform):
(8)                     """
(8)                     Return a new version of this marker with the transform applied.
(8)                     Parameters
(8)                     ----------
(8)                     transform : `~matplotlib.transforms.Affine2D`
(12)                        Transform will be combined with current user supplied transform.
(8)                     """
(8)                     new_marker = MarkerStyle(self)
(8)                     if new_marker._user_transform is not None:
(12)                        new_marker._user_transform += transform
(8)                     else:
(12)                        new_marker._user_transform = transform
(8)                     return new_marker
(4)                 def rotated(self, *, deg=None, rad=None):
(8)                     """
(8)                     Return a new version of this marker rotated by specified angle.
(8)                     Parameters
```

```
(8)                     ----------
(8)                     deg : float, optional
(12)                        Rotation angle in degrees.
(8)                     rad : float, optional
(12)                        Rotation angle in radians.
(8)                     .. note:: You must specify exactly one of deg or rad.
(8)                     """
(8)                     if deg is None and rad is None:
(12)                        raise ValueError('One of deg or rad is required')
(8)                     if deg is not None and rad is not None:
(12)                        raise ValueError('Only one of deg and rad can be supplied')
(8)                     new_marker = MarkerStyle(self)
(8)                     if new_marker._user_transform is None:
(12)                        new_marker._user_transform = Affine2D()
(8)                     if deg is not None:
(12)                        new_marker._user_transform.rotate_deg(deg)
(8)                     if rad is not None:
(12)                        new_marker._user_transform.rotate(rad)
(8)                     return new_marker
(4)                 def scaled(self, sx, sy=None):
(8)                     """
(8)                     Return new marker scaled by specified scale factors.
(8)                     If *sy* is not given, the same scale is applied in both the *x*- and
(8)                     *y*-directions.
(8)                     Parameters
(8)                     ----------
(8)                     sx : float
(12)                        *X*-direction scaling factor.
(8)                     sy : float, optional
(12)                        *Y*-direction scaling factor.
(8)                     """
(8)                     if sy is None:
(12)                        sy = sx
(8)                     new_marker = MarkerStyle(self)
(8)                     _transform = new_marker._user_transform or Affine2D()
(8)                     new_marker._user_transform = _transform.scale(sx, sy)
(8)                     return new_marker
(4)                 def _set_nothing(self):
(8)                     self._filled = False
(4)                 def _set_custom_marker(self, path):
(8)                     rescale = np.max(np.abs(path.vertices))  # max of x's and y's.
(8)                     self._transform = Affine2D().scale(0.5 / rescale)
(8)                     self._path = path
(4)                 def _set_path_marker(self):
(8)                     self._set_custom_marker(self._marker)
(4)                 def _set_vertices(self):
(8)                     self._set_custom_marker(Path(self._marker))
(4)                 def _set_tuple_marker(self):
(8)                     marker = self._marker
(8)                     if len(marker) == 2:
(12)                        numsides, rotation = marker[0], 0.0
(8)                     elif len(marker) == 3:
(12)                        numsides, rotation = marker[0], marker[2]
(8)                     symstyle = marker[1]
(8)                     if symstyle == 0:
(12)                        self._path = Path.unit_regular_polygon(numsides)
(12)                        self._joinstyle = self._user_joinstyle or JoinStyle.miter
(8)                     elif symstyle == 1:
(12)                        self._path = Path.unit_regular_star(numsides)
(12)                        self._joinstyle = self._user_joinstyle or JoinStyle.bevel
(8)                     elif symstyle == 2:
(12)                        self._path = Path.unit_regular_asterisk(numsides)
(12)                        self._filled = False
(12)                        self._joinstyle = self._user_joinstyle or JoinStyle.bevel
(8)                     else:
(12)                        raise ValueError(f"Unexpected tuple marker: {marker}")
(8)                     self._transform = Affine2D().scale(0.5).rotate_deg(rotation)
(4)                 def _set_mathtext_path(self):
(8)                     """
```

```
416: (8)                    Draw mathtext markers '$...$' using `.TextPath` object.
417: (8)                    Submitted by tcb
418: (8)                    """
419: (8)                    from matplotlib.text import TextPath
420: (8)                    text = TextPath(xy=(0, 0), s=self.get_marker(),
421: (24)                               usetex=mpl.rcParams['text.usetex'])
422: (8)                    if len(text.vertices) == 0:
423: (12)                       return
424: (8)                    bbox = text.get_extents()
425: (8)                    max_dim = max(bbox.width, bbox.height)
426: (8)                    self._transform = (
427: (12)                       Affine2D()
428: (12)                       .translate(-bbox.xmin + 0.5 * -bbox.width, -bbox.ymin + 0.5 * -
bbox.height)
429: (12)                       .scale(1.0 / max_dim))
430: (8)                    self._path = text
431: (8)                    self._snap = False
432: (4)                def _half_fill(self):
433: (8)                    return self.get_fillstyle() in self._half_fillstyles
434: (4)                def _set_circle(self, size=1.0):
435: (8)                    self._transform = Affine2D().scale(0.5 * size)
436: (8)                    self._snap_threshold = np.inf
437: (8)                    if not self._half_fill():
438: (12)                       self._path = Path.unit_circle()
439: (8)                    else:
440: (12)                       self._path = self._alt_path = Path.unit_circle_righthalf()
441: (12)                       fs = self.get_fillstyle()
442: (12)                       self._transform.rotate_deg(
443: (16)                           {'right': 0, 'top': 90, 'left': 180, 'bottom': 270}[fs])
444: (12)                       self._alt_transform = self._transform.frozen().rotate_deg(180.)
445: (4)                def _set_point(self):
446: (8)                    self._set_circle(size=0.5)
447: (4)                def _set_pixel(self):
448: (8)                    self._path = Path.unit_rectangle()
449: (8)                    self._transform = Affine2D().translate(-0.49999, -0.49999)
450: (8)                    self._snap_threshold = None
451: (4)                _triangle_path = Path._create_closed([[0, 1], [-1, -1], [1, -1]])
452: (4)                _triangle_path_u = Path._create_closed([[0, 1], [-3/5, -1/5], [3/5,
-1/5]])
453: (4)                _triangle_path_d = Path._create_closed(
454: (8)                    [[-3/5, -1/5], [3/5, -1/5], [1, -1], [-1, -1]])
455: (4)                _triangle_path_l = Path._create_closed([[0, 1], [0, -1], [-1, -1]])
456: (4)                _triangle_path_r = Path._create_closed([[0, 1], [0, -1], [1, -1]])
457: (4)                def _set_triangle(self, rot, skip):
458: (8)                    self._transform = Affine2D().scale(0.5).rotate_deg(rot)
459: (8)                    self._snap_threshold = 5.0
460: (8)                    if not self._half_fill():
461: (12)                       self._path = self._triangle_path
462: (8)                    else:
463: (12)                       mpaths = [self._triangle_path_u,
464: (22)                                 self._triangle_path_l,
465: (22)                                 self._triangle_path_d,
466: (22)                                 self._triangle_path_r]
467: (12)                       fs = self.get_fillstyle()
468: (12)                       if fs == 'top':
469: (16)                           self._path = mpaths[(0 + skip) % 4]
470: (16)                           self._alt_path = mpaths[(2 + skip) % 4]
471: (12)                       elif fs == 'bottom':
472: (16)                           self._path = mpaths[(2 + skip) % 4]
473: (16)                           self._alt_path = mpaths[(0 + skip) % 4]
474: (12)                       elif fs == 'left':
475: (16)                           self._path = mpaths[(1 + skip) % 4]
476: (16)                           self._alt_path = mpaths[(3 + skip) % 4]
477: (12)                       else:
478: (16)                           self._path = mpaths[(3 + skip) % 4]
479: (16)                           self._alt_path = mpaths[(1 + skip) % 4]
480: (12)                       self._alt_transform = self._transform
481: (8)                    self._joinstyle = self._user_joinstyle or JoinStyle.miter
482: (4)                def _set_triangle_up(self):
```

```
(8)                 return self._set_triangle(0.0, 0)
(4)             def _set_triangle_down(self):
(8)                 return self._set_triangle(180.0, 2)
(4)             def _set_triangle_left(self):
(8)                 return self._set_triangle(90.0, 3)
(4)             def _set_triangle_right(self):
(8)                 return self._set_triangle(270.0, 1)
(4)             def _set_square(self):
(8)                 self._transform = Affine2D().translate(-0.5, -0.5)
(8)                 self._snap_threshold = 2.0
(8)                 if not self._half_fill():
(12)                    self._path = Path.unit_rectangle()
(8)                 else:
(12)                    self._path = Path([[0.0, 0.0], [1.0, 0.0], [1.0, 0.5],
(31)                                       [0.0, 0.5], [0.0, 0.0]])
(12)                    self._alt_path = Path([[0.0, 0.5], [1.0, 0.5], [1.0, 1.0],
(35)                                           [0.0, 1.0], [0.0, 0.5]])
(12)                    fs = self.get_fillstyle()
(12)                    rotate = {'bottom': 0, 'right': 90, 'top': 180, 'left': 270}[fs]
(12)                    self._transform.rotate_deg(rotate)
(12)                    self._alt_transform = self._transform
(8)                 self._joinstyle = self._user_joinstyle or JoinStyle.miter
(4)             def _set_diamond(self):
(8)                 self._transform = Affine2D().translate(-0.5, -0.5).rotate_deg(45)
(8)                 self._snap_threshold = 5.0
(8)                 if not self._half_fill():
(12)                    self._path = Path.unit_rectangle()
(8)                 else:
(12)                    self._path = Path([[0, 0], [1, 0], [1, 1], [0, 0]])
(12)                    self._alt_path = Path([[0, 0], [0, 1], [1, 1], [0, 0]])
(12)                    fs = self.get_fillstyle()
(12)                    rotate = {'right': 0, 'top': 90, 'left': 180, 'bottom': 270}[fs]
(12)                    self._transform.rotate_deg(rotate)
(12)                    self._alt_transform = self._transform
(8)                 self._joinstyle = self._user_joinstyle or JoinStyle.miter
(4)             def _set_thin_diamond(self):
(8)                 self._set_diamond()
(8)                 self._transform.scale(0.6, 1.0)
(4)             def _set_pentagon(self):
(8)                 self._transform = Affine2D().scale(0.5)
(8)                 self._snap_threshold = 5.0
(8)                 polypath = Path.unit_regular_polygon(5)
(8)                 if not self._half_fill():
(12)                    self._path = polypath
(8)                 else:
(12)                    verts = polypath.vertices
(12)                    y = (1 + np.sqrt(5)) / 4.
(12)                    top = Path(verts[[0, 1, 4, 0]])
(12)                    bottom = Path(verts[[1, 2, 3, 4, 1]])
(12)                    left = Path([verts[0], verts[1], verts[2], [0, -y], verts[0]])
(12)                    right = Path([verts[0], verts[4], verts[3], [0, -y], verts[0]])
(12)                    self._path, self._alt_path = {
(16)                        'top': (top, bottom), 'bottom': (bottom, top),
(16)                        'left': (left, right), 'right': (right, left),
(12)                    }[self.get_fillstyle()]
(12)                    self._alt_transform = self._transform
(8)                 self._joinstyle = self._user_joinstyle or JoinStyle.miter
(4)             def _set_star(self):
(8)                 self._transform = Affine2D().scale(0.5)
(8)                 self._snap_threshold = 5.0
(8)                 polypath = Path.unit_regular_star(5, innerCircle=0.381966)
(8)                 if not self._half_fill():
(12)                    self._path = polypath
(8)                 else:
(12)                    verts = polypath.vertices
(12)                    top = Path(np.concatenate([verts[0:4], verts[7:10], verts[0:1]]))
(12)                    bottom = Path(np.concatenate([verts[3:8], verts[3:4]]))
(12)                    left = Path(np.concatenate([verts[0:6], verts[0:1]]))
(12)                    right = Path(np.concatenate([verts[0:1], verts[5:10],
```

```
verts[0:1]]))
552: (12)                        self._path, self._alt_path = {
553: (16)                            'top': (top, bottom), 'bottom': (bottom, top),
554: (16)                            'left': (left, right), 'right': (right, left),
555: (12)                        }[self.get_fillstyle()]
556: (12)                        self._alt_transform = self._transform
557: (8)                    self._joinstyle = self._user_joinstyle or JoinStyle.bevel
558: (4)                def _set_hexagon1(self):
559: (8)                    self._transform = Affine2D().scale(0.5)
560: (8)                    self._snap_threshold = None
561: (8)                    polypath = Path.unit_regular_polygon(6)
562: (8)                    if not self._half_fill():
563: (12)                        self._path = polypath
564: (8)                    else:
565: (12)                        verts = polypath.vertices
566: (12)                        x = np.abs(np.cos(5 * np.pi / 6.))
567: (12)                        top = Path(np.concatenate([[(-x, 0)], verts[[1, 0, 5]], [(x,
0)]]))
568: (12)                        bottom = Path(np.concatenate([[(-x, 0)], verts[2:5], [(x, 0)]]))
569: (12)                        left = Path(verts[0:4])
570: (12)                        right = Path(verts[[0, 5, 4, 3]])
571: (12)                        self._path, self._alt_path = {
572: (16)                            'top': (top, bottom), 'bottom': (bottom, top),
573: (16)                            'left': (left, right), 'right': (right, left),
574: (12)                        }[self.get_fillstyle()]
575: (12)                        self._alt_transform = self._transform
576: (8)                    self._joinstyle = self._user_joinstyle or JoinStyle.miter
577: (4)                def _set_hexagon2(self):
578: (8)                    self._transform = Affine2D().scale(0.5).rotate_deg(30)
579: (8)                    self._snap_threshold = None
580: (8)                    polypath = Path.unit_regular_polygon(6)
581: (8)                    if not self._half_fill():
582: (12)                        self._path = polypath
583: (8)                    else:
584: (12)                        verts = polypath.vertices
585: (12)                        x, y = np.sqrt(3) / 4, 3 / 4.
586: (12)                        top = Path(verts[[1, 0, 5, 4, 1]])
587: (12)                        bottom = Path(verts[1:5])
588: (12)                        left = Path(np.concatenate([
589: (16)                            [(x, y)], verts[:3], [(-x, -y), (x, y)]]))
590: (12)                        right = Path(np.concatenate([
591: (16)                            [(x, y)], verts[5:2:-1], [(-x, -y)]]))
592: (12)                        self._path, self._alt_path = {
593: (16)                            'top': (top, bottom), 'bottom': (bottom, top),
594: (16)                            'left': (left, right), 'right': (right, left),
595: (12)                        }[self.get_fillstyle()]
596: (12)                        self._alt_transform = self._transform
597: (8)                    self._joinstyle = self._user_joinstyle or JoinStyle.miter
598: (4)                def _set_octagon(self):
599: (8)                    self._transform = Affine2D().scale(0.5)
600: (8)                    self._snap_threshold = 5.0
601: (8)                    polypath = Path.unit_regular_polygon(8)
602: (8)                    if not self._half_fill():
603: (12)                        self._transform.rotate_deg(22.5)
604: (12)                        self._path = polypath
605: (8)                    else:
606: (12)                        x = np.sqrt(2.) / 4.
607: (12)                        self._path = self._alt_path = Path(
608: (16)                            [[0, -1], [0, 1], [-x, 1], [-1, x],
609: (17)                             [-1, -x], [-x, -1], [0, -1]])
610: (12)                        fs = self.get_fillstyle()
611: (12)                        self._transform.rotate_deg(
612: (16)                            {'left': 0, 'bottom': 90, 'right': 180, 'top': 270}[fs])
613: (12)                        self._alt_transform = self._transform.frozen().rotate_deg(180.0)
614: (8)                    self._joinstyle = self._user_joinstyle or JoinStyle.miter
615: (4)                _line_marker_path = Path([[0.0, -1.0], [0.0, 1.0]])
616: (4)                def _set_vline(self):
617: (8)                    self._transform = Affine2D().scale(0.5)
618: (8)                    self._snap_threshold = 1.0
```

```
(8)                 self._filled = False
(8)                 self._path = self._line_marker_path
(4)             def _set_hline(self):
(8)                 self._set_vline()
(8)                 self._transform = self._transform.rotate_deg(90)
(4)             _tickhoriz_path = Path([[0.0, 0.0], [1.0, 0.0]])
(4)             def _set_tickleft(self):
(8)                 self._transform = Affine2D().scale(-1.0, 1.0)
(8)                 self._snap_threshold = 1.0
(8)                 self._filled = False
(8)                 self._path = self._tickhoriz_path
(4)             def _set_tickright(self):
(8)                 self._transform = Affine2D().scale(1.0, 1.0)
(8)                 self._snap_threshold = 1.0
(8)                 self._filled = False
(8)                 self._path = self._tickhoriz_path
(4)             _tickvert_path = Path([[-0.0, 0.0], [-0.0, 1.0]])
(4)             def _set_tickup(self):
(8)                 self._transform = Affine2D().scale(1.0, 1.0)
(8)                 self._snap_threshold = 1.0
(8)                 self._filled = False
(8)                 self._path = self._tickvert_path
(4)             def _set_tickdown(self):
(8)                 self._transform = Affine2D().scale(1.0, -1.0)
(8)                 self._snap_threshold = 1.0
(8)                 self._filled = False
(8)                 self._path = self._tickvert_path
(4)             _tri_path = Path([[0.0, 0.0], [0.0, -1.0],
(22)                              [0.0, 0.0], [0.8, 0.5],
(22)                              [0.0, 0.0], [-0.8, 0.5]],
(21)                             [Path.MOVETO, Path.LINETO,
(22)                              Path.MOVETO, Path.LINETO,
(22)                              Path.MOVETO, Path.LINETO])
(4)             def _set_tri_down(self):
(8)                 self._transform = Affine2D().scale(0.5)
(8)                 self._snap_threshold = 5.0
(8)                 self._filled = False
(8)                 self._path = self._tri_path
(4)             def _set_tri_up(self):
(8)                 self._set_tri_down()
(8)                 self._transform = self._transform.rotate_deg(180)
(4)             def _set_tri_left(self):
(8)                 self._set_tri_down()
(8)                 self._transform = self._transform.rotate_deg(270)
(4)             def _set_tri_right(self):
(8)                 self._set_tri_down()
(8)                 self._transform = self._transform.rotate_deg(90)
(4)             _caret_path = Path([[-1.0, 1.5], [0.0, 0.0], [1.0, 1.5]])
(4)             def _set_caretdown(self):
(8)                 self._transform = Affine2D().scale(0.5)
(8)                 self._snap_threshold = 3.0
(8)                 self._filled = False
(8)                 self._path = self._caret_path
(8)                 self._joinstyle = self._user_joinstyle or JoinStyle.miter
(4)             def _set_caretup(self):
(8)                 self._set_caretdown()
(8)                 self._transform = self._transform.rotate_deg(180)
(4)             def _set_caretleft(self):
(8)                 self._set_caretdown()
(8)                 self._transform = self._transform.rotate_deg(270)
(4)             def _set_caretright(self):
(8)                 self._set_caretdown()
(8)                 self._transform = self._transform.rotate_deg(90)
(4)             _caret_path_base = Path([[-1.0, 0.0], [0.0, -1.5], [1.0, 0]])
(4)             def _set_caretdownbase(self):
(8)                 self._set_caretdown()
(8)                 self._path = self._caret_path_base
(4)             def _set_caretupbase(self):
(8)                 self._set_caretdownbase()
```

```
688: (8)                    self._transform = self._transform.rotate_deg(180)
689: (4)                def _set_caretleftbase(self):
690: (8)                    self._set_caretdownbase()
691: (8)                    self._transform = self._transform.rotate_deg(270)
692: (4)                def _set_caretrightbase(self):
693: (8)                    self._set_caretdownbase()
694: (8)                    self._transform = self._transform.rotate_deg(90)
695: (4)                _plus_path = Path([[-1.0, 0.0], [1.0, 0.0],
696: (23)                                  [0.0, -1.0], [0.0, 1.0]],
697: (22)                                 [Path.MOVETO, Path.LINETO,
698: (23)                                  Path.MOVETO, Path.LINETO])
699: (4)                def _set_plus(self):
700: (8)                    self._transform = Affine2D().scale(0.5)
701: (8)                    self._snap_threshold = 1.0
702: (8)                    self._filled = False
703: (8)                    self._path = self._plus_path
704: (4)                _x_path = Path([[-1.0, -1.0], [1.0, 1.0],
705: (20)                               [-1.0, 1.0], [1.0, -1.0]],
706: (19)                              [Path.MOVETO, Path.LINETO,
707: (20)                               Path.MOVETO, Path.LINETO])
708: (4)                def _set_x(self):
709: (8)                    self._transform = Affine2D().scale(0.5)
710: (8)                    self._snap_threshold = 3.0
711: (8)                    self._filled = False
712: (8)                    self._path = self._x_path
713: (4)                _plus_filled_path = Path._create_closed(np.array([
714: (8)                    (-1, -3), (+1, -3), (+1, -1), (+3, -1), (+3, +1), (+1, +1),
715: (8)                    (+1, +3), (-1, +3), (-1, +1), (-3, +1), (-3, -1), (-1, -1)]) / 6)
716: (4)                _plus_filled_path_t = Path._create_closed(np.array([
717: (8)                    (+3, 0), (+3, +1), (+1, +1), (+1, +3),
718: (8)                    (-1, +3), (-1, +1), (-3, +1), (-3, 0)]) / 6)
719: (4)                def _set_plus_filled(self):
720: (8)                    self._transform = Affine2D()
721: (8)                    self._snap_threshold = 5.0
722: (8)                    self._joinstyle = self._user_joinstyle or JoinStyle.miter
723: (8)                    if not self._half_fill():
724: (12)                       self._path = self._plus_filled_path
725: (8)                    else:
726: (12)                       self._path = self._alt_path = self._plus_filled_path_t
727: (12)                       fs = self.get_fillstyle()
728: (12)                       self._transform.rotate_deg(
729: (16)                           {'top': 0, 'left': 90, 'bottom': 180, 'right': 270}[fs])
730: (12)                       self._alt_transform = self._transform.frozen().rotate_deg(180)
731: (4)                _x_filled_path = Path._create_closed(np.array([
732: (8)                    (-1, -2), (0, -1), (+1, -2), (+2, -1), (+1, 0), (+2, +1),
733: (8)                    (+1, +2), (0, +1), (-1, +2), (-2, +1), (-1, 0), (-2, -1)]) / 4)
734: (4)                _x_filled_path_t = Path._create_closed(np.array([
735: (8)                    (+1, 0), (+2, +1), (+1, +2), (0, +1),
736: (8)                    (-1, +2), (-2, +1), (-1, 0)]) / 4)
737: (4)                def _set_x_filled(self):
738: (8)                    self._transform = Affine2D()
739: (8)                    self._snap_threshold = 5.0
740: (8)                    self._joinstyle = self._user_joinstyle or JoinStyle.miter
741: (8)                    if not self._half_fill():
742: (12)                       self._path = self._x_filled_path
743: (8)                    else:
744: (12)                       self._path = self._alt_path = self._x_filled_path_t
745: (12)                       fs = self.get_fillstyle()
746: (12)                       self._transform.rotate_deg(
747: (16)                           {'top': 0, 'left': 90, 'bottom': 180, 'right': 270}[fs])
748: (12)                       self._alt_transform = self._transform.frozen().rotate_deg(180)
```

File 48 - mathtext.py:

```
1: (0)              r"""
2: (0)              A module for parsing a subset of the TeX math syntax and rendering it to a
3: (0)              Matplotlib backend.
```

```
4: (0)              For a tutorial of its usage, see :ref:`mathtext`.  This
5: (0)              document is primarily concerned with implementation details.
6: (0)              The module uses pyparsing_ to parse the TeX expression.
7: (0)              .. _pyparsing: https://pypi.org/project/pyparsing/
8: (0)              The Bakoma distribution of the TeX Computer Modern fonts, and STIX
9: (0)              fonts are supported.  There is experimental support for using
10: (0)             arbitrary fonts, but results may vary without proper tweaking and
11: (0)             metrics for those fonts.
12: (0)             """
13: (0)             import functools
14: (0)             import logging
15: (0)             import matplotlib as mpl
16: (0)             from matplotlib import _api, _mathtext
17: (0)             from matplotlib.ft2font import LOAD_NO_HINTING
18: (0)             from matplotlib.font_manager import FontProperties
19: (0)             from ._mathtext import (  # noqa: reexported API
20: (4)                 RasterParse, VectorParse, get_unicode_index)
21: (0)             _log = logging.getLogger(__name__)
22: (0)             get_unicode_index.__module__ = __name__
23: (0)             class MathTextParser:
24: (4)                 _parser = None
25: (4)                 _font_type_mapping = {
26: (8)                     'cm':          _mathtext.BakomaFonts,
27: (8)                     'dejavuserif': _mathtext.DejaVuSerifFonts,
28: (8)                     'dejavusans':  _mathtext.DejaVuSansFonts,
29: (8)                     'stix':        _mathtext.StixFonts,
30: (8)                     'stixsans':    _mathtext.StixSansFonts,
31: (8)                     'custom':      _mathtext.UnicodeFonts,
32: (4)                 }
33: (4)                 def __init__(self, output):
34: (8)                     """
35: (8)                     Create a MathTextParser for the given backend *output*.
36: (8)                     Parameters
37: (8)                     ----------
38: (8)                     output : {"path", "agg"}
39: (12)                        Whether to return a `VectorParse` ("path") or a
40: (12)                        `RasterParse` ("agg", or its synonym "macosx").
41: (8)                     """
42: (8)                     self._output_type = _api.check_getitem(
43: (12)                        {"path": "vector", "agg": "raster", "macosx": "raster"},
44: (12)                        output=output.lower())
45: (4)                 def parse(self, s, dpi=72, prop=None, *, antialiased=None):
46: (8)                     """
47: (8)                     Parse the given math expression *s* at the given *dpi*.  If *prop* is
48: (8)                     provided, it is a `.FontProperties` object specifying the "default"
49: (8)                     font to use in the math expression, used for all non-math text.
50: (8)                     The results are cached, so multiple calls to `parse`
51: (8)                     with the same expression should be fast.
52: (8)                     Depending on the *output* type, this returns either a `VectorParse` or
53: (8)                     a `RasterParse`.
54: (8)                     """
55: (8)                     prop = prop.copy() if prop is not None else None
56: (8)                     antialiased = mpl._val_or_rc(antialiased, 'text.antialiased')
57: (8)                     return self._parse_cached(s, dpi, prop, antialiased)
58: (4)                 @functools.lru_cache(50)
59: (4)                 def _parse_cached(self, s, dpi, prop, antialiased):
60: (8)                     from matplotlib.backends import backend_agg
61: (8)                     if prop is None:
62: (12)                        prop = FontProperties()
63: (8)                     fontset_class = _api.check_getitem(
64: (12)                        self._font_type_mapping, fontset=prop.get_math_fontfamily())
65: (8)                     load_glyph_flags = {
66: (12)                        "vector": LOAD_NO_HINTING,
67: (12)                        "raster": backend_agg.get_hinting_flag(),
68: (8)                     }[self._output_type]
69: (8)                     fontset = fontset_class(prop, load_glyph_flags)
70: (8)                     fontsize = prop.get_size_in_points()
71: (8)                     if self._parser is None:  # Cache the parser globally.
72: (12)                        self.__class__._parser = _mathtext.Parser()
```

```
73: (8)                     box = self._parser.parse(s, fontset, fontsize, dpi)
74: (8)                     output = _mathtext.ship(box)
75: (8)                     if self._output_type == "vector":
76: (12)                        return output.to_vector()
77: (8)                     elif self._output_type == "raster":
78: (12)                        return output.to_raster(antialiased=antialiased)
79: (0)             def math_to_image(s, filename_or_obj, prop=None, dpi=None, format=None,
80: (18)                              *, color=None):
81: (4)                 """
82: (4)                 Given a math expression, renders it in a closely-clipped bounding
83: (4)                 box to an image file.
84: (4)                 Parameters
85: (4)                 ----------
86: (4)                 s : str
87: (8)                     A math expression.  The math portion must be enclosed in dollar signs.
88: (4)                 filename_or_obj : str or path-like or file-like
89: (8)                     Where to write the image data.
90: (4)                 prop : `.FontProperties`, optional
91: (8)                     The size and style of the text.
92: (4)                 dpi : float, optional
93: (8)                     The output dpi.  If not set, the dpi is determined as for
94: (8)                     `.Figure.savefig`.
95: (4)                 format : str, optional
96: (8)                     The output format, e.g., 'svg', 'pdf', 'ps' or 'png'.  If not set, the
97: (8)                     format is determined as for `.Figure.savefig`.
98: (4)                 color : str, optional
99: (8)                     Foreground color, defaults to :rc:`text.color`.
100: (4)                """
101: (4)                from matplotlib import figure
102: (4)                parser = MathTextParser('path')
103: (4)                width, height, depth, _, _ = parser.parse(s, dpi=72, prop=prop)
104: (4)                fig = figure.Figure(figsize=(width / 72.0, height / 72.0))
105: (4)                fig.text(0, depth/height, s, fontproperties=prop, color=color)
106: (4)                fig.savefig(filename_or_obj, dpi=dpi, format=format)
107: (4)                return depth
```

File 49 - mlab.py:

```
1: (0)              """
2: (0)              Numerical Python functions written for compatibility with MATLAB
3: (0)              commands with the same names. Most numerical Python functions can be found in
4: (0)              the `NumPy`_ and `SciPy`_ libraries. What remains here is code for performing
5: (0)              spectral computations and kernel density estimations.
6: (0)              .. _NumPy: https://numpy.org
7: (0)              .. _SciPy: https://www.scipy.org
8: (0)              Spectral functions
9: (0)              ------------------
10: (0)             `cohere`
11: (4)                 Coherence (normalized cross spectral density)
12: (0)             `csd`
13: (4)                 Cross spectral density using Welch's average periodogram
14: (0)             `detrend`
15: (4)                 Remove the mean or best fit line from an array
16: (0)             `psd`
17: (4)                 Power spectral density using Welch's average periodogram
18: (0)             `specgram`
19: (4)                 Spectrogram (spectrum over segments of time)
20: (0)             `complex_spectrum`
21: (4)                 Return the complex-valued frequency spectrum of a signal
22: (0)             `magnitude_spectrum`
23: (4)                 Return the magnitude of the frequency spectrum of a signal
24: (0)             `angle_spectrum`
25: (4)                 Return the angle (wrapped phase) of the frequency spectrum of a signal
26: (0)             `phase_spectrum`
27: (4)                 Return the phase (unwrapped angle) of the frequency spectrum of a signal
28: (0)             `detrend_mean`
29: (4)                 Remove the mean from a line.
```

```
(0)             `detrend_linear`
(4)                 Remove the best fit line from a line.
(0)             `detrend_none`
(4)                 Return the original line.
(0)             """
(0)             import functools
(0)             from numbers import Number
(0)             import numpy as np
(0)             from matplotlib import _api, _docstring, cbook
(0)             def window_hanning(x):
(4)                 """
(4)                 Return *x* times the Hanning (or Hann) window of len(*x*).
(4)                 See Also
(4)                 --------
(4)                 window_none : Another window algorithm.
(4)                 """
(4)                 return np.hanning(len(x))*x
(0)             def window_none(x):
(4)                 """
(4)                 No window function; simply return *x*.
(4)                 See Also
(4)                 --------
(4)                 window_hanning : Another window algorithm.
(4)                 """
(4)                 return x
(0)             def detrend(x, key=None, axis=None):
(4)                 """
(4)                 Return *x* with its trend removed.
(4)                 Parameters
(4)                 ----------
(4)                 x : array or sequence
(8)                     Array or sequence containing the data.
(4)                 key : {'default', 'constant', 'mean', 'linear', 'none'} or function
(8)                     The detrending algorithm to use. 'default', 'mean', and 'constant' are
(8)                     the same as `detrend_mean`. 'linear' is the same as `detrend_linear`.
(8)                     'none' is the same as `detrend_none`. The default is 'mean'. See the
(8)                     corresponding functions for more details regarding the algorithms. Can
(8)                     also be a function that carries out the detrend operation.
(4)                 axis : int
(8)                     The axis along which to do the detrending.
(4)                 See Also
(4)                 --------
(4)                 detrend_mean : Implementation of the 'mean' algorithm.
(4)                 detrend_linear : Implementation of the 'linear' algorithm.
(4)                 detrend_none : Implementation of the 'none' algorithm.
(4)                 """
(4)                 if key is None or key in ['constant', 'mean', 'default']:
(8)                     return detrend(x, key=detrend_mean, axis=axis)
(4)                 elif key == 'linear':
(8)                     return detrend(x, key=detrend_linear, axis=axis)
(4)                 elif key == 'none':
(8)                     return detrend(x, key=detrend_none, axis=axis)
(4)                 elif callable(key):
(8)                     x = np.asarray(x)
(8)                     if axis is not None and axis + 1 > x.ndim:
(12)                        raise ValueError(f'axis(={axis}) out of bounds')
(8)                     if (axis is None and x.ndim == 0) or (not axis and x.ndim == 1):
(12)                        return key(x)
(8)                     try:
(12)                        return key(x, axis=axis)
(8)                     except TypeError:
(12)                        return np.apply_along_axis(key, axis=axis, arr=x)
(4)                 else:
(8)                     raise ValueError(
(12)                        f"Unknown value for key: {key!r}, must be one of: 'default', "
(12)                        f"'constant', 'mean', 'linear', or a function")
(0)             def detrend_mean(x, axis=None):
(4)                 """
(4)                 Return *x* minus the mean(*x*).
```

```
(4)                 Parameters
(4)                 ----------
(4)                 x : array or sequence
(8)                     Array or sequence containing the data
(8)                     Can have any dimensionality
(4)                 axis : int
(8)                     The axis along which to take the mean.  See `numpy.mean` for a
(8)                     description of this argument.
(4)                 See Also
(4)                 --------
(4)                 detrend_linear : Another detrend algorithm.
(4)                 detrend_none : Another detrend algorithm.
(4)                 detrend : A wrapper around all the detrend algorithms.
(4)                 """
(4)                 x = np.asarray(x)
(4)                 if axis is not None and axis+1 > x.ndim:
(8)                     raise ValueError('axis(=%s) out of bounds' % axis)
(4)                 return x - x.mean(axis, keepdims=True)
(0)             def detrend_none(x, axis=None):
(4)                 """
(4)                 Return *x*: no detrending.
(4)                 Parameters
(4)                 ----------
(4)                 x : any object
(8)                     An object containing the data
(4)                 axis : int
(8)                     This parameter is ignored.
(8)                     It is included for compatibility with detrend_mean
(4)                 See Also
(4)                 --------
(4)                 detrend_mean : Another detrend algorithm.
(4)                 detrend_linear : Another detrend algorithm.
(4)                 detrend : A wrapper around all the detrend algorithms.
(4)                 """
(4)                 return x
(0)             def detrend_linear(y):
(4)                 """
(4)                 Return *x* minus best fit line; 'linear' detrending.
(4)                 Parameters
(4)                 ----------
(4)                 y : 0-D or 1-D array or sequence
(8)                     Array or sequence containing the data
(4)                 See Also
(4)                 --------
(4)                 detrend_mean : Another detrend algorithm.
(4)                 detrend_none : Another detrend algorithm.
(4)                 detrend : A wrapper around all the detrend algorithms.
(4)                 """
(4)                 y = np.asarray(y)
(4)                 if y.ndim > 1:
(8)                     raise ValueError('y cannot have ndim > 1')
(4)                 if not y.ndim:
(8)                     return np.array(0., dtype=y.dtype)
(4)                 x = np.arange(y.size, dtype=float)
(4)                 C = np.cov(x, y, bias=1)
(4)                 b = C[0, 1]/C[0, 0]
(4)                 a = y.mean() - b*x.mean()
(4)                 return y - (b*x + a)
(0)             def _spectral_helper(x, y=None, NFFT=None, Fs=None, detrend_func=None,
(21)                                 window=None, noverlap=None, pad_to=None,
(21)                                 sides=None, scale_by_freq=None, mode=None):
(4)                 """
(4)                 Private helper implementing the common parts between the psd, csd,
(4)                 spectrogram and complex, magnitude, angle, and phase spectrums.
(4)                 """
(4)                 if y is None:
(8)                     same_data = True
(4)                 else:
(8)                     same_data = y is x
```

```
(4)     if Fs is None:
(8)         Fs = 2
(4)     if noverlap is None:
(8)         noverlap = 0
(4)     if detrend_func is None:
(8)         detrend_func = detrend_none
(4)     if window is None:
(8)         window = window_hanning
(4)     if NFFT is None:
(8)         NFFT = 256
(4)     if noverlap >= NFFT:
(8)         raise ValueError('noverlap must be less than NFFT')
(4)     if mode is None or mode == 'default':
(8)         mode = 'psd'
(4)     _api.check_in_list(
(8)         ['default', 'psd', 'complex', 'magnitude', 'angle', 'phase'],
(8)         mode=mode)
(4)     if not same_data and mode != 'psd':
(8)         raise ValueError("x and y must be equal if mode is not 'psd'")
(4)     x = np.asarray(x)
(4)     if not same_data:
(8)         y = np.asarray(y)
(4)     if sides is None or sides == 'default':
(8)         if np.iscomplexobj(x):
(12)            sides = 'twosided'
(8)         else:
(12)            sides = 'onesided'
(4)     _api.check_in_list(['default', 'onesided', 'twosided'], sides=sides)
(4)     if len(x) < NFFT:
(8)         n = len(x)
(8)         x = np.resize(x, NFFT)
(8)         x[n:] = 0
(4)     if not same_data and len(y) < NFFT:
(8)         n = len(y)
(8)         y = np.resize(y, NFFT)
(8)         y[n:] = 0
(4)     if pad_to is None:
(8)         pad_to = NFFT
(4)     if mode != 'psd':
(8)         scale_by_freq = False
(4)     elif scale_by_freq is None:
(8)         scale_by_freq = True
(4)     if sides == 'twosided':
(8)         numFreqs = pad_to
(8)         if pad_to % 2:
(12)            freqcenter = (pad_to - 1)//2 + 1
(8)         else:
(12)            freqcenter = pad_to//2
(8)         scaling_factor = 1.
(4)     elif sides == 'onesided':
(8)         if pad_to % 2:
(12)            numFreqs = (pad_to + 1)//2
(8)         else:
(12)            numFreqs = pad_to//2 + 1
(8)         scaling_factor = 2.
(4)     if not np.iterable(window):
(8)         window = window(np.ones(NFFT, x.dtype))
(4)     if len(window) != NFFT:
(8)         raise ValueError(
(12)            "The window length must match the data's first dimension")
(4)     result = np.lib.stride_tricks.sliding_window_view(
(8)         x, NFFT, axis=0)[::NFFT - noverlap].T
(4)     result = detrend(result, detrend_func, axis=0)
(4)     result = result * window.reshape((-1, 1))
(4)     result = np.fft.fft(result, n=pad_to, axis=0)[:numFreqs, :]
(4)     freqs = np.fft.fftfreq(pad_to, 1/Fs)[:numFreqs]
(4)     if not same_data:
(8)         resultY = np.lib.stride_tricks.sliding_window_view(
(12)            y, NFFT, axis=0)[::NFFT - noverlap].T
```

```
237: (8)                    resultY = detrend(resultY, detrend_func, axis=0)
238: (8)                    resultY = resultY * window.reshape((-1, 1))
239: (8)                    resultY = np.fft.fft(resultY, n=pad_to, axis=0)[:numFreqs, :]
240: (8)                    result = np.conj(result) * resultY
241: (4)                elif mode == 'psd':
242: (8)                    result = np.conj(result) * result
243: (4)                elif mode == 'magnitude':
244: (8)                    result = np.abs(result) / window.sum()
245: (4)                elif mode == 'angle' or mode == 'phase':
246: (8)                    result = np.angle(result)
247: (4)                elif mode == 'complex':
248: (8)                    result /= window.sum()
249: (4)                if mode == 'psd':
250: (8)                    if not NFFT % 2:
251: (12)                       slc = slice(1, -1, None)
252: (8)                    else:
253: (12)                       slc = slice(1, None, None)
254: (8)                    result[slc] *= scaling_factor
255: (8)                    if scale_by_freq:
256: (12)                       result /= Fs
257: (12)                       result /= (window**2).sum()
258: (8)                    else:
259: (12)                       result /= window.sum()**2
260: (4)                t = np.arange(NFFT/2, len(x) - NFFT/2 + 1, NFFT - noverlap)/Fs
261: (4)                if sides == 'twosided':
262: (8)                    freqs = np.roll(freqs, -freqcenter, axis=0)
263: (8)                    result = np.roll(result, -freqcenter, axis=0)
264: (4)                elif not pad_to % 2:
265: (8)                    freqs[-1] *= -1
266: (4)                if mode == 'phase':
267: (8)                    result = np.unwrap(result, axis=0)
268: (4)                return result, freqs, t
269: (0)            def _single_spectrum_helper(
270: (8)                    mode, x, Fs=None, window=None, pad_to=None, sides=None):
271: (4)                """
272: (4)                Private helper implementing the commonality between the complex,
magnitude,
273: (4)                angle, and phase spectrums.
274: (4)                """
275: (4)                _api.check_in_list(['complex', 'magnitude', 'angle', 'phase'], mode=mode)
276: (4)                if pad_to is None:
277: (8)                    pad_to = len(x)
278: (4)                spec, freqs, _ = _spectral_helper(x=x, y=None, NFFT=len(x), Fs=Fs,
279: (38)                                                 detrend_func=detrend_none,
window=window,
280: (38)                                                 noverlap=0, pad_to=pad_to,
281: (38)                                                 sides=sides,
282: (38)                                                 scale_by_freq=False,
283: (38)                                                 mode=mode)
284: (4)                if mode != 'complex':
285: (8)                    spec = spec.real
286: (4)                if spec.ndim == 2 and spec.shape[1] == 1:
287: (8)                    spec = spec[:, 0]
288: (4)                return spec, freqs
289: (0)            _docstring.interpd.update(
290: (4)                Spectral="""\
291: (0)            Fs : float, default: 2
292: (4)                The sampling frequency (samples per time unit).  It is used to calculate
293: (4)                the Fourier frequencies, *freqs*, in cycles per time unit.
294: (0)            window : callable or ndarray, default: `.window_hanning`
295: (4)                A function or a vector of length *NFFT*.  To create window vectors see
296: (4)                `.window_hanning`, `.window_none`, `numpy.blackman`, `numpy.hamming`,
297: (4)                `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc.  If a
298: (4)                function is passed as the argument, it must take a data segment as an
299: (4)                argument and return the windowed version of the segment.
300: (0)            sides : {'default', 'onesided', 'twosided'}, optional
301: (4)                Which sides of the spectrum to return. 'default' is one-sided for real
302: (4)                data and two-sided for complex data. 'onesided' forces the return of a
303: (4)                one-sided spectrum, while 'twosided' forces two-sided.""",
```

```
(4)                 Single_Spectrum="""\
(0)             pad_to : int, optional
(4)                 The number of points to which the data segment is padded when performing
(4)                 the FFT.  While not increasing the actual resolution of the spectrum (the
(4)                 minimum distance between resolvable peaks), this can give more points in
(4)                 the plot, allowing for more detail. This corresponds to the *n* parameter
(4)                 in the call to `~numpy.fft.fft`.  The default is None, which sets *pad_to*
(4)                 equal to the length of the input signal (i.e. no padding)."""",
(4)                 PSD="""\
(0)             pad_to : int, optional
(4)                 The number of points to which the data segment is padded when performing
(4)                 the FFT.  This can be different from *NFFT*, which specifies the number
(4)                 of data points used.  While not increasing the actual resolution of the
(4)                 spectrum (the minimum distance between resolvable peaks), this can give
(4)                 more points in the plot, allowing for more detail. This corresponds to
(4)                 the *n* parameter in the call to `~numpy.fft.fft`. The default is None,
(4)                 which sets *pad_to* equal to *NFFT*
(0)             NFFT : int, default: 256
(4)                 The number of data points used in each block for the FFT.  A power 2 is
(4)                 most efficient.  This should *NOT* be used to get zero padding, or the
(4)                 scaling of the result will be incorrect; use *pad_to* for this instead.
(0)             detrend : {'none', 'mean', 'linear'} or callable, default: 'none'
(4)                 The function applied to each segment before fft-ing, designed to remove
(4)                 the mean or linear trend.  Unlike in MATLAB, where the *detrend* parameter
(4)                 is a vector, in Matplotlib it is a function.  The :mod:`~matplotlib.mlab`
(4)                 module defines `.detrend_none`, `.detrend_mean`, and `.detrend_linear`,
(4)                 but you can use a custom function as well.  You can also use a string to
(4)                 choose one of the functions: 'none' calls `.detrend_none`. 'mean' calls
(4)                 `.detrend_mean`. 'linear' calls `.detrend_linear`.
(0)             scale_by_freq : bool, default: True
(4)                 Whether the resulting density values should be scaled by the scaling
(4)                 frequency, which gives density in units of 1/Hz.  This allows for
(4)                 integration over the returned frequency values.  The default is True for
(4)                 MATLAB compatibility.""")
(0)             @_docstring.dedent_interpd
(0)             def psd(x, NFFT=None, Fs=None, detrend=None, window=None,
(8)                     noverlap=None, pad_to=None, sides=None, scale_by_freq=None):
(4)                 r"""
(4)                 Compute the power spectral density.
(4)                 The power spectral density :math:`P_{xx}` by Welch's average
(4)                 periodogram method.  The vector *x* is divided into *NFFT* length
(4)                 segments.  Each segment is detrended by function *detrend* and
(4)                 windowed by function *window*.  *noverlap* gives the length of
(4)                 the overlap between segments.  The :math:`|\mathrm{fft}(i)|^2`
(4)                 of each segment :math:`i` are averaged to compute :math:`P_{xx}`.
(4)                 If len(*x*) < *NFFT*, it will be zero padded to *NFFT*.
(4)                 Parameters
(4)                 ----------
(4)                 x : 1-D array or sequence
(8)                     Array or sequence containing the data
(4)                 %(Spectral)s
(4)                 %(PSD)s
(4)                 noverlap : int, default: 0 (no overlap)
(8)                     The number of points of overlap between segments.
(4)                 Returns
(4)                 -------
(4)                 Pxx : 1-D array
(8)                     The values for the power spectrum :math:`P_{xx}` (real valued)
(4)                 freqs : 1-D array
(8)                     The frequencies corresponding to the elements in *Pxx*
(4)                 References
(4)                 ----------
(4)                 Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John
(4)                 Wiley & Sons (1986)
(4)                 See Also
(4)                 --------
(4)                 specgram
(8)                     `specgram` differs in the default overlap; in not returning the mean
of
```

```
(8)                 the segment periodograms; and in returning the times of the segments.
(4)             magnitude_spectrum : returns the magnitude spectrum.
(4)             csd : returns the spectral density between two signals.
(4)             """
(4)             Pxx, freqs = csd(x=x, y=None, NFFT=NFFT, Fs=Fs, detrend=detrend,
(21)                             window=window, noverlap=noverlap, pad_to=pad_to,
(21)                             sides=sides, scale_by_freq=scale_by_freq)
(4)             return Pxx.real, freqs
(0)         @_docstring.dedent_interpd
(0)         def csd(x, y, NFFT=None, Fs=None, detrend=None, window=None,
(8)                 noverlap=None, pad_to=None, sides=None, scale_by_freq=None):
(4)             """
(4)             Compute the cross-spectral density.
(4)             The cross spectral density :math:`P_{xy}` by Welch's average
(4)             periodogram method.  The vectors *x* and *y* are divided into
(4)             *NFFT* length segments.  Each segment is detrended by function
(4)             *detrend* and windowed by function *window*.  *noverlap* gives
(4)             the length of the overlap between segments.  The product of
(4)             the direct FFTs of *x* and *y* are averaged over each segment
(4)             to compute :math:`P_{xy}`, with a scaling to correct for power
(4)             loss due to windowing.
(4)             If len(*x*) < *NFFT* or len(*y*) < *NFFT*, they will be zero
(4)             padded to *NFFT*.
(4)             Parameters
(4)             ----------
(4)             x, y : 1-D arrays or sequences
(8)                 Arrays or sequences containing the data
(4)             %(Spectral)s
(4)             %(PSD)s
(4)             noverlap : int, default: 0 (no overlap)
(8)                 The number of points of overlap between segments.
(4)             Returns
(4)             -------
(4)             Pxy : 1-D array
(8)                 The values for the cross spectrum :math:`P_{xy}` before scaling (real
(8)                 valued)
(4)             freqs : 1-D array
(8)                 The frequencies corresponding to the elements in *Pxy*
(4)             References
(4)             ----------
(4)             Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John
(4)             Wiley & Sons (1986)
(4)             See Also
(4)             --------
(4)             psd : equivalent to setting ``y = x``.
(4)             """
(4)             if NFFT is None:
(8)                 NFFT = 256
(4)             Pxy, freqs, _ = _spectral_helper(x=x, y=y, NFFT=NFFT, Fs=Fs,
(37)                                             detrend_func=detrend, window=window,
(37)                                             noverlap=noverlap, pad_to=pad_to,
(37)                                             sides=sides, scale_by_freq=scale_by_freq,
(37)                                             mode='psd')
(4)             if Pxy.ndim == 2:
(8)                 if Pxy.shape[1] > 1:
(12)                    Pxy = Pxy.mean(axis=1)
(8)                 else:
(12)                    Pxy = Pxy[:, 0]
(4)             return Pxy, freqs
(0)         _single_spectrum_docs = """\
(0)         Compute the {quantity} of *x*.
(0)         Data is padded to a length of *pad_to* and the windowing function *window* is
(0)         applied to the signal.
(0)         Parameters
(0)         ----------
(0)         x : 1-D array or sequence
(4)             Array or sequence containing the data
(0)         {Spectral}
(0)         {Single_Spectrum}
```

```
(0)             Returns
(0)             -------
(0)             spectrum : 1-D array
(4)                 The {quantity}.
(0)             freqs : 1-D array
(4)                 The frequencies corresponding to the elements in *spectrum*.
(0)             See Also
(0)             --------
(0)             psd
(4)                 Returns the power spectral density.
(0)             complex_spectrum
(4)                 Returns the complex-valued frequency spectrum.
(0)             magnitude_spectrum
(4)                 Returns the absolute value of the `complex_spectrum`.
(0)             angle_spectrum
(4)                 Returns the angle of the `complex_spectrum`.
(0)             phase_spectrum
(4)                 Returns the phase (unwrapped angle) of the `complex_spectrum`.
(0)             specgram
(4)                 Can return the complex spectrum of segments within the signal.
(0)             """
(0)             complex_spectrum = functools.partial(_single_spectrum_helper, "complex")
(0)             complex_spectrum.__doc__ = _single_spectrum_docs.format(
(4)                 quantity="complex-valued frequency spectrum",
(4)                 **_docstring.interpd.params)
(0)             magnitude_spectrum = functools.partial(_single_spectrum_helper, "magnitude")
(0)             magnitude_spectrum.__doc__ = _single_spectrum_docs.format(
(4)                 quantity="magnitude (absolute value) of the frequency spectrum",
(4)                 **_docstring.interpd.params)
(0)             angle_spectrum = functools.partial(_single_spectrum_helper, "angle")
(0)             angle_spectrum.__doc__ = _single_spectrum_docs.format(
(4)                 quantity="angle of the frequency spectrum (wrapped phase spectrum)",
(4)                 **_docstring.interpd.params)
(0)             phase_spectrum = functools.partial(_single_spectrum_helper, "phase")
(0)             phase_spectrum.__doc__ = _single_spectrum_docs.format(
(4)                 quantity="phase of the frequency spectrum (unwrapped phase spectrum)",
(4)                 **_docstring.interpd.params)
(0)             @_docstring.dedent_interpd
(0)             def specgram(x, NFFT=None, Fs=None, detrend=None, window=None,
(13)                         noverlap=None, pad_to=None, sides=None, scale_by_freq=None,
(13)                         mode=None):
(4)                 """
(4)                 Compute a spectrogram.
(4)                 Compute and plot a spectrogram of data in *x*.  Data are split into
(4)                 *NFFT* length segments and the spectrum of each section is
(4)                 computed.  The windowing function *window* is applied to each
(4)                 segment, and the amount of overlap of each segment is
(4)                 specified with *noverlap*.
(4)                 Parameters
(4)                 ----------
(4)                 x : array-like
(8)                     1-D array or sequence.
(4)                 %(Spectral)s
(4)                 %(PSD)s
(4)                 noverlap : int, default: 128
(8)                     The number of points of overlap between blocks.
(4)                 mode : str, default: 'psd'
(8)                     What sort of spectrum to use:
(12)                        'psd'
(16)                            Returns the power spectral density.
(12)                        'complex'
(16)                            Returns the complex-valued frequency spectrum.
(12)                        'magnitude'
(16)                            Returns the magnitude spectrum.
(12)                        'angle'
(16)                            Returns the phase spectrum without unwrapping.
(12)                        'phase'
(16)                            Returns the phase spectrum with unwrapping.
(4)                 Returns
```

```
(4)                 -------
(4)                 spectrum : array-like
(8)                     2D array, columns are the periodograms of successive segments.
(4)                 freqs : array-like
(8)                     1-D array, frequencies corresponding to the rows in *spectrum*.
(4)                 t : array-like
(8)                     1-D array, the times corresponding to midpoints of segments
(8)                     (i.e the columns in *spectrum*).
(4)                 See Also
(4)                 --------
(4)                 psd : differs in the overlap and in the return values.
(4)                 complex_spectrum : similar, but with complex valued frequencies.
(4)                 magnitude_spectrum : similar single segment when *mode* is 'magnitude'.
(4)                 angle_spectrum : similar to single segment when *mode* is 'angle'.
(4)                 phase_spectrum : similar to single segment when *mode* is 'phase'.
(4)                 Notes
(4)                 -----
(4)                 *detrend* and *scale_by_freq* only apply when *mode* is set to 'psd'.
(4)                 """
(4)                 if noverlap is None:
(8)                     noverlap = 128  # default in _spectral_helper() is noverlap = 0
(4)                 if NFFT is None:
(8)                     NFFT = 256  # same default as in _spectral_helper()
(4)                 if len(x) <= NFFT:
(8)                     _api.warn_external("Only one segment is calculated since parameter "
(27)                                        f"NFFT (={NFFT}) >= signal length (={len(x)}).")
(4)                 spec, freqs, t = _spectral_helper(x=x, y=None, NFFT=NFFT, Fs=Fs,
(38)                                                  detrend_func=detrend, window=window,
(38)                                                  noverlap=noverlap, pad_to=pad_to,
(38)                                                  sides=sides,
(38)                                                  scale_by_freq=scale_by_freq,
(38)                                                  mode=mode)
(4)                 if mode != 'complex':
(8)                     spec = spec.real  # Needed since helper implements generically
(4)                 return spec, freqs, t
(0)             @_docstring.dedent_interpd
(0)             def cohere(x, y, NFFT=256, Fs=2, detrend=detrend_none, window=window_hanning,
(11)                       noverlap=0, pad_to=None, sides='default', scale_by_freq=None):
(4)                 r"""
(4)                 The coherence between *x* and *y*.  Coherence is the normalized
(4)                 cross spectral density:
(4)                 .. math::
(8)                     C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}
(4)                 Parameters
(4)                 ----------
(4)                 x, y
(8)                     Array or sequence containing the data
(4)                 %(Spectral)s
(4)                 %(PSD)s
(4)                 noverlap : int, default: 0 (no overlap)
(8)                     The number of points of overlap between segments.
(4)                 Returns
(4)                 -------
(4)                 Cxy : 1-D array
(8)                     The coherence vector.
(4)                 freqs : 1-D array
(12)                        The frequencies for the elements in *Cxy*.
(4)                 See Also
(4)                 --------
(4)                 :func:`psd`, :func:`csd` :
(8)                     For information about the methods used to compute :math:`P_{xy}`,
(8)                     :math:`P_{xx}` and :math:`P_{yy}`.
(4)                 """
(4)                 if len(x) < 2 * NFFT:
(8)                     raise ValueError(
(12)                        "Coherence is calculated by averaging over *NFFT* length "
(12)                        "segments.  Your signal is too short for your choice of *NFFT*.")
(4)                 Pxx, f = psd(x, NFFT, Fs, detrend, window, noverlap, pad_to, sides,
(17)                            scale_by_freq)
```

```
(4)                 Pyy, f = psd(y, NFFT, Fs, detrend, window, noverlap, pad_to, sides,
(17)                            scale_by_freq)
(4)                 Pxy, f = csd(x, y, NFFT, Fs, detrend, window, noverlap, pad_to, sides,
(17)                            scale_by_freq)
(4)                 Cxy = np.abs(Pxy) ** 2 / (Pxx * Pyy)
(4)                 return Cxy, f
(0)             class GaussianKDE:
(4)                 """
(4)                 Representation of a kernel-density estimate using Gaussian kernels.
(4)                 Parameters
(4)                 ----------
(4)                 dataset : array-like
(8)                     Datapoints to estimate from. In case of univariate data this is a 1-D
(8)                     array, otherwise a 2D array with shape (# of dims, # of data).
(4)                 bw_method : {'scott', 'silverman'} or float or callable, optional
(8)                     The method used to calculate the estimator bandwidth.  If a
(8)                     float, this will be used directly as `kde.factor`.  If a
(8)                     callable, it should take a `GaussianKDE` instance as only
(8)                     parameter and return a float. If None (default), 'scott' is used.
(4)                 Attributes
(4)                 ----------
(4)                 dataset : ndarray
(8)                     The dataset passed to the constructor.
(4)                 dim : int
(8)                     Number of dimensions.
(4)                 num_dp : int
(8)                     Number of datapoints.
(4)                 factor : float
(8)                     The bandwidth factor, obtained from `kde.covariance_factor`, with
which
(8)                     the covariance matrix is multiplied.
(4)                 covariance : ndarray
(8)                     The covariance matrix of *dataset*, scaled by the calculated bandwidth
(8)                     (`kde.factor`).
(4)                 inv_cov : ndarray
(8)                     The inverse of *covariance*.
(4)                 Methods
(4)                 -------
(4)                 kde.evaluate(points) : ndarray
(8)                     Evaluate the estimated pdf on a provided set of points.
(4)                 kde(points) : ndarray
(8)                     Same as kde.evaluate(points)
(4)                 """
(4)                 def __init__(self, dataset, bw_method=None):
(8)                     self.dataset = np.atleast_2d(dataset)
(8)                     if not np.array(self.dataset).size > 1:
(12)                        raise ValueError("`dataset` input should have multiple elements.")
(8)                     self.dim, self.num_dp = np.array(self.dataset).shape
(8)                     if bw_method is None:
(12)                        pass
(8)                     elif cbook._str_equal(bw_method, 'scott'):
(12)                        self.covariance_factor = self.scotts_factor
(8)                     elif cbook._str_equal(bw_method, 'silverman'):
(12)                        self.covariance_factor = self.silverman_factor
(8)                     elif isinstance(bw_method, Number):
(12)                        self._bw_method = 'use constant'
(12)                        self.covariance_factor = lambda: bw_method
(8)                     elif callable(bw_method):
(12)                        self._bw_method = bw_method
(12)                        self.covariance_factor = lambda: self._bw_method(self)
(8)                     else:
(12)                        raise ValueError("`bw_method` should be 'scott', 'silverman', a "
(29)                                         "scalar or a callable")
(8)                     self.factor = self.covariance_factor()
(8)                     if not hasattr(self, '_data_inv_cov'):
(12)                        self.data_covariance = np.atleast_2d(
(16)                            np.cov(
(20)                                self.dataset,
(20)                                rowvar=1,
```

```
647: (20)                                bias=False))
648: (12)                        self.data_inv_cov = np.linalg.inv(self.data_covariance)
649: (8)                    self.covariance = self.data_covariance * self.factor ** 2
650: (8)                    self.inv_cov = self.data_inv_cov / self.factor ** 2
651: (8)                    self.norm_factor = (np.sqrt(np.linalg.det(2 * np.pi *
self.covariance))
652: (28)                                        * self.num_dp)
653: (4)                def scotts_factor(self):
654: (8)                    return np.power(self.num_dp, -1. / (self.dim + 4))
655: (4)                def silverman_factor(self):
656: (8)                    return np.power(
657: (12)                       self.num_dp * (self.dim + 2.0) / 4.0, -1. / (self.dim + 4))
658: (4)                covariance_factor = scotts_factor
659: (4)                def evaluate(self, points):
660: (8)                    """
661: (8)                    Evaluate the estimated pdf on a set of points.
662: (8)                    Parameters
663: (8)                    ----------
664: (8)                    points : (# of dimensions, # of points)-array
665: (12)                       Alternatively, a (# of dimensions,) vector can be passed in and
666: (12)                       treated as a single point.
667: (8)                    Returns
668: (8)                    -------
669: (8)                    (# of points,)-array
670: (12)                       The values at each point.
671: (8)                    Raises
672: (8)                    ------
673: (8)                    ValueError : if the dimensionality of the input points is different
674: (21)                               than the dimensionality of the KDE.
675: (8)                    """
676: (8)                    points = np.atleast_2d(points)
677: (8)                    dim, num_m = np.array(points).shape
678: (8)                    if dim != self.dim:
679: (12)                       raise ValueError(f"points have dimension {dim}, dataset has "
680: (29)                                        f"dimension {self.dim}")
681: (8)                    result = np.zeros(num_m)
682: (8)                    if num_m >= self.num_dp:
683: (12)                       for i in range(self.num_dp):
684: (16)                           diff = self.dataset[:, i, np.newaxis] - points
685: (16)                           tdiff = np.dot(self.inv_cov, diff)
686: (16)                           energy = np.sum(diff * tdiff, axis=0) / 2.0
687: (16)                           result = result + np.exp(-energy)
688: (8)                    else:
689: (12)                       for i in range(num_m):
690: (16)                           diff = self.dataset - points[:, i, np.newaxis]
691: (16)                           tdiff = np.dot(self.inv_cov, diff)
692: (16)                           energy = np.sum(diff * tdiff, axis=0) / 2.0
693: (16)                           result[i] = np.sum(np.exp(-energy), axis=0)
694: (8)                    result = result / self.norm_factor
695: (8)                    return result
696: (4)                __call__ = evaluate
```

File 50 - offsetbox.py:

```
1: (0)              r"""
2: (0)              Container classes for `.Artist`\s.
3: (0)              `OffsetBox`
4: (4)                  The base of all container artists defined in this module.
5: (0)              `AnchoredOffsetbox`, `AnchoredText`
6: (4)                  Anchor and align an arbitrary `.Artist` or a text relative to the parent
7: (4)                  axes or a specific anchor point.
8: (0)              `DrawingArea`
9: (4)                  A container with fixed width and height. Children have a fixed position
10: (4)                 inside the container and may be clipped.
11: (0)             `HPacker`, `VPacker`
12: (4)                 Containers for layouting their children vertically or horizontally.
13: (0)             `PaddedBox`
```

```
(4)                 A container to add a padding around an `.Artist`.
(0)             `TextArea`
(4)                 Contains a single `.Text` instance.
(0)             """
(0)             import functools
(0)             import numpy as np
(0)             import matplotlib as mpl
(0)             from matplotlib import _api, _docstring
(0)             import matplotlib.artist as martist
(0)             import matplotlib.path as mpath
(0)             import matplotlib.text as mtext
(0)             import matplotlib.transforms as mtransforms
(0)             from matplotlib.font_manager import FontProperties
(0)             from matplotlib.image import BboxImage
(0)             from matplotlib.patches import (
(4)                 FancyBboxPatch, FancyArrowPatch, bbox_artist as mbbox_artist)
(0)             from matplotlib.transforms import Bbox, BboxBase, TransformedBbox
(0)             DEBUG = False
(0)             def _compat_get_offset(meth):
(4)                 """
(4)                 Decorator for the get_offset method of OffsetBox and subclasses, that
(4)                 allows supporting both the new signature (self, bbox, renderer) and the
old
(4)                 signature (self, width, height, xdescent, ydescent, renderer).
(4)                 """
(4)                 sigs = [lambda self, width, height, xdescent, ydescent, renderer:
locals(),
(12)                        lambda self, bbox, renderer: locals()]
(4)                 @functools.wraps(meth)
(4)                 def get_offset(self, *args, **kwargs):
(8)                     params = _api.select_matching_signature(sigs, self, *args, **kwargs)
(8)                     bbox = (params["bbox"] if "bbox" in params else
(16)                            Bbox.from_bounds(-params["xdescent"], -params["ydescent"],
(33)                                             params["width"], params["height"]))
(8)                     return meth(params["self"], bbox, params["renderer"])
(4)                 return get_offset
(0)             def _bbox_artist(*args, **kwargs):
(4)                 if DEBUG:
(8)                     mbbox_artist(*args, **kwargs)
(0)             def _get_packed_offsets(widths, total, sep, mode="fixed"):
(4)                 r"""
(4)                 Pack boxes specified by their *widths*.
(4)                 For simplicity of the description, the terminology used here assumes a
(4)                 horizontal layout, but the function works equally for a vertical layout.
(4)                 There are three packing *mode*\s:
(4)                 - 'fixed': The elements are packed tight to the left with a spacing of
(6)                   *sep* in between. If *total* is *None* the returned total will be the
(6)                   right edge of the last box. A non-*None* total will be passed unchecked
(6)                   to the output. In particular this means that right edge of the last
(6)                   box may be further to the right than the returned total.
(4)                 - 'expand': Distribute the boxes with equal spacing so that the left edge
(6)                   of the first box is at 0, and the right edge of the last box is at
(6)                   *total*. The parameter *sep* is ignored in this mode. A total of *None*
(6)                   is accepted and considered equal to 1. The total is returned unchanged
(6)                   (except for the conversion *None* to 1). If the total is smaller than
(6)                   the sum of the widths, the laid out boxes will overlap.
(4)                 - 'equal': If *total* is given, the total space is divided in N equal
(6)                   ranges and each box is left-aligned within its subspace.
(6)                   Otherwise (*total* is *None*), *sep* must be provided and each box is
(6)                   left-aligned in its subspace of width ``(max(widths) + sep)``. The
(6)                   total width is then calculated to be ``N * (max(widths) + sep)``.
(4)                 Parameters
(4)                 ----------
(4)                 widths : list of float
(8)                     Widths of boxes to be packed.
(4)                 total : float or None
(8)                     Intended total length. *None* if not used.
(4)                 sep : float or None
(8)                     Spacing between boxes.
```

```
(4)                 mode : {'fixed', 'expand', 'equal'}
(8)                     The packing mode.
(4)                 Returns
(4)                 -------
(4)                 total : float
(8)                     The total width needed to accommodate the laid out boxes.
(4)                 offsets : array of float
(8)                     The left offsets of the boxes.
(4)                 """
(4)                 _api.check_in_list(["fixed", "expand", "equal"], mode=mode)
(4)                 if mode == "fixed":
(8)                     offsets_ = np.cumsum([0] + [w + sep for w in widths])
(8)                     offsets = offsets_[:-1]
(8)                     if total is None:
(12)                        total = offsets_[-1] - sep
(8)                     return total, offsets
(4)                 elif mode == "expand":
(8)                     if total is None:
(12)                        total = 1
(8)                     if len(widths) > 1:
(12)                        sep = (total - sum(widths)) / (len(widths) - 1)
(8)                     else:
(12)                        sep = 0
(8)                     offsets_ = np.cumsum([0] + [w + sep for w in widths])
(8)                     offsets = offsets_[:-1]
(8)                     return total, offsets
(4)                 elif mode == "equal":
(8)                     maxh = max(widths)
(8)                     if total is None:
(12)                        if sep is None:
(16)                            raise ValueError("total and sep cannot both be None when "
(33)                                             "using layout mode 'equal'")
(12)                        total = (maxh + sep) * len(widths)
(8)                     else:
(12)                        sep = total / len(widths) - maxh
(8)                     offsets = (maxh + sep) * np.arange(len(widths))
(8)                     return total, offsets
(0)             def _get_aligned_offsets(yspans, height, align="baseline"):
(4)                 """
(4)                 Align boxes each specified by their ``(y0, y1)`` spans.
(4)                 For simplicity of the description, the terminology used here assumes a
(4)                 horizontal layout (i.e., vertical alignment), but the function works
(4)                 equally for a vertical layout.
(4)                 Parameters
(4)                 ----------
(4)                 yspans
(8)                     List of (y0, y1) spans of boxes to be aligned.
(4)                 height : float or None
(8)                     Intended total height. If None, the maximum of the heights
(8)                     (``y1 - y0``) in *yspans* is used.
(4)                 align : {'baseline', 'left', 'top', 'right', 'bottom', 'center'}
(8)                     The alignment anchor of the boxes.
(4)                 Returns
(4)                 -------
(4)                 (y0, y1)
(8)                     y range spanned by the packing.  If a *height* was originally passed
(8)                     in, then for all alignments other than "baseline", a span of ``(0,
(8)                     height)`` is used without checking that it is actually large enough).
(4)                 descent
(8)                     The descent of the packing.
(4)                 offsets
(8)                     The bottom offsets of the boxes.
(4)                 """
(4)                 _api.check_in_list(
(8)                     ["baseline", "left", "top", "right", "bottom", "center"], align=align)
(4)                 if height is None:
(8)                     height = max(y1 - y0 for y0, y1 in yspans)
(4)                 if align == "baseline":
(8)                     yspan = (min(y0 for y0, y1 in yspans), max(y1 for y0, y1 in yspans))
```

```
150: (8)                    offsets = [0] * len(yspans)
151: (4)                elif align in ["left", "bottom"]:
152: (8)                    yspan = (0, height)
153: (8)                    offsets = [-y0 for y0, y1 in yspans]
154: (4)                elif align in ["right", "top"]:
155: (8)                    yspan = (0, height)
156: (8)                    offsets = [height - y1 for y0, y1 in yspans]
157: (4)                elif align == "center":
158: (8)                    yspan = (0, height)
159: (8)                    offsets = [(height - (y1 - y0)) * .5 - y0 for y0, y1 in yspans]
160: (4)                return yspan, offsets
161: (0)            class OffsetBox(martist.Artist):
162: (4)                """
163: (4)                The OffsetBox is a simple container artist.
164: (4)                The child artists are meant to be drawn at a relative position to its
165: (4)                parent.
166: (4)                Being an artist itself, all parameters are passed on to `.Artist`.
167: (4)                """
168: (4)                def __init__(self, *args, **kwargs):
169: (8)                    super().__init__(*args)
170: (8)                    self._internal_update(kwargs)
171: (8)                    self.set_clip_on(False)
172: (8)                    self._children = []
173: (8)                    self._offset = (0, 0)
174: (4)                def set_figure(self, fig):
175: (8)                    """
176: (8)                    Set the `.Figure` for the `.OffsetBox` and all its children.
177: (8)                    Parameters
178: (8)                    ----------
179: (8)                    fig : `~matplotlib.figure.Figure`
180: (8)                    """
181: (8)                    super().set_figure(fig)
182: (8)                    for c in self.get_children():
183: (12)                       c.set_figure(fig)
184: (4)                @martist.Artist.axes.setter
185: (4)                def axes(self, ax):
186: (8)                    martist.Artist.axes.fset(self, ax)
187: (8)                    for c in self.get_children():
188: (12)                       if c is not None:
189: (16)                           c.axes = ax
190: (4)                def contains(self, mouseevent):
191: (8)                    """
192: (8)                    Delegate the mouse event contains-check to the children.
193: (8)                    As a container, the `.OffsetBox` does not respond itself to
194: (8)                    mouseevents.
195: (8)                    Parameters
196: (8)                    ----------
197: (8)                    mouseevent : `~matplotlib.backend_bases.MouseEvent`
198: (8)                    Returns
199: (8)                    -------
200: (8)                    contains : bool
201: (12)                       Whether any values are within the radius.
202: (8)                    details : dict
203: (12)                       An artist-specific dictionary of details of the event context,
204: (12)                       such as which points are contained in the pick radius. See the
205: (12)                       individual Artist subclasses for details.
206: (8)                    See Also
207: (8)                    --------
208: (8)                    .Artist.contains
209: (8)                    """
210: (8)                    if self._different_canvas(mouseevent):
211: (12)                       return False, {}
212: (8)                    for c in self.get_children():
213: (12)                       a, b = c.contains(mouseevent)
214: (12)                       if a:
215: (16)                           return a, b
216: (8)                    return False, {}
217: (4)                def set_offset(self, xy):
218: (8)                    """
```

```
(8)                     Set the offset.
(8)                     Parameters
(8)                     ----------
(8)                     xy : (float, float) or callable
(12)                        The (x, y) coordinates of the offset in display units. These can
(12)                        either be given explicitly as a tuple (x, y), or by providing a
(12)                        function that converts the extent into the offset. This function
(12)                        must have the signature::
(16)                            def offset(width, height, xdescent, ydescent, renderer) \
(0)             -> (float, float)
(8)                     """
(8)                     self._offset = xy
(8)                     self.stale = True
(4)                 @_compat_get_offset
(4)                 def get_offset(self, bbox, renderer):
(8)                     """
(8)                     Return the offset as a tuple (x, y).
(8)                     The extent parameters have to be provided to handle the case where the
(8)                     offset is dynamically determined by a callable (see
(8)                     `~.OffsetBox.set_offset`).
(8)                     Parameters
(8)                     ----------
(8)                     bbox : `.Bbox`
(8)                     renderer : `.RendererBase` subclass
(8)                     """
(8)                     return (
(12)                        self._offset(bbox.width, bbox.height, -bbox.x0, -bbox.y0,
renderer)
(12)                        if callable(self._offset)
(12)                        else self._offset)
(4)                 def set_width(self, width):
(8)                     """
(8)                     Set the width of the box.
(8)                     Parameters
(8)                     ----------
(8)                     width : float
(8)                     """
(8)                     self.width = width
(8)                     self.stale = True
(4)                 def set_height(self, height):
(8)                     """
(8)                     Set the height of the box.
(8)                     Parameters
(8)                     ----------
(8)                     height : float
(8)                     """
(8)                     self.height = height
(8)                     self.stale = True
(4)                 def get_visible_children(self):
(8)                     r"""Return a list of the visible child `.Artist`\s."""
(8)                     return [c for c in self._children if c.get_visible()]
(4)                 def get_children(self):
(8)                     r"""Return a list of the child `.Artist`\s."""
(8)                     return self._children
(4)                 def _get_bbox_and_child_offsets(self, renderer):
(8)                     """
(8)                     Return the bbox of the offsetbox and the child offsets.
(8)                     The bbox should satisfy ``x0 <= x1 and y0 <= y1``.
(8)                     Parameters
(8)                     ----------
(8)                     renderer : `.RendererBase` subclass
(8)                     Returns
(8)                     -------
(8)                     bbox
(8)                     list of (xoffset, yoffset) pairs
(8)                     """
(8)                     raise NotImplementedError(
(12)                        "get_bbox_and_offsets must be overridden in derived classes")
(4)                 def get_bbox(self, renderer):
```

```
287: (8)                    """Return the bbox of the offsetbox, ignoring parent offsets."""
288: (8)                    bbox, offsets = self._get_bbox_and_child_offsets(renderer)
289: (8)                    return bbox
290: (4)                def get_window_extent(self, renderer=None):
291: (8)                    if renderer is None:
292: (12)                       renderer = self.figure._get_renderer()
293: (8)                    bbox = self.get_bbox(renderer)
294: (8)                    try:  # Some subclasses redefine get_offset to take no args.
295: (12)                       px, py = self.get_offset(bbox, renderer)
296: (8)                    except TypeError:
297: (12)                       px, py = self.get_offset()
298: (8)                    return bbox.translated(px, py)
299: (4)                def draw(self, renderer):
300: (8)                    """
301: (8)                    Update the location of children if necessary and draw them
302: (8)                    to the given *renderer*.
303: (8)                    """
304: (8)                    bbox, offsets = self._get_bbox_and_child_offsets(renderer)
305: (8)                    px, py = self.get_offset(bbox, renderer)
306: (8)                    for c, (ox, oy) in zip(self.get_visible_children(), offsets):
307: (12)                       c.set_offset((px + ox, py + oy))
308: (12)                       c.draw(renderer)
309: (8)                    _bbox_artist(self, renderer, fill=False, props=dict(pad=0.))
310: (8)                    self.stale = False
311: (0)            class PackerBase(OffsetBox):
312: (4)                def __init__(self, pad=0., sep=0., width=None, height=None,
313: (17)                           align="baseline", mode="fixed", children=None):
314: (8)                    """
315: (8)                    Parameters
316: (8)                    ----------
317: (8)                    pad : float, default: 0.0
318: (12)                       The boundary padding in points.
319: (8)                    sep : float, default: 0.0
320: (12)                       The spacing between items in points.
321: (8)                    width, height : float, optional
322: (12)                       Width and height of the container box in pixels, calculated if
323: (12)                       *None*.
324: (8)                    align : {'top', 'bottom', 'left', 'right', 'center', 'baseline'}, \
325: (0)            default: 'baseline'
326: (12)                       Alignment of boxes.
327: (8)                    mode : {'fixed', 'expand', 'equal'}, default: 'fixed'
328: (12)                       The packing mode.
329: (12)                       - 'fixed' packs the given `.Artist`\\s tight with *sep* spacing.
330: (12)                       - 'expand' uses the maximal available space to distribute the
331: (14)                         artists with equal spacing in between.
332: (12)                       - 'equal': Each artist an equal fraction of the available space
333: (14)                         and is left-aligned (or top-aligned) therein.
334: (8)                    children : list of `.Artist`
335: (12)                       The artists to pack.
336: (8)                    Notes
337: (8)                    -----
338: (8)                    *pad* and *sep* are in points and will be scaled with the renderer
339: (8)                    dpi, while *width* and *height* are in pixels.
340: (8)                    """
341: (8)                    super().__init__()
342: (8)                    self.height = height
343: (8)                    self.width = width
344: (8)                    self.sep = sep
345: (8)                    self.pad = pad
346: (8)                    self.mode = mode
347: (8)                    self.align = align
348: (8)                    self._children = children
349: (0)            class VPacker(PackerBase):
350: (4)                """
351: (4)                VPacker packs its children vertically, automatically adjusting their
352: (4)                relative positions at draw time.
353: (4)                """
354: (4)                def _get_bbox_and_child_offsets(self, renderer):
355: (8)                    dpicor = renderer.points_to_pixels(1.)
```

```
(8)                     pad = self.pad * dpicor
(8)                     sep = self.sep * dpicor
(8)                     if self.width is not None:
(12)                        for c in self.get_visible_children():
(16)                            if isinstance(c, PackerBase) and c.mode == "expand":
(20)                                c.set_width(self.width)
(8)                     bboxes = [c.get_bbox(renderer) for c in self.get_visible_children()]
(8)                     (x0, x1), xoffsets = _get_aligned_offsets(
(12)                        [bbox.intervalx for bbox in bboxes], self.width, self.align)
(8)                     height, yoffsets = _get_packed_offsets(
(12)                        [bbox.height for bbox in bboxes], self.height, sep, self.mode)
(8)                     yoffsets = height - (yoffsets + [bbox.y1 for bbox in bboxes])
(8)                     ydescent = yoffsets[0]
(8)                     yoffsets = yoffsets - ydescent
(8)                     return (
(12)                        Bbox.from_bounds(x0, -ydescent, x1 - x0, height).padded(pad),
(12)                        [*zip(xoffsets, yoffsets)])
(0)             class HPacker(PackerBase):
(4)                 """
(4)                 HPacker packs its children horizontally, automatically adjusting their
(4)                 relative positions at draw time.
(4)                 """
(4)                 def _get_bbox_and_child_offsets(self, renderer):
(8)                     dpicor = renderer.points_to_pixels(1.)
(8)                     pad = self.pad * dpicor
(8)                     sep = self.sep * dpicor
(8)                     bboxes = [c.get_bbox(renderer) for c in self.get_visible_children()]
(8)                     if not bboxes:
(12)                        return Bbox.from_bounds(0, 0, 0, 0).padded(pad), []
(8)                     (y0, y1), yoffsets = _get_aligned_offsets(
(12)                        [bbox.intervaly for bbox in bboxes], self.height, self.align)
(8)                     width, xoffsets = _get_packed_offsets(
(12)                        [bbox.width for bbox in bboxes], self.width, sep, self.mode)
(8)                     x0 = bboxes[0].x0
(8)                     xoffsets -= ([bbox.x0 for bbox in bboxes] - x0)
(8)                     return (Bbox.from_bounds(x0, y0, width, y1 - y0).padded(pad),
(16)                            [*zip(xoffsets, yoffsets)])
(0)             class PaddedBox(OffsetBox):
(4)                 """
(4)                 A container to add a padding around an `.Artist`.
(4)                 The `.PaddedBox` contains a `.FancyBboxPatch` that is used to visualize
(4)                 it when rendering.
(4)                 """
(4)                 def __init__(self, child, pad=0., *, draw_frame=False, patch_attrs=None):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     child : `~matplotlib.artist.Artist`
(12)                        The contained `.Artist`.
(8)                     pad : float, default: 0.0
(12)                        The padding in points. This will be scaled with the renderer dpi.
(12)                        In contrast, *width* and *height* are in *pixels* and thus not
(12)                        scaled.
(8)                     draw_frame : bool
(12)                        Whether to draw the contained `.FancyBboxPatch`.
(8)                     patch_attrs : dict or None
(12)                        Additional parameters passed to the contained `.FancyBboxPatch`.
(8)                     """
(8)                     super().__init__()
(8)                     self.pad = pad
(8)                     self._children = [child]
(8)                     self.patch = FancyBboxPatch(
(12)                        xy=(0.0, 0.0), width=1., height=1.,
(12)                        facecolor='w', edgecolor='k',
(12)                        mutation_scale=1,  # self.prop.get_size_in_points(),
(12)                        snap=True,
(12)                        visible=draw_frame,
(12)                        boxstyle="square,pad=0",
(8)                     )
```

```
425: (8)                    if patch_attrs is not None:
426: (12)                       self.patch.update(patch_attrs)
427: (4)                def _get_bbox_and_child_offsets(self, renderer):
428: (8)                    pad = self.pad * renderer.points_to_pixels(1.)
429: (8)                    return (self._children[0].get_bbox(renderer).padded(pad), [(0, 0)])
430: (4)                def draw(self, renderer):
431: (8)                    bbox, offsets = self._get_bbox_and_child_offsets(renderer)
432: (8)                    px, py = self.get_offset(bbox, renderer)
433: (8)                    for c, (ox, oy) in zip(self.get_visible_children(), offsets):
434: (12)                       c.set_offset((px + ox, py + oy))
435: (8)                    self.draw_frame(renderer)
436: (8)                    for c in self.get_visible_children():
437: (12)                       c.draw(renderer)
438: (8)                    self.stale = False
439: (4)                def update_frame(self, bbox, fontsize=None):
440: (8)                    self.patch.set_bounds(bbox.bounds)
441: (8)                    if fontsize:
442: (12)                       self.patch.set_mutation_scale(fontsize)
443: (8)                    self.stale = True
444: (4)                def draw_frame(self, renderer):
445: (8)                    self.update_frame(self.get_window_extent(renderer))
446: (8)                    self.patch.draw(renderer)
447: (0)            class DrawingArea(OffsetBox):
448: (4)                """
449: (4)                The DrawingArea can contain any Artist as a child. The DrawingArea
450: (4)                has a fixed width and height. The position of children relative to
451: (4)                the parent is fixed. The children can be clipped at the
452: (4)                boundaries of the parent.
453: (4)                """
454: (4)                def __init__(self, width, height, xdescent=0., ydescent=0., clip=False):
455: (8)                    """
456: (8)                    Parameters
457: (8)                    ----------
458: (8)                    width, height : float
459: (12)                       Width and height of the container box.
460: (8)                    xdescent, ydescent : float
461: (12)                       Descent of the box in x- and y-direction.
462: (8)                    clip : bool
463: (12)                       Whether to clip the children to the box.
464: (8)                    """
465: (8)                    super().__init__()
466: (8)                    self.width = width
467: (8)                    self.height = height
468: (8)                    self.xdescent = xdescent
469: (8)                    self.ydescent = ydescent
470: (8)                    self._clip_children = clip
471: (8)                    self.offset_transform = mtransforms.Affine2D()
472: (8)                    self.dpi_transform = mtransforms.Affine2D()
473: (4)                @property
474: (4)                def clip_children(self):
475: (8)                    """
476: (8)                    If the children of this DrawingArea should be clipped
477: (8)                    by DrawingArea bounding box.
478: (8)                    """
479: (8)                    return self._clip_children
480: (4)                @clip_children.setter
481: (4)                def clip_children(self, val):
482: (8)                    self._clip_children = bool(val)
483: (8)                    self.stale = True
484: (4)                def get_transform(self):
485: (8)                    """
486: (8)                    Return the `~matplotlib.transforms.Transform` applied to the children.
487: (8)                    """
488: (8)                    return self.dpi_transform + self.offset_transform
489: (4)                def set_transform(self, t):
490: (8)                    """
491: (8)                    set_transform is ignored.
492: (8)                    """
493: (4)                def set_offset(self, xy):
```

```
(8)                 """
(8)                 Set the offset of the container.
(8)                 Parameters
(8)                 ----------
(8)                 xy : (float, float)
(12)                    The (x, y) coordinates of the offset in display units.
(8)                 """
(8)                 self._offset = xy
(8)                 self.offset_transform.clear()
(8)                 self.offset_transform.translate(xy[0], xy[1])
(8)                 self.stale = True
(4)             def get_offset(self):
(8)                 """Return offset of the container."""
(8)                 return self._offset
(4)             def get_bbox(self, renderer):
(8)                 dpi_cor = renderer.points_to_pixels(1.)
(8)                 return Bbox.from_bounds(
(12)                    -self.xdescent * dpi_cor, -self.ydescent * dpi_cor,
(12)                    self.width * dpi_cor, self.height * dpi_cor)
(4)             def add_artist(self, a):
(8)                 """Add an `.Artist` to the container box."""
(8)                 self._children.append(a)
(8)                 if not a.is_transform_set():
(12)                    a.set_transform(self.get_transform())
(8)                 if self.axes is not None:
(12)                    a.axes = self.axes
(8)                 fig = self.figure
(8)                 if fig is not None:
(12)                    a.set_figure(fig)
(4)             def draw(self, renderer):
(8)                 dpi_cor = renderer.points_to_pixels(1.)
(8)                 self.dpi_transform.clear()
(8)                 self.dpi_transform.scale(dpi_cor)
(8)                 tpath = mtransforms.TransformedPath(
(12)                    mpath.Path([[0, 0], [0, self.height],
(24)                                [self.width, self.height],
(24)                                [self.width, 0]]),
(12)                    self.get_transform())
(8)                 for c in self._children:
(12)                    if self._clip_children and not (c.clipbox or c._clippath):
(16)                        c.set_clip_path(tpath)
(12)                    c.draw(renderer)
(8)                 _bbox_artist(self, renderer, fill=False, props=dict(pad=0.))
(8)                 self.stale = False
(0)         class TextArea(OffsetBox):
(4)             """
(4)             The TextArea is a container artist for a single Text instance.
(4)             The text is placed at (0, 0) with baseline+left alignment, by default. The
(4)             width and height of the TextArea instance is the width and height of its
(4)             child text.
(4)             """
(4)             def __init__(self, s,
(17)                         *,
(17)                         textprops=None,
(17)                         multilinebaseline=False,
(17)                         ):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 s : str
(12)                    The text to be displayed.
(8)                 textprops : dict, default: {}
(12)                    Dictionary of keyword parameters to be passed to the `.Text`
(12)                    instance in the TextArea.
(8)                 multilinebaseline : bool, default: False
(12)                    Whether the baseline for multiline text is adjusted so that it
(12)                    is (approximately) center-aligned with single-line text.
(8)                 """
(8)                 if textprops is None:
```

```
(12)                        textprops = {}
(8)                     self._text = mtext.Text(0, 0, s, **textprops)
(8)                     super().__init__()
(8)                     self._children = [self._text]
(8)                     self.offset_transform = mtransforms.Affine2D()
(8)                     self._baseline_transform = mtransforms.Affine2D()
(8)                     self._text.set_transform(self.offset_transform +
(33)                                             self._baseline_transform)
(8)                     self._multilinebaseline = multilinebaseline
(4)                 def set_text(self, s):
(8)                     """Set the text of this area as a string."""
(8)                     self._text.set_text(s)
(8)                     self.stale = True
(4)                 def get_text(self):
(8)                     """Return the string representation of this area's text."""
(8)                     return self._text.get_text()
(4)                 def set_multilinebaseline(self, t):
(8)                     """
(8)                     Set multilinebaseline.
(8)                     If True, the baseline for multiline text is adjusted so that it is
(8)                     (approximately) center-aligned with single-line text.  This is used
(8)                     e.g. by the legend implementation so that single-line labels are
(8)                     baseline-aligned, but multiline labels are "center"-aligned with them.
(8)                     """
(8)                     self._multilinebaseline = t
(8)                     self.stale = True
(4)                 def get_multilinebaseline(self):
(8)                     """
(8)                     Get multilinebaseline.
(8)                     """
(8)                     return self._multilinebaseline
(4)                 def set_transform(self, t):
(8)                     """
(8)                     set_transform is ignored.
(8)                     """
(4)                 def set_offset(self, xy):
(8)                     """
(8)                     Set the offset of the container.
(8)                     Parameters
(8)                     ----------
(8)                     xy : (float, float)
(12)                        The (x, y) coordinates of the offset in display units.
(8)                     """
(8)                     self._offset = xy
(8)                     self.offset_transform.clear()
(8)                     self.offset_transform.translate(xy[0], xy[1])
(8)                     self.stale = True
(4)                 def get_offset(self):
(8)                     """Return offset of the container."""
(8)                     return self._offset
(4)                 def get_bbox(self, renderer):
(8)                     _, h_, d_ = renderer.get_text_width_height_descent(
(12)                        "lp", self._text._fontproperties,
(12)                        ismath="TeX" if self._text.get_usetex() else False)
(8)                     bbox, info, yd = self._text._get_layout(renderer)
(8)                     w, h = bbox.size
(8)                     self._baseline_transform.clear()
(8)                     if len(info) > 1 and self._multilinebaseline:
(12)                        yd_new = 0.5 * h - 0.5 * (h_ - d_)
(12)                        self._baseline_transform.translate(0, yd - yd_new)
(12)                        yd = yd_new
(8)                     else:  # single line
(12)                        h_d = max(h_ - d_, h - yd)
(12)                        h = h_d + yd
(8)                     ha = self._text.get_horizontalalignment()
(8)                     x0 = {"left": 0, "center": -w / 2, "right": -w}[ha]
(8)                     return Bbox.from_bounds(x0, -yd, w, h)
(4)                 def draw(self, renderer):
(8)                     self._text.draw(renderer)
```

```
(8)                 _bbox_artist(self, renderer, fill=False, props=dict(pad=0.))
(8)                 self.stale = False
(0)         class AuxTransformBox(OffsetBox):
(4)             """
(4)             Offset Box with the aux_transform. Its children will be
(4)             transformed with the aux_transform first then will be
(4)             offsetted. The absolute coordinate of the aux_transform is meaning
(4)             as it will be automatically adjust so that the left-lower corner
(4)             of the bounding box of children will be set to (0, 0) before the
(4)             offset transform.
(4)             It is similar to drawing area, except that the extent of the box
(4)             is not predetermined but calculated from the window extent of its
(4)             children. Furthermore, the extent of the children will be
(4)             calculated in the transformed coordinate.
(4)             """
(4)             def __init__(self, aux_transform):
(8)                 self.aux_transform = aux_transform
(8)                 super().__init__()
(8)                 self.offset_transform = mtransforms.Affine2D()
(8)                 self.ref_offset_transform = mtransforms.Affine2D()
(4)             def add_artist(self, a):
(8)                 """Add an `.Artist` to the container box."""
(8)                 self._children.append(a)
(8)                 a.set_transform(self.get_transform())
(8)                 self.stale = True
(4)             def get_transform(self):
(8)                 """
(8)                 Return the :class:`~matplotlib.transforms.Transform` applied
(8)                 to the children
(8)                 """
(8)                 return (self.aux_transform
(16)                        + self.ref_offset_transform
(16)                        + self.offset_transform)
(4)             def set_transform(self, t):
(8)                 """
(8)                 set_transform is ignored.
(8)                 """
(4)             def set_offset(self, xy):
(8)                 """
(8)                 Set the offset of the container.
(8)                 Parameters
(8)                 ----------
(8)                 xy : (float, float)
(12)                    The (x, y) coordinates of the offset in display units.
(8)                 """
(8)                 self._offset = xy
(8)                 self.offset_transform.clear()
(8)                 self.offset_transform.translate(xy[0], xy[1])
(8)                 self.stale = True
(4)             def get_offset(self):
(8)                 """Return offset of the container."""
(8)                 return self._offset
(4)             def get_bbox(self, renderer):
(8)                 _off = self.offset_transform.get_matrix()  # to be restored later
(8)                 self.ref_offset_transform.clear()
(8)                 self.offset_transform.clear()
(8)                 bboxes = [c.get_window_extent(renderer) for c in self._children]
(8)                 ub = Bbox.union(bboxes)
(8)                 self.ref_offset_transform.translate(-ub.x0, -ub.y0)
(8)                 self.offset_transform.set_matrix(_off)
(8)                 return Bbox.from_bounds(0, 0, ub.width, ub.height)
(4)             def draw(self, renderer):
(8)                 for c in self._children:
(12)                    c.draw(renderer)
(8)                 _bbox_artist(self, renderer, fill=False, props=dict(pad=0.))
(8)                 self.stale = False
(0)         class AnchoredOffsetbox(OffsetBox):
(4)             """
(4)             An offset box placed according to location *loc*.
```

```
(4)                 AnchoredOffsetbox has a single child.  When multiple children are needed,
(4)                 use an extra OffsetBox to enclose them.  By default, the offset box is
(4)                 anchored against its parent Axes. You may explicitly specify the
(4)                 *bbox_to_anchor*.
(4)                 """
(4)                 zorder = 5  # zorder of the legend
(4)                 codes = {'upper right': 1,
(13)                         'upper left': 2,
(13)                         'lower left': 3,
(13)                         'lower right': 4,
(13)                         'right': 5,
(13)                         'center left': 6,
(13)                         'center right': 7,
(13)                         'lower center': 8,
(13)                         'upper center': 9,
(13)                         'center': 10,
(13)                         }
(4)                 def __init__(self, loc, *,
(17)                             pad=0.4, borderpad=0.5,
(17)                             child=None, prop=None, frameon=True,
(17)                             bbox_to_anchor=None,
(17)                             bbox_transform=None,
(17)                             **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     loc : str
(12)                        The box location.  Valid locations are
(12)                        'upper left', 'upper center', 'upper right',
(12)                        'center left', 'center', 'center right',
(12)                        'lower left', 'lower center', 'lower right'.
(12)                        For backward compatibility, numeric values are accepted as well.
(12)                        See the parameter *loc* of `.Legend` for details.
(8)                     pad : float, default: 0.4
(12)                        Padding around the child as fraction of the fontsize.
(8)                     borderpad : float, default: 0.5
(12)                        Padding between the offsetbox frame and the *bbox_to_anchor*.
(8)                     child : `.OffsetBox`
(12)                        The box that will be anchored.
(8)                     prop : `.FontProperties`
(12)                        This is only used as a reference for paddings. If not given,
(12)                        :rc:`legend.fontsize` is used.
(8)                     frameon : bool
(12)                        Whether to draw a frame around the box.
(8)                     bbox_to_anchor : `.BboxBase`, 2-tuple, or 4-tuple of floats
(12)                        Box that is used to position the legend in conjunction with *loc*.
(8)                     bbox_transform : None or :class:`matplotlib.transforms.Transform`
(12)                        The transform for the bounding box (*bbox_to_anchor*).
(8)                     **kwargs
(12)                        All other parameters are passed on to `.OffsetBox`.
(8)                     Notes
(8)                     -----
(8)                     See `.Legend` for a detailed description of the anchoring mechanism.
(8)                     """
(8)                     super().__init__(**kwargs)
(8)                     self.set_bbox_to_anchor(bbox_to_anchor, bbox_transform)
(8)                     self.set_child(child)
(8)                     if isinstance(loc, str):
(12)                        loc = _api.check_getitem(self.codes, loc=loc)
(8)                     self.loc = loc
(8)                     self.borderpad = borderpad
(8)                     self.pad = pad
(8)                     if prop is None:
(12)                        self.prop = FontProperties(size=mpl.rcParams["legend.fontsize"])
(8)                     else:
(12)                        self.prop = FontProperties._from_any(prop)
(12)                        if isinstance(prop, dict) and "size" not in prop:
(16)                            self.prop.set_size(mpl.rcParams["legend.fontsize"])
(8)                     self.patch = FancyBboxPatch(
```

```
(12)                        xy=(0.0, 0.0), width=1., height=1.,
(12)                        facecolor='w', edgecolor='k',
(12)                        mutation_scale=self.prop.get_size_in_points(),
(12)                        snap=True,
(12)                        visible=frameon,
(12)                        boxstyle="square,pad=0",
(8)                     )
(4)                 def set_child(self, child):
(8)                     """Set the child to be anchored."""
(8)                     self._child = child
(8)                     if child is not None:
(12)                        child.axes = self.axes
(8)                     self.stale = True
(4)                 def get_child(self):
(8)                     """Return the child."""
(8)                     return self._child
(4)                 def get_children(self):
(8)                     """Return the list of children."""
(8)                     return [self._child]
(4)                 def get_bbox(self, renderer):
(8)                     fontsize = renderer.points_to_pixels(self.prop.get_size_in_points())
(8)                     pad = self.pad * fontsize
(8)                     return self.get_child().get_bbox(renderer).padded(pad)
(4)                 def get_bbox_to_anchor(self):
(8)                     """Return the bbox that the box is anchored to."""
(8)                     if self._bbox_to_anchor is None:
(12)                        return self.axes.bbox
(8)                     else:
(12)                        transform = self._bbox_to_anchor_transform
(12)                        if transform is None:
(16)                            return self._bbox_to_anchor
(12)                        else:
(16)                            return TransformedBbox(self._bbox_to_anchor, transform)
(4)                 def set_bbox_to_anchor(self, bbox, transform=None):
(8)                     """
(8)                     Set the bbox that the box is anchored to.
(8)                     *bbox* can be a Bbox instance, a list of [left, bottom, width,
(8)                     height], or a list of [left, bottom] where the width and
(8)                     height will be assumed to be zero. The bbox will be
(8)                     transformed to display coordinate by the given transform.
(8)                     """
(8)                     if bbox is None or isinstance(bbox, BboxBase):
(12)                        self._bbox_to_anchor = bbox
(8)                     else:
(12)                        try:
(16)                            l = len(bbox)
(12)                        except TypeError as err:
(16)                            raise ValueError(f"Invalid bbox: {bbox}") from err
(12)                        if l == 2:
(16)                            bbox = [bbox[0], bbox[1], 0, 0]
(12)                        self._bbox_to_anchor = Bbox.from_bounds(*bbox)
(8)                     self._bbox_to_anchor_transform = transform
(8)                     self.stale = True
(4)                 @_compat_get_offset
(4)                 def get_offset(self, bbox, renderer):
(8)                     pad = (self.borderpad
(15)                           * renderer.points_to_pixels(self.prop.get_size_in_points()))
(8)                     bbox_to_anchor = self.get_bbox_to_anchor()
(8)                     x0, y0 = _get_anchored_bbox(
(12)                        self.loc, Bbox.from_bounds(0, 0, bbox.width, bbox.height),
(12)                        bbox_to_anchor, pad)
(8)                     return x0 - bbox.x0, y0 - bbox.y0
(4)                 def update_frame(self, bbox, fontsize=None):
(8)                     self.patch.set_bounds(bbox.bounds)
(8)                     if fontsize:
(12)                        self.patch.set_mutation_scale(fontsize)
(4)                 def draw(self, renderer):
(8)                     if not self.get_visible():
(12)                        return
```

```
(8)                     bbox = self.get_window_extent(renderer)
(8)                     fontsize = renderer.points_to_pixels(self.prop.get_size_in_points())
(8)                     self.update_frame(bbox, fontsize)
(8)                     self.patch.draw(renderer)
(8)                     px, py = self.get_offset(self.get_bbox(renderer), renderer)
(8)                     self.get_child().set_offset((px, py))
(8)                     self.get_child().draw(renderer)
(8)                     self.stale = False
(0)             def _get_anchored_bbox(loc, bbox, parentbbox, borderpad):
(4)                 """
(4)                 Return the (x, y) position of the *bbox* anchored at the *parentbbox* with
(4)                 the *loc* code with the *borderpad*.
(4)                 """
(4)                 c = [None, "NE", "NW", "SW", "SE", "E", "W", "E", "S", "N", "C"][loc]
(4)                 container = parentbbox.padded(-borderpad)
(4)                 return bbox.anchored(c, container=container).p0
(0)             class AnchoredText(AnchoredOffsetbox):
(4)                 """
(4)                 AnchoredOffsetbox with Text.
(4)                 """
(4)                 def __init__(self, s, loc, *, pad=0.4, borderpad=0.5, prop=None,
**kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     s : str
(12)                        Text.
(8)                     loc : str
(12)                        Location code. See `AnchoredOffsetbox`.
(8)                     pad : float, default: 0.4
(12)                        Padding around the text as fraction of the fontsize.
(8)                     borderpad : float, default: 0.5
(12)                        Spacing between the offsetbox frame and the *bbox_to_anchor*.
(8)                     prop : dict, optional
(12)                        Dictionary of keyword parameters to be passed to the
(12)                        `~matplotlib.text.Text` instance contained inside AnchoredText.
(8)                     **kwargs
(12)                        All other parameters are passed to `AnchoredOffsetbox`.
(8)                     """
(8)                     if prop is None:
(12)                        prop = {}
(8)                     badkwargs = {'va', 'verticalalignment'}
(8)                     if badkwargs & set(prop):
(12)                        raise ValueError(
(16)                            'Mixing verticalalignment with AnchoredText is not
supported.')
(8)                     self.txt = TextArea(s, textprops=prop)
(8)                     fp = self.txt._text.get_fontproperties()
(8)                     super().__init__(
(12)                        loc, pad=pad, borderpad=borderpad, child=self.txt, prop=fp,
(12)                        **kwargs)
(0)             class OffsetImage(OffsetBox):
(4)                 def __init__(self, arr, *,
(17)                             zoom=1,
(17)                             cmap=None,
(17)                             norm=None,
(17)                             interpolation=None,
(17)                             origin=None,
(17)                             filternorm=True,
(17)                             filterrad=4.0,
(17)                             resample=False,
(17)                             dpi_cor=True,
(17)                             **kwargs
(17)                             ):
(8)                     super().__init__()
(8)                     self._dpi_cor = dpi_cor
(8)                     self.image = BboxImage(bbox=self.get_window_extent,
(31)                                            cmap=cmap,
(31)                                            norm=norm,
```

```
(31)                                        interpolation=interpolation,
(31)                                        origin=origin,
(31)                                        filternorm=filternorm,
(31)                                        filterrad=filterrad,
(31)                                        resample=resample,
(31)                                        **kwargs
(31)                                        )
(8)                     self._children = [self.image]
(8)                     self.set_zoom(zoom)
(8)                     self.set_data(arr)
(4)                 def set_data(self, arr):
(8)                     self._data = np.asarray(arr)
(8)                     self.image.set_data(self._data)
(8)                     self.stale = True
(4)                 def get_data(self):
(8)                     return self._data
(4)                 def set_zoom(self, zoom):
(8)                     self._zoom = zoom
(8)                     self.stale = True
(4)                 def get_zoom(self):
(8)                     return self._zoom
(4)                 def get_offset(self):
(8)                     """Return offset of the container."""
(8)                     return self._offset
(4)                 def get_children(self):
(8)                     return [self.image]
(4)                 def get_bbox(self, renderer):
(8)                     dpi_cor = renderer.points_to_pixels(1.) if self._dpi_cor else 1.
(8)                     zoom = self.get_zoom()
(8)                     data = self.get_data()
(8)                     ny, nx = data.shape[:2]
(8)                     w, h = dpi_cor * nx * zoom, dpi_cor * ny * zoom
(8)                     return Bbox.from_bounds(0, 0, w, h)
(4)                 def draw(self, renderer):
(8)                     self.image.draw(renderer)
(8)                     self.stale = False
(0)             class AnnotationBbox(martist.Artist, mtext._AnnotationBase):
(4)                 """
(4)                 Container for an `OffsetBox` referring to a specific position *xy*.
(4)                 Optionally an arrow pointing from the offsetbox to *xy* can be drawn.
(4)                 This is like `.Annotation`, but with `OffsetBox` instead of `.Text`.
(4)                 """
(4)                 zorder = 3
(4)                 def __str__(self):
(8)                     return f"AnnotationBbox({self.xy[0]:g},{self.xy[1]:g})"
(4)                 @_docstring.dedent_interpd
(4)                 def __init__(self, offsetbox, xy, xybox=None, xycoords='data',
boxcoords=None, *,
(17)                            frameon=True, pad=0.4,  # FancyBboxPatch boxstyle.
(17)                            annotation_clip=None,
(17)                            box_alignment=(0.5, 0.5),
(17)                            bboxprops=None,
(17)                            arrowprops=None,
(17)                            fontsize=None,
(17)                            **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     offsetbox : `OffsetBox`
(8)                     xy : (float, float)
(12)                        The point *(x, y)* to annotate. The coordinate system is
determined
(12)                        by *xycoords*.
(8)                     xybox : (float, float), default: *xy*
(12)                        The position *(x, y)* to place the text at. The coordinate system
(12)                        is determined by *boxcoords*.
(8)                     xycoords : single or two-tuple of str or `.Artist` or `.Transform` or
\
(0)             callable, default: 'data'
```

```
(12)                        The coordinate system that *xy* is given in. See the parameter
(12)                        *xycoords* in `.Annotation` for a detailed description.
(8)                     boxcoords : single or two-tuple of str or `.Artist` or `.Transform` \
(0)             or callable, default: value of *xycoords*
(12)                        The coordinate system that *xybox* is given in. See the parameter
(12)                        *textcoords* in `.Annotation` for a detailed description.
(8)                     frameon : bool, default: True
(12)                        By default, the text is surrounded by a white `.FancyBboxPatch`
(12)                        (accessible as the ``patch`` attribute of the `.AnnotationBbox`).
(12)                        If *frameon* is set to False, this patch is made invisible.
(8)                     annotation_clip: bool or None, default: None
(12)                        Whether to clip (i.e. not draw) the annotation when the annotation
(12)                        point *xy* is outside the Axes area.
(12)                        - If *True*, the annotation will be clipped when *xy* is outside
(14)                          the Axes.
(12)                        - If *False*, the annotation will always be drawn.
(12)                        - If *None*, the annotation will be clipped when *xy* is outside
(14)                          the Axes and *xycoords* is 'data'.
(8)                     pad : float, default: 0.4
(12)                        Padding around the offsetbox.
(8)                     box_alignment : (float, float)
(12)                        A tuple of two floats for a vertical and horizontal alignment of
(12)                        the offset box w.r.t. the *boxcoords*.
(12)                        The lower-left corner is (0, 0) and upper-right corner is (1, 1).
(8)                     bboxprops : dict, optional
(12)                        A dictionary of properties to set for the annotation bounding box,
(12)                        for example *boxstyle* and *alpha*.  See `.FancyBboxPatch` for
(12)                        details.
(8)                     arrowprops: dict, optional
(12)                        Arrow properties, see `.Annotation` for description.
(8)                     fontsize: float or str, optional
(12)                        Translated to points and passed as *mutation_scale* into
(12)                        `.FancyBboxPatch` to scale attributes of the box style (e.g. pad
(12)                        or rounding_size).  The name is chosen in analogy to `.Text` where
(12)                        *fontsize* defines the mutation scale as well.  If not given,
(12)                        :rc:`legend.fontsize` is used.  See `.Text.set_fontsize` for valid
(12)                        values.
(8)                     **kwargs
(12)                        Other `AnnotationBbox` properties.  See `.AnnotationBbox.set` for
(12)                        a list.
(8)                     """
(8)                     martist.Artist.__init__(self)
(8)                     mtext._AnnotationBase.__init__(
(12)                        self, xy, xycoords=xycoords, annotation_clip=annotation_clip)
(8)                     self.offsetbox = offsetbox
(8)                     self.arrowprops = arrowprops.copy() if arrowprops is not None else
None
(8)                     self.set_fontsize(fontsize)
(8)                     self.xybox = xybox if xybox is not None else xy
(8)                     self.boxcoords = boxcoords if boxcoords is not None else xycoords
(8)                     self._box_alignment = box_alignment
(8)                     if arrowprops is not None:
(12)                        self._arrow_relpos = self.arrowprops.pop("relpos", (0.5, 0.5))
(12)                        self.arrow_patch = FancyArrowPatch((0, 0), (1, 1),
(47)                                                           **self.arrowprops)
(8)                     else:
(12)                        self._arrow_relpos = None
(12)                        self.arrow_patch = None
(8)                     self.patch = FancyBboxPatch(  # frame
(12)                        xy=(0.0, 0.0), width=1., height=1.,
(12)                        facecolor='w', edgecolor='k',
(12)                        mutation_scale=self.prop.get_size_in_points(),
(12)                        snap=True,
(12)                        visible=frameon,
(8)                     )
(8)                     self.patch.set_boxstyle("square", pad=pad)
(8)                     if bboxprops:
(12)                        self.patch.set(**bboxprops)
(8)                     self._internal_update(kwargs)
```

```
    @property
    def xyann(self):
        return self.xybox
    @xyann.setter
    def xyann(self, xyann):
        self.xybox = xyann
        self.stale = True
    @property
    def anncoords(self):
        return self.boxcoords
    @anncoords.setter
    def anncoords(self, coords):
        self.boxcoords = coords
        self.stale = True
    def contains(self, mouseevent):
        if self._different_canvas(mouseevent):
            return False, {}
        if not self._check_xy(None):
            return False, {}
        return self.offsetbox.contains(mouseevent)
    def get_children(self):
        children = [self.offsetbox, self.patch]
        if self.arrow_patch:
            children.append(self.arrow_patch)
        return children
    def set_figure(self, fig):
        if self.arrow_patch is not None:
            self.arrow_patch.set_figure(fig)
        self.offsetbox.set_figure(fig)
        martist.Artist.set_figure(self, fig)
    def set_fontsize(self, s=None):
        """
        Set the fontsize in points.
        If *s* is not given, reset to :rc:`legend.fontsize`.
        """
        if s is None:
            s = mpl.rcParams["legend.fontsize"]
        self.prop = FontProperties(size=s)
        self.stale = True
    def get_fontsize(self):
        """Return the fontsize in points."""
        return self.prop.get_size_in_points()
    def get_window_extent(self, renderer=None):
        if renderer is None:
            renderer = self.figure._get_renderer()
        self.update_positions(renderer)
        return Bbox.union([child.get_window_extent(renderer)
                           for child in self.get_children()])
    def get_tightbbox(self, renderer=None):
        if renderer is None:
            renderer = self.figure._get_renderer()
        self.update_positions(renderer)
        return Bbox.union([child.get_tightbbox(renderer)
                           for child in self.get_children()])
    def update_positions(self, renderer):
        """Update pixel positions for the annotated point, the text, and the arrow."""
        ox0, oy0 = self._get_xy(renderer, self.xybox, self.boxcoords)
        bbox = self.offsetbox.get_bbox(renderer)
        fw, fh = self._box_alignment
        self.offsetbox.set_offset(
            (ox0 - fw*bbox.width - bbox.x0, oy0 - fh*bbox.height - bbox.y0))
        bbox = self.offsetbox.get_window_extent(renderer)
        self.patch.set_bounds(bbox.bounds)
        mutation_scale = renderer.points_to_pixels(self.get_fontsize())
        self.patch.set_mutation_scale(mutation_scale)
        if self.arrowprops:
            arrow_begin = bbox.p0 + bbox.size * self._arrow_relpos
            arrow_end = self._get_position_xy(renderer)
```

```
(12)                        self.arrow_patch.set_positions(arrow_begin, arrow_end)
(12)                        if "mutation_scale" in self.arrowprops:
(16)                            mutation_scale = renderer.points_to_pixels(
(20)                                self.arrowprops["mutation_scale"])
(12)                        self.arrow_patch.set_mutation_scale(mutation_scale)
(12)                        patchA = self.arrowprops.get("patchA", self.patch)
(12)                        self.arrow_patch.set_patchA(patchA)
(4)                 def draw(self, renderer):
(8)                     if not self.get_visible() or not self._check_xy(renderer):
(12)                        return
(8)                     renderer.open_group(self.__class__.__name__, gid=self.get_gid())
(8)                     self.update_positions(renderer)
(8)                     if self.arrow_patch is not None:
(12)                        if self.arrow_patch.figure is None and self.figure is not None:
(16)                            self.arrow_patch.figure = self.figure
(12)                        self.arrow_patch.draw(renderer)
(8)                     self.patch.draw(renderer)
(8)                     self.offsetbox.draw(renderer)
(8)                     renderer.close_group(self.__class__.__name__)
(8)                     self.stale = False
(0)             class DraggableBase:
(4)                 """
(4)                 Helper base class for a draggable artist (legend, offsetbox).
(4)                 Derived classes must override the following methods::
(8)                     def save_offset(self):
(12)                        '''
(12)                        Called when the object is picked for dragging; should save the
(12)                        reference position of the artist.
(12)                        '''
(8)                     def update_offset(self, dx, dy):
(12)                        '''
(12)                        Called during the dragging; (*dx*, *dy*) is the pixel offset from
(12)                        the point where the mouse drag started.
(12)                        '''
(4)                 Optionally, you may override the following method::
(8)                     def finalize_offset(self):
(12)                        '''Called when the mouse is released.'''
(4)                 In the current implementation of `.DraggableLegend` and
(4)                 `DraggableAnnotation`, `update_offset` places the artists in display
(4)                 coordinates, and `finalize_offset` recalculates their position in axes
(4)                 coordinate and set a relevant attribute.
(4)                 """
(4)                 def __init__(self, ref_artist, use_blit=False):
(8)                     self.ref_artist = ref_artist
(8)                     if not ref_artist.pickable():
(12)                        ref_artist.set_picker(True)
(8)                     self.got_artist = False
(8)                     self._use_blit = use_blit and self.canvas.supports_blit
(8)                     callbacks = self.canvas.callbacks
(8)                     self._disconnectors = [
(12)                        functools.partial(
(16)                            callbacks.disconnect, callbacks._connect_picklable(name,
func))
(12)                        for name, func in [
(16)                            ("pick_event", self.on_pick),
(16)                            ("button_release_event", self.on_release),
(16)                            ("motion_notify_event", self.on_motion),
(12)                        ]
(8)                     ]
(4)                 canvas = property(lambda self: self.ref_artist.figure.canvas)
(4)                 cids = property(lambda self: [
(8)                     disconnect.args[0] for disconnect in self._disconnectors[:2]])
(4)                 def on_motion(self, evt):
(8)                     if self._check_still_parented() and self.got_artist:
(12)                        dx = evt.x - self.mouse_x
(12)                        dy = evt.y - self.mouse_y
(12)                        self.update_offset(dx, dy)
(12)                        if self._use_blit:
(16)                            self.canvas.restore_region(self.background)
```

```
(16)                        self.ref_artist.draw(
(20)                            self.ref_artist.figure._get_renderer())
(16)                        self.canvas.blit()
(12)                    else:
(16)                        self.canvas.draw()
(4)             def on_pick(self, evt):
(8)                 if self._check_still_parented() and evt.artist == self.ref_artist:
(12)                    self.mouse_x = evt.mouseevent.x
(12)                    self.mouse_y = evt.mouseevent.y
(12)                    self.got_artist = True
(12)                    if self._use_blit:
(16)                        self.ref_artist.set_animated(True)
(16)                        self.canvas.draw()
(16)                        self.background = \
(20)                            self.canvas.copy_from_bbox(self.ref_artist.figure.bbox)
(16)                        self.ref_artist.draw(
(20)                            self.ref_artist.figure._get_renderer())
(16)                        self.canvas.blit()
(12)                    self.save_offset()
(4)             def on_release(self, event):
(8)                 if self._check_still_parented() and self.got_artist:
(12)                    self.finalize_offset()
(12)                    self.got_artist = False
(12)                    if self._use_blit:
(16)                        self.ref_artist.set_animated(False)
(4)             def _check_still_parented(self):
(8)                 if self.ref_artist.figure is None:
(12)                    self.disconnect()
(12)                    return False
(8)                 else:
(12)                    return True
(4)             def disconnect(self):
(8)                 """Disconnect the callbacks."""
(8)                 for disconnector in self._disconnectors:
(12)                    disconnector()
(4)             def save_offset(self):
(8)                 pass
(4)             def update_offset(self, dx, dy):
(8)                 pass
(4)             def finalize_offset(self):
(8)                 pass
(0)         class DraggableOffsetBox(DraggableBase):
(4)             def __init__(self, ref_artist, offsetbox, use_blit=False):
(8)                 super().__init__(ref_artist, use_blit=use_blit)
(8)                 self.offsetbox = offsetbox
(4)             def save_offset(self):
(8)                 offsetbox = self.offsetbox
(8)                 renderer = offsetbox.figure._get_renderer()
(8)                 offset = offsetbox.get_offset(offsetbox.get_bbox(renderer), renderer)
(8)                 self.offsetbox_x, self.offsetbox_y = offset
(8)                 self.offsetbox.set_offset(offset)
(4)             def update_offset(self, dx, dy):
(8)                 loc_in_canvas = self.offsetbox_x + dx, self.offsetbox_y + dy
(8)                 self.offsetbox.set_offset(loc_in_canvas)
(4)             def get_loc_in_canvas(self):
(8)                 offsetbox = self.offsetbox
(8)                 renderer = offsetbox.figure._get_renderer()
(8)                 bbox = offsetbox.get_bbox(renderer)
(8)                 ox, oy = offsetbox._offset
(8)                 loc_in_canvas = (ox + bbox.x0, oy + bbox.y0)
(8)                 return loc_in_canvas
(0)         class DraggableAnnotation(DraggableBase):
(4)             def __init__(self, annotation, use_blit=False):
(8)                 super().__init__(annotation, use_blit=use_blit)
(8)                 self.annotation = annotation
(4)             def save_offset(self):
(8)                 ann = self.annotation
(8)                 self.ox, self.oy = ann.get_transform().transform(ann.xyann)
(4)             def update_offset(self, dx, dy):
```

```
1245: (8)                   ann = self.annotation
1246: (8)                   ann.xyann = ann.get_transform().inverted().transform(
1247: (12)                      (self.ox + dx, self.oy + dy))
```

File 51 - patches.py:

```
1: (0)              r"""
2: (0)              Patches are `.Artist`\s with a face color and an edge color.
3: (0)              """
4: (0)              import functools
5: (0)              import inspect
6: (0)              import math
7: (0)              from numbers import Number, Real
8: (0)              import textwrap
9: (0)              from types import SimpleNamespace
10: (0)             from collections import namedtuple
11: (0)             from matplotlib.transforms import Affine2D
12: (0)             import numpy as np
13: (0)             import matplotlib as mpl
14: (0)             from . import (_api, artist, cbook, colors, _docstring, hatch as mhatch,
15: (15)                           lines as mlines, transforms)
16: (0)             from .bezier import (
17: (4)                 NonIntersectingPathException, get_cos_sin, get_intersection,
18: (4)                 get_parallels, inside_circle, make_wedged_bezier2,
19: (4)                 split_bezier_intersecting_with_closedpath, split_path_inout)
20: (0)             from .path import Path
21: (0)             from ._enums import JoinStyle, CapStyle
22: (0)             @_docstring.interpd
23: (0)             @_api.define_aliases({
24: (4)                 "antialiased": ["aa"],
25: (4)                 "edgecolor": ["ec"],
26: (4)                 "facecolor": ["fc"],
27: (4)                 "linestyle": ["ls"],
28: (4)                 "linewidth": ["lw"],
29: (0)             })
30: (0)             class Patch(artist.Artist):
31: (4)                 """
32: (4)                 A patch is a 2D artist with a face color and an edge color.
33: (4)                 If any of *edgecolor*, *facecolor*, *linewidth*, or *antialiased*
34: (4)                 are *None*, they default to their rc params setting.
35: (4)                 """
36: (4)                 zorder = 1
37: (4)                 _edge_default = False
38: (4)                 def __init__(self, *,
39: (17)                            edgecolor=None,
40: (17)                            facecolor=None,
41: (17)                            color=None,
42: (17)                            linewidth=None,
43: (17)                            linestyle=None,
44: (17)                            antialiased=None,
45: (17)                            hatch=None,
46: (17)                            fill=True,
47: (17)                            capstyle=None,
48: (17)                            joinstyle=None,
49: (17)                            **kwargs):
50: (8)                     """
51: (8)                     The following kwarg properties are supported
52: (8)                     %(Patch:kwdoc)s
53: (8)                     """
54: (8)                     super().__init__()
55: (8)                     if linestyle is None:
56: (12)                        linestyle = "solid"
57: (8)                     if capstyle is None:
58: (12)                        capstyle = CapStyle.butt
59: (8)                     if joinstyle is None:
60: (12)                        joinstyle = JoinStyle.miter
61: (8)                     self._hatch_color = colors.to_rgba(mpl.rcParams['hatch.color'])
```

```
62: (8)                     self._fill = bool(fill)  # needed for set_facecolor call
63: (8)                     if color is not None:
64: (12)                        if edgecolor is not None or facecolor is not None:
65: (16)                            _api.warn_external(
66: (20)                                "Setting the 'color' property will override "
67: (20)                                "the edgecolor or facecolor properties.")
68: (12)                        self.set_color(color)
69: (8)                     else:
70: (12)                        self.set_edgecolor(edgecolor)
71: (12)                        self.set_facecolor(facecolor)
72: (8)                     self._linewidth = 0
73: (8)                     self._unscaled_dash_pattern = (0, None)  # offset, dash
74: (8)                     self._dash_pattern = (0, None)  # offset, dash (scaled by linewidth)
75: (8)                     self.set_linestyle(linestyle)
76: (8)                     self.set_linewidth(linewidth)
77: (8)                     self.set_antialiased(antialiased)
78: (8)                     self.set_hatch(hatch)
79: (8)                     self.set_capstyle(capstyle)
80: (8)                     self.set_joinstyle(joinstyle)
81: (8)                     if len(kwargs):
82: (12)                        self._internal_update(kwargs)
83: (4)                 def get_verts(self):
84: (8)                     """
85: (8)                     Return a copy of the vertices used in this patch.
86: (8)                     If the patch contains Bézier curves, the curves will be interpolated
by
87: (8)                     line segments.  To access the curves as curves, use `get_path`.
88: (8)                     """
89: (8)                     trans = self.get_transform()
90: (8)                     path = self.get_path()
91: (8)                     polygons = path.to_polygons(trans)
92: (8)                     if len(polygons):
93: (12)                        return polygons[0]
94: (8)                     return []
95: (4)                 def _process_radius(self, radius):
96: (8)                     if radius is not None:
97: (12)                        return radius
98: (8)                     if isinstance(self._picker, Number):
99: (12)                        _radius = self._picker
100: (8)                    else:
101: (12)                       if self.get_edgecolor()[3] == 0:
102: (16)                           _radius = 0
103: (12)                       else:
104: (16)                           _radius = self.get_linewidth()
105: (8)                    return _radius
106: (4)                def contains(self, mouseevent, radius=None):
107: (8)                    """
108: (8)                    Test whether the mouse event occurred in the patch.
109: (8)                    Parameters
110: (8)                    ----------
111: (8)                    mouseevent : `~matplotlib.backend_bases.MouseEvent`
112: (12)                       Where the user clicked.
113: (8)                    radius : float, optional
114: (12)                       Additional margin on the patch in target coordinates of
115: (12)                       `.Patch.get_transform`. See `.Path.contains_point` for further
116: (12)                       details.
117: (12)                       If `None`, the default value depends on the state of the object:
118: (12)                       - If `.Artist.get_picker` is a number, the default
119: (14)                         is that value.  This is so that picking works as expected.
120: (12)                       - Otherwise if the edge color has a non-zero alpha, the default
121: (14)                         is half of the linewidth.  This is so that all the colored
122: (14)                         pixels are "in" the patch.
123: (12)                       - Finally, if the edge has 0 alpha, the default is 0.  This is
124: (14)                         so that patches without a stroked edge do not have points
125: (14)                         outside of the filled region report as "in" due to an
126: (14)                         invisible edge.
127: (8)                    Returns
128: (8)                    -------
129: (8)                    (bool, empty dict)
```

```
(8)                 """
(8)                 if self._different_canvas(mouseevent):
(12)                    return False, {}
(8)                 radius = self._process_radius(radius)
(8)                 codes = self.get_path().codes
(8)                 if codes is not None:
(12)                    vertices = self.get_path().vertices
(12)                    idxs, = np.where(codes == Path.MOVETO)
(12)                    idxs = idxs[1:]
(12)                    subpaths = map(
(16)                        Path, np.split(vertices, idxs), np.split(codes, idxs))
(8)                 else:
(12)                    subpaths = [self.get_path()]
(8)                 inside = any(
(12)                    subpath.contains_point(
(16)                        (mouseevent.x, mouseevent.y), self.get_transform(), radius)
(12)                    for subpath in subpaths)
(8)                 return inside, {}
(4)             def contains_point(self, point, radius=None):
(8)                 """
(8)                 Return whether the given point is inside the patch.
(8)                 Parameters
(8)                 ----------
(8)                 point : (float, float)
(12)                    The point (x, y) to check, in target coordinates of
(12)                    ``.Patch.get_transform()``. These are display coordinates for
patches
(12)                    that are added to a figure or Axes.
(8)                 radius : float, optional
(12)                    Additional margin on the patch in target coordinates of
(12)                    `.Patch.get_transform`. See `.Path.contains_point` for further
(12)                    details.
(12)                    If `None`, the default value depends on the state of the object:
(12)                    - If `.Artist.get_picker` is a number, the default
(14)                      is that value.  This is so that picking works as expected.
(12)                    - Otherwise if the edge color has a non-zero alpha, the default
(14)                      is half of the linewidth.  This is so that all the colored
(14)                      pixels are "in" the patch.
(12)                    - Finally, if the edge has 0 alpha, the default is 0.  This is
(14)                      so that patches without a stroked edge do not have points
(14)                      outside of the filled region report as "in" due to an
(14)                      invisible edge.
(8)                 Returns
(8)                 -------
(8)                 bool
(8)                 Notes
(8)                 -----
(8)                 The proper use of this method depends on the transform of the patch.
(8)                 Isolated patches do not have a transform. In this case, the patch
(8)                 creation coordinates and the point coordinates match. The following
(8)                 example checks that the center of a circle is within the circle
(8)                 >>> center = 0, 0
(8)                 >>> c = Circle(center, radius=1)
(8)                 >>> c.contains_point(center)
(8)                 True
(8)                 The convention of checking against the transformed patch stems from
(8)                 the fact that this method is predominantly used to check if display
(8)                 coordinates (e.g. from mouse events) are within the patch. If you want
(8)                 to do the above check with data coordinates, you have to properly
(8)                 transform them first:
(8)                 >>> center = 0, 0
(8)                 >>> c = Circle(center, radius=3)
(8)                 >>> plt.gca().add_patch(c)
(8)                 >>> transformed_interior_point = c.get_data_transform().transform((0,
2))
(8)                 >>> c.contains_point(transformed_interior_point)
(8)                 True
(8)                 """
(8)                 radius = self._process_radius(radius)
```

```
197: (8)                    return self.get_path().contains_point(point,
198: (46)                                                 self.get_transform(),
199: (46)                                                 radius)
200: (4)                def contains_points(self, points, radius=None):
201: (8)                    """
202: (8)                    Return whether the given points are inside the patch.
203: (8)                    Parameters
204: (8)                    ----------
205: (8)                    points : (N, 2) array
206: (12)                       The points to check, in target coordinates of
207: (12)                       ``self.get_transform()``. These are display coordinates for
patches
208: (12)                       that are added to a figure or Axes. Columns contain x and y
values.
209: (8)                    radius : float, optional
210: (12)                       Additional margin on the patch in target coordinates of
211: (12)                       `.Patch.get_transform`. See `.Path.contains_point` for further
212: (12)                       details.
213: (12)                       If `None`, the default value depends on the state of the object:
214: (12)                       - If `.Artist.get_picker` is a number, the default
215: (14)                         is that value.  This is so that picking works as expected.
216: (12)                       - Otherwise if the edge color has a non-zero alpha, the default
217: (14)                         is half of the linewidth.  This is so that all the colored
218: (14)                         pixels are "in" the patch.
219: (12)                       - Finally, if the edge has 0 alpha, the default is 0.  This is
220: (14)                         so that patches without a stroked edge do not have points
221: (14)                         outside of the filled region report as "in" due to an
222: (14)                         invisible edge.
223: (8)                    Returns
224: (8)                    -------
225: (8)                    length-N bool array
226: (8)                    Notes
227: (8)                    -----
228: (8)                    The proper use of this method depends on the transform of the patch.
229: (8)                    See the notes on `.Patch.contains_point`.
230: (8)                    """
231: (8)                    radius = self._process_radius(radius)
232: (8)                    return self.get_path().contains_points(points,
233: (47)                                                  self.get_transform(),
234: (47)                                                  radius)
235: (4)                def update_from(self, other):
236: (8)                    super().update_from(other)
237: (8)                    self._edgecolor = other._edgecolor
238: (8)                    self._facecolor = other._facecolor
239: (8)                    self._original_edgecolor = other._original_edgecolor
240: (8)                    self._original_facecolor = other._original_facecolor
241: (8)                    self._fill = other._fill
242: (8)                    self._hatch = other._hatch
243: (8)                    self._hatch_color = other._hatch_color
244: (8)                    self._unscaled_dash_pattern = other._unscaled_dash_pattern
245: (8)                    self.set_linewidth(other._linewidth)  # also sets scaled dashes
246: (8)                    self.set_transform(other.get_data_transform())
247: (8)                    self._transformSet = other.is_transform_set()
248: (4)                def get_extents(self):
249: (8)                    """
250: (8)                    Return the `Patch`'s axis-aligned extents as a `~.transforms.Bbox`.
251: (8)                    """
252: (8)                    return self.get_path().get_extents(self.get_transform())
253: (4)                def get_transform(self):
254: (8)                    """Return the `~.transforms.Transform` applied to the `Patch`."""
255: (8)                    return self.get_patch_transform() + artist.Artist.get_transform(self)
256: (4)                def get_data_transform(self):
257: (8)                    """
258: (8)                    Return the `~.transforms.Transform` mapping data coordinates to
259: (8)                    physical coordinates.
260: (8)                    """
261: (8)                    return artist.Artist.get_transform(self)
262: (4)                def get_patch_transform(self):
263: (8)                    """
```

```
(8)                 Return the `~.transforms.Transform` instance mapping patch coordinates
(8)                 to data coordinates.
(8)                 For example, one may define a patch of a circle which represents a
(8)                 radius of 5 by providing coordinates for a unit circle, and a
(8)                 transform which scales the coordinates (the patch coordinate) by 5.
(8)                 """
(8)                 return transforms.IdentityTransform()
(4)             def get_antialiased(self):
(8)                 """Return whether antialiasing is used for drawing."""
(8)                 return self._antialiased
(4)             def get_edgecolor(self):
(8)                 """Return the edge color."""
(8)                 return self._edgecolor
(4)             def get_facecolor(self):
(8)                 """Return the face color."""
(8)                 return self._facecolor
(4)             def get_linewidth(self):
(8)                 """Return the line width in points."""
(8)                 return self._linewidth
(4)             def get_linestyle(self):
(8)                 """Return the linestyle."""
(8)                 return self._linestyle
(4)             def set_antialiased(self, aa):
(8)                 """
(8)                 Set whether to use antialiased rendering.
(8)                 Parameters
(8)                 ----------
(8)                 aa : bool or None
(8)                 """
(8)                 if aa is None:
(12)                    aa = mpl.rcParams['patch.antialiased']
(8)                 self._antialiased = aa
(8)                 self.stale = True
(4)             def _set_edgecolor(self, color):
(8)                 set_hatch_color = True
(8)                 if color is None:
(12)                    if (mpl.rcParams['patch.force_edgecolor'] or
(20)                            not self._fill or self._edge_default):
(16)                        color = mpl.rcParams['patch.edgecolor']
(12)                    else:
(16)                        color = 'none'
(16)                        set_hatch_color = False
(8)                 self._edgecolor = colors.to_rgba(color, self._alpha)
(8)                 if set_hatch_color:
(12)                    self._hatch_color = self._edgecolor
(8)                 self.stale = True
(4)             def set_edgecolor(self, color):
(8)                 """
(8)                 Set the patch edge color.
(8)                 Parameters
(8)                 ----------
(8)                 color : :mpltype:`color` or None
(8)                 """
(8)                 self._original_edgecolor = color
(8)                 self._set_edgecolor(color)
(4)             def _set_facecolor(self, color):
(8)                 if color is None:
(12)                    color = mpl.rcParams['patch.facecolor']
(8)                 alpha = self._alpha if self._fill else 0
(8)                 self._facecolor = colors.to_rgba(color, alpha)
(8)                 self.stale = True
(4)             def set_facecolor(self, color):
(8)                 """
(8)                 Set the patch face color.
(8)                 Parameters
(8)                 ----------
(8)                 color : :mpltype:`color` or None
(8)                 """
(8)                 self._original_facecolor = color
```

```
(8)                 self._set_facecolor(color)
(4)             def set_color(self, c):
(8)                 """
(8)                 Set both the edgecolor and the facecolor.
(8)                 Parameters
(8)                 ----------
(8)                 c : :mpltype:`color`
(8)                 See Also
(8)                 --------
(8)                 Patch.set_facecolor, Patch.set_edgecolor
(12)                    For setting the edge or face color individually.
(8)                 """
(8)                 self.set_facecolor(c)
(8)                 self.set_edgecolor(c)
(4)             def set_alpha(self, alpha):
(8)                 super().set_alpha(alpha)
(8)                 self._set_facecolor(self._original_facecolor)
(8)                 self._set_edgecolor(self._original_edgecolor)
(4)             def set_linewidth(self, w):
(8)                 """
(8)                 Set the patch linewidth in points.
(8)                 Parameters
(8)                 ----------
(8)                 w : float or None
(8)                 """
(8)                 if w is None:
(12)                    w = mpl.rcParams['patch.linewidth']
(8)                 self._linewidth = float(w)
(8)                 self._dash_pattern = mlines._scale_dashes(
(12)                    *self._unscaled_dash_pattern, w)
(8)                 self.stale = True
(4)             def set_linestyle(self, ls):
(8)                 """
(8)                 Set the patch linestyle.
(8)                 ==========================================  =================
(8)                 linestyle                                   description
(8)                 ==========================================  =================
(8)                 ``'-'`` or ``'solid'``                      solid line
(8)                 ``'--'`` or  ``'dashed'``                   dashed line
(8)                 ``'-.'`` or  ``'dashdot'``                  dash-dotted line
(8)                 ``':'`` or ``'dotted'``                     dotted line
(8)                 ``'none'``, ``'None'``, ``' '``, or ``''``  draw nothing
(8)                 ==========================================  =================
(8)                 Alternatively a dash tuple of the following form can be provided::
(12)                    (offset, onoffseq)
(8)                 where ``onoffseq`` is an even length tuple of on and off ink in
points.
(8)                 Parameters
(8)                 ----------
(8)                 ls : {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
(12)                    The line style.
(8)                 """
(8)                 if ls is None:
(12)                    ls = "solid"
(8)                 if ls in [' ', '', 'none']:
(12)                    ls = 'None'
(8)                 self._linestyle = ls
(8)                 self._unscaled_dash_pattern = mlines._get_dash_pattern(ls)
(8)                 self._dash_pattern = mlines._scale_dashes(
(12)                    *self._unscaled_dash_pattern, self._linewidth)
(8)                 self.stale = True
(4)             def set_fill(self, b):
(8)                 """
(8)                 Set whether to fill the patch.
(8)                 Parameters
(8)                 ----------
(8)                 b : bool
(8)                 """
(8)                 self._fill = bool(b)
```

```
(8)                 self._set_facecolor(self._original_facecolor)
(8)                 self._set_edgecolor(self._original_edgecolor)
(8)                 self.stale = True
(4)             def get_fill(self):
(8)                 """Return whether the patch is filled."""
(8)                 return self._fill
(4)             fill = property(get_fill, set_fill)
(4)             @_docstring.interpd
(4)             def set_capstyle(self, s):
(8)                 """
(8)                 Set the `.CapStyle`.
(8)                 The default capstyle is 'round' for `.FancyArrowPatch` and 'butt' for
(8)                 all other patches.
(8)                 Parameters
(8)                 ----------
(8)                 s : `.CapStyle` or %(CapStyle)s
(8)                 """
(8)                 cs = CapStyle(s)
(8)                 self._capstyle = cs
(8)                 self.stale = True
(4)             def get_capstyle(self):
(8)                 """Return the capstyle."""
(8)                 return self._capstyle.name
(4)             @_docstring.interpd
(4)             def set_joinstyle(self, s):
(8)                 """
(8)                 Set the `.JoinStyle`.
(8)                 The default joinstyle is 'round' for `.FancyArrowPatch` and 'miter'
for
(8)                 all other patches.
(8)                 Parameters
(8)                 ----------
(8)                 s : `.JoinStyle` or %(JoinStyle)s
(8)                 """
(8)                 js = JoinStyle(s)
(8)                 self._joinstyle = js
(8)                 self.stale = True
(4)             def get_joinstyle(self):
(8)                 """Return the joinstyle."""
(8)                 return self._joinstyle.name
(4)             def set_hatch(self, hatch):
(8)                 r"""
(8)                 Set the hatching pattern.
(8)                 *hatch* can be one of::
(10)                  /   - diagonal hatching
(10)                  \   - back diagonal
(10)                  |   - vertical
(10)                  -   - horizontal
(10)                  +   - crossed
(10)                  x   - crossed diagonal
(10)                  o   - small circle
(10)                  O   - large circle
(10)                  .   - dots
(10)                  *   - stars
(8)                 Letters can be combined, in which case all the specified
(8)                 hatchings are done.  If same letter repeats, it increases the
(8)                 density of hatching of that pattern.
(8)                 Parameters
(8)                 ----------
(8)                 hatch : {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
(8)                 """
(8)                 mhatch._validate_hatch_pattern(hatch)
(8)                 self._hatch = hatch
(8)                 self.stale = True
(4)             def get_hatch(self):
(8)                 """Return the hatching pattern."""
(8)                 return self._hatch
(4)             def _draw_paths_with_artist_properties(
(12)                    self, renderer, draw_path_args_list):
```

```
(8)                 """
(8)                 ``draw()`` helper factored out for sharing with `FancyArrowPatch`.
(8)                 Configure *renderer* and the associated graphics context *gc*
(8)                 from the artist properties, then repeatedly call
(8)                 ``renderer.draw_path(gc, *draw_path_args)`` for each tuple
(8)                 *draw_path_args* in *draw_path_args_list*.
(8)                 """
(8)                 renderer.open_group('patch', self.get_gid())
(8)                 gc = renderer.new_gc()
(8)                 gc.set_foreground(self._edgecolor, isRGBA=True)
(8)                 lw = self._linewidth
(8)                 if self._edgecolor[3] == 0 or self._linestyle == 'None':
(12)                    lw = 0
(8)                 gc.set_linewidth(lw)
(8)                 gc.set_dashes(*self._dash_pattern)
(8)                 gc.set_capstyle(self._capstyle)
(8)                 gc.set_joinstyle(self._joinstyle)
(8)                 gc.set_antialiased(self._antialiased)
(8)                 self._set_gc_clip(gc)
(8)                 gc.set_url(self._url)
(8)                 gc.set_snap(self.get_snap())
(8)                 gc.set_alpha(self._alpha)
(8)                 if self._hatch:
(12)                    gc.set_hatch(self._hatch)
(12)                    gc.set_hatch_color(self._hatch_color)
(8)                 if self.get_sketch_params() is not None:
(12)                    gc.set_sketch_params(*self.get_sketch_params())
(8)                 if self.get_path_effects():
(12)                    from matplotlib.patheffects import PathEffectRenderer
(12)                    renderer = PathEffectRenderer(self.get_path_effects(), renderer)
(8)                 for draw_path_args in draw_path_args_list:
(12)                    renderer.draw_path(gc, *draw_path_args)
(8)                 gc.restore()
(8)                 renderer.close_group('patch')
(8)                 self.stale = False
(4)             @artist.allow_rasterization
(4)             def draw(self, renderer):
(8)                 if not self.get_visible():
(12)                    return
(8)                 path = self.get_path()
(8)                 transform = self.get_transform()
(8)                 tpath = transform.transform_path_non_affine(path)
(8)                 affine = transform.get_affine()
(8)                 self._draw_paths_with_artist_properties(
(12)                    renderer,
(12)                    [(tpath, affine,
(14)                      self._facecolor if self._facecolor[3] else None)])
(4)             def get_path(self):
(8)                 """Return the path of this patch."""
(8)                 raise NotImplementedError('Derived must override')
(4)             def get_window_extent(self, renderer=None):
(8)                 return self.get_path().get_extents(self.get_transform())
(4)             def _convert_xy_units(self, xy):
(8)                 """Convert x and y units for a tuple (x, y)."""
(8)                 x = self.convert_xunits(xy[0])
(8)                 y = self.convert_yunits(xy[1])
(8)                 return x, y
(0)         class Shadow(Patch):
(4)             def __str__(self):
(8)                 return f"Shadow({self.patch})"
(4)             @_docstring.dedent_interpd
(4)             def __init__(self, patch, ox, oy, *, shade=0.7, **kwargs):
(8)                 """
(8)                 Create a shadow of the given *patch*.
(8)                 By default, the shadow will have the same face color as the *patch*,
(8)                 but darkened. The darkness can be controlled by *shade*.
(8)                 Parameters
(8)                 ----------
(8)                 patch : `~matplotlib.patches.Patch`
```

```
538: (12)                       The patch to create the shadow for.
539: (8)                    ox, oy : float
540: (12)                       The shift of the shadow in data coordinates, scaled by a factor
541: (12)                       of dpi/72.
542: (8)                    shade : float, default: 0.7
543: (12)                       How the darkness of the shadow relates to the original color. If
1, the
544: (12)                       shadow is black, if 0, the shadow has the same color as the
*patch*.
545: (12)                       .. versionadded:: 3.8
546: (8)                    **kwargs
547: (12)                       Properties of the shadow patch. Supported keys are:
548: (12)                       %(Patch:kwdoc)s
549: (8)                    """
550: (8)                    super().__init__()
551: (8)                    self.patch = patch
552: (8)                    self._ox, self._oy = ox, oy
553: (8)                    self._shadow_transform = transforms.Affine2D()
554: (8)                    self.update_from(self.patch)
555: (8)                    if not 0 <= shade <= 1:
556: (12)                       raise ValueError("shade must be between 0 and 1.")
557: (8)                    color = (1 - shade) *
np.asarray(colors.to_rgb(self.patch.get_facecolor()))
558: (8)                    self.update({'facecolor': color, 'edgecolor': color, 'alpha': 0.5,
559: (21)                               'zorder': np.nextafter(self.patch.zorder, -np.inf),
560: (21)                               **kwargs})
561: (4)                def _update_transform(self, renderer):
562: (8)                    ox = renderer.points_to_pixels(self._ox)
563: (8)                    oy = renderer.points_to_pixels(self._oy)
564: (8)                    self._shadow_transform.clear().translate(ox, oy)
565: (4)                def get_path(self):
566: (8)                    return self.patch.get_path()
567: (4)                def get_patch_transform(self):
568: (8)                    return self.patch.get_patch_transform() + self._shadow_transform
569: (4)                def draw(self, renderer):
570: (8)                    self._update_transform(renderer)
571: (8)                    super().draw(renderer)
572: (0)            class Rectangle(Patch):
573: (4)                """
574: (4)                A rectangle defined via an anchor point *xy* and its *width* and *height*.
575: (4)                The rectangle extends from ``xy[0]`` to ``xy[0] + width`` in x-direction
576: (4)                and from ``xy[1]`` to ``xy[1] + height`` in y-direction. ::
577: (6)                  :               +------------------+
578: (6)                  :               |                  |
579: (6)                  :             height               |
580: (6)                  :               |                  |
581: (6)                  :              (xy)---- width -----+
582: (4)                One may picture *xy* as the bottom left corner, but which corner *xy* is
583: (4)                actually depends on the direction of the axis and the sign of *width*
584: (4)                and *height*; e.g. *xy* would be the bottom right corner if the x-axis
585: (4)                was inverted or if *width* was negative.
586: (4)                """
587: (4)                def __str__(self):
588: (8)                    pars = self._x0, self._y0, self._width, self._height, self.angle
589: (8)                    fmt = "Rectangle(xy=(%g, %g), width=%g, height=%g, angle=%g)"
590: (8)                    return fmt % pars
591: (4)                @_docstring.dedent_interpd
592: (4)                def __init__(self, xy, width, height, *,
593: (17)                            angle=0.0, rotation_point='xy', **kwargs):
594: (8)                    """
595: (8)                    Parameters
596: (8)                    ----------
597: (8)                    xy : (float, float)
598: (12)                       The anchor point.
599: (8)                    width : float
600: (12)                       Rectangle width.
601: (8)                    height : float
602: (12)                       Rectangle height.
603: (8)                    angle : float, default: 0
```

```
604: (12)                        Rotation in degrees anti-clockwise about the rotation point.
605: (8)                     rotation_point : {'xy', 'center', (number, number)}, default: 'xy'
606: (12)                        If ``'xy'``, rotate around the anchor point. If ``'center'``
rotate
607: (12)                        around the center. If 2-tuple of number, rotate around this
608: (12)                        coordinate.
609: (8)                     Other Parameters
610: (8)                     ----------------
611: (8)                     **kwargs : `~matplotlib.patches.Patch` properties
612: (12)                        %(Patch:kwdoc)s
613: (8)                     """
614: (8)                     super().__init__(**kwargs)
615: (8)                     self._x0 = xy[0]
616: (8)                     self._y0 = xy[1]
617: (8)                     self._width = width
618: (8)                     self._height = height
619: (8)                     self.angle = float(angle)
620: (8)                     self.rotation_point = rotation_point
621: (8)                     self._aspect_ratio_correction = 1.0
622: (8)                     self._convert_units()  # Validate the inputs.
623: (4)                 def get_path(self):
624: (8)                     """Return the vertices of the rectangle."""
625: (8)                     return Path.unit_rectangle()
626: (4)                 def _convert_units(self):
627: (8)                     """Convert bounds of the rectangle."""
628: (8)                     x0 = self.convert_xunits(self._x0)
629: (8)                     y0 = self.convert_yunits(self._y0)
630: (8)                     x1 = self.convert_xunits(self._x0 + self._width)
631: (8)                     y1 = self.convert_yunits(self._y0 + self._height)
632: (8)                     return x0, y0, x1, y1
633: (4)                 def get_patch_transform(self):
634: (8)                     bbox = self.get_bbox()
635: (8)                     if self.rotation_point == 'center':
636: (12)                        width, height = bbox.x1 - bbox.x0, bbox.y1 - bbox.y0
637: (12)                        rotation_point = bbox.x0 + width / 2., bbox.y0 + height / 2.
638: (8)                     elif self.rotation_point == 'xy':
639: (12)                        rotation_point = bbox.x0, bbox.y0
640: (8)                     else:
641: (12)                        rotation_point = self.rotation_point
642: (8)                     return transforms.BboxTransformTo(bbox) \
643: (16)                            + transforms.Affine2D() \
644: (16)                            .translate(-rotation_point[0], -rotation_point[1]) \
645: (16)                            .scale(1, self._aspect_ratio_correction) \
646: (16)                            .rotate_deg(self.angle) \
647: (16)                            .scale(1, 1 / self._aspect_ratio_correction) \
648: (16)                            .translate(*rotation_point)
649: (4)                 @property
650: (4)                 def rotation_point(self):
651: (8)                     """The rotation point of the patch."""
652: (8)                     return self._rotation_point
653: (4)                 @rotation_point.setter
654: (4)                 def rotation_point(self, value):
655: (8)                     if value in ['center', 'xy'] or (
656: (16)                            isinstance(value, tuple) and len(value) == 2 and
657: (16)                            isinstance(value[0], Real) and isinstance(value[1], Real)
658: (16)                            ):
659: (12)                        self._rotation_point = value
660: (8)                     else:
661: (12)                        raise ValueError("`rotation_point` must be one of "
662: (29)                                         "{'xy', 'center', (number, number)}.")
663: (4)                 def get_x(self):
664: (8)                     """Return the left coordinate of the rectangle."""
665: (8)                     return self._x0
666: (4)                 def get_y(self):
667: (8)                     """Return the bottom coordinate of the rectangle."""
668: (8)                     return self._y0
669: (4)                 def get_xy(self):
670: (8)                     """Return the left and bottom coords of the rectangle as a tuple."""
671: (8)                     return self._x0, self._y0
```

```
    def get_corners(self):
        """
        Return the corners of the rectangle, moving anti-clockwise from
        (x0, y0).
        """
        return self.get_patch_transform().transform(
            [(0, 0), (1, 0), (1, 1), (0, 1)])
    def get_center(self):
        """Return the centre of the rectangle."""
        return self.get_patch_transform().transform((0.5, 0.5))
    def get_width(self):
        """Return the width of the rectangle."""
        return self._width
    def get_height(self):
        """Return the height of the rectangle."""
        return self._height
    def get_angle(self):
        """Get the rotation angle in degrees."""
        return self.angle
    def set_x(self, x):
        """Set the left coordinate of the rectangle."""
        self._x0 = x
        self.stale = True
    def set_y(self, y):
        """Set the bottom coordinate of the rectangle."""
        self._y0 = y
        self.stale = True
    def set_angle(self, angle):
        """
        Set the rotation angle in degrees.
        The rotation is performed anti-clockwise around *xy*.
        """
        self.angle = angle
        self.stale = True
    def set_xy(self, xy):
        """
        Set the left and bottom coordinates of the rectangle.
        Parameters
        ----------
        xy : (float, float)
        """
        self._x0, self._y0 = xy
        self.stale = True
    def set_width(self, w):
        """Set the width of the rectangle."""
        self._width = w
        self.stale = True
    def set_height(self, h):
        """Set the height of the rectangle."""
        self._height = h
        self.stale = True
    def set_bounds(self, *args):
        """
        Set the bounds of the rectangle as *left*, *bottom*, *width*, *height*.
        The values may be passed as separate parameters or as a tuple::
            set_bounds(left, bottom, width, height)
            set_bounds((left, bottom, width, height))
        .. ACCEPTS: (left, bottom, width, height)
        """
        if len(args) == 1:
            l, b, w, h = args[0]
        else:
            l, b, w, h = args
        self._x0 = l
        self._y0 = b
        self._width = w
        self._height = h
        self.stale = True
```

```
(4)                 def get_bbox(self):
(8)                     """Return the `.Bbox`."""
(8)                     return transforms.Bbox.from_extents(*self._convert_units())
(4)                 xy = property(get_xy, set_xy)
(0)             class RegularPolygon(Patch):
(4)                 """A regular polygon patch."""
(4)                 def __str__(self):
(8)                     s = "RegularPolygon((%g, %g), %d, radius=%g, orientation=%g)"
(8)                     return s % (self.xy[0], self.xy[1], self.numvertices, self.radius,
(20)                                self.orientation)
(4)                 @_docstring.dedent_interpd
(4)                 def __init__(self, xy, numVertices, *,
(17)                             radius=5, orientation=0, **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     xy : (float, float)
(12)                        The center position.
(8)                     numVertices : int
(12)                        The number of vertices.
(8)                     radius : float
(12)                        The distance from the center to each of the vertices.
(8)                     orientation : float
(12)                        The polygon rotation angle (in radians).
(8)                     **kwargs
(12)                        `Patch` properties:
(12)                        %(Patch:kwdoc)s
(8)                     """
(8)                     self.xy = xy
(8)                     self.numvertices = numVertices
(8)                     self.orientation = orientation
(8)                     self.radius = radius
(8)                     self._path = Path.unit_regular_polygon(numVertices)
(8)                     self._patch_transform = transforms.Affine2D()
(8)                     super().__init__(**kwargs)
(4)                 def get_path(self):
(8)                     return self._path
(4)                 def get_patch_transform(self):
(8)                     return self._patch_transform.clear() \
(12)                        .scale(self.radius) \
(12)                        .rotate(self.orientation) \
(12)                        .translate(*self.xy)
(0)             class PathPatch(Patch):
(4)                 """A general polycurve path patch."""
(4)                 _edge_default = True
(4)                 def __str__(self):
(8)                     s = "PathPatch%d((%g, %g) ...)"
(8)                     return s % (len(self._path.vertices), *tuple(self._path.vertices[0]))
(4)                 @_docstring.dedent_interpd
(4)                 def __init__(self, path, **kwargs):
(8)                     """
(8)                     *path* is a `.Path` object.
(8)                     Valid keyword arguments are:
(8)                     %(Patch:kwdoc)s
(8)                     """
(8)                     super().__init__(**kwargs)
(8)                     self._path = path
(4)                 def get_path(self):
(8)                     return self._path
(4)                 def set_path(self, path):
(8)                     self._path = path
(0)             class StepPatch(PathPatch):
(4)                 """
(4)                 A path patch describing a stepwise constant function.
(4)                 By default, the path is not closed and starts and stops at
(4)                 baseline value.
(4)                 """
(4)                 _edge_default = False
(4)                 @_docstring.dedent_interpd
```

```
(4)     def __init__(self, values, edges, *,
(17)                 orientation='vertical', baseline=0, **kwargs):
(8)         """
(8)         Parameters
(8)         ----------
(8)         values : array-like
(12)            The step heights.
(8)         edges : array-like
(12)            The edge positions, with ``len(edges) == len(vals) + 1``,
(12)            between which the curve takes on vals values.
(8)         orientation : {'vertical', 'horizontal'}, default: 'vertical'
(12)            The direction of the steps. Vertical means that *values* are
(12)            along the y-axis, and edges are along the x-axis.
(8)         baseline : float, array-like or None, default: 0
(12)            The bottom value of the bounding edges or when
(12)            ``fill=True``, position of lower edge. If *fill* is
(12)            True or an array is passed to *baseline*, a closed
(12)            path is drawn.
(8)         **kwargs
(12)            `Patch` properties:
(12)            %(Patch:kwdoc)s
(8)         """
(8)         self.orientation = orientation
(8)         self._edges = np.asarray(edges)
(8)         self._values = np.asarray(values)
(8)         self._baseline = np.asarray(baseline) if baseline is not None else
None
(8)         self._update_path()
(8)         super().__init__(self._path, **kwargs)
(4)     def _update_path(self):
(8)         if np.isnan(np.sum(self._edges)):
(12)            raise ValueError('Nan values in "edges" are disallowed')
(8)         if self._edges.size - 1 != self._values.size:
(12)            raise ValueError('Size mismatch between "values" and "edges". '
(29)                             "Expected `len(values) + 1 == len(edges)`, but "
(29)                             f"`len(values) = {self._values.size}` and "
(29)                             f"`len(edges) = {self._edges.size}`.")
(8)         verts, codes = [np.empty((0, 2))], [np.empty(0, dtype=Path.code_type)]
(8)         _nan_mask = np.isnan(self._values)
(8)         if self._baseline is not None:
(12)            _nan_mask |= np.isnan(self._baseline)
(8)         for idx0, idx1 in cbook.contiguous_regions(~_nan_mask):
(12)            x = np.repeat(self._edges[idx0:idx1+1], 2)
(12)            y = np.repeat(self._values[idx0:idx1], 2)
(12)            if self._baseline is None:
(16)                y = np.concatenate([y[:1], y, y[-1:]])
(12)            elif self._baseline.ndim == 0:  # single baseline value
(16)                y = np.concatenate([[self._baseline], y, [self._baseline]])
(12)            elif self._baseline.ndim == 1:  # baseline array
(16)                base = np.repeat(self._baseline[idx0:idx1], 2)[::-1]
(16)                x = np.concatenate([x, x[::-1]])
(16)                y = np.concatenate([base[-1:], y, base[:1],
(36)                                    base[:1], base, base[-1:]])
(12)            else:  # no baseline
(16)                raise ValueError('Invalid `baseline` specified')
(12)            if self.orientation == 'vertical':
(16)                xy = np.column_stack([x, y])
(12)            else:
(16)                xy = np.column_stack([y, x])
(12)            verts.append(xy)
(12)            codes.append([Path.MOVETO] + [Path.LINETO]*(len(xy)-1))
(8)         self._path = Path(np.concatenate(verts), np.concatenate(codes))
(4)     def get_data(self):
(8)         """Get `.StepPatch` values, edges and baseline as namedtuple."""
(8)         StairData = namedtuple('StairData', 'values edges baseline')
(8)         return StairData(self._values, self._edges, self._baseline)
(4)     def set_data(self, values=None, edges=None, baseline=None):
(8)         """
(8)         Set `.StepPatch` values, edges and baseline.
```

```
(8)                     Parameters
(8)                     ----------
(8)                     values : 1D array-like or None
(12)                        Will not update values, if passing None
(8)                     edges : 1D array-like, optional
(8)                     baseline : float, 1D array-like or None
(8)                     """
(8)                     if values is None and edges is None and baseline is None:
(12)                        raise ValueError("Must set *values*, *edges* or *baseline*.")
(8)                     if values is not None:
(12)                        self._values = np.asarray(values)
(8)                     if edges is not None:
(12)                        self._edges = np.asarray(edges)
(8)                     if baseline is not None:
(12)                        self._baseline = np.asarray(baseline)
(8)                     self._update_path()
(8)                     self.stale = True
(0)             class Polygon(Patch):
(4)                 """A general polygon patch."""
(4)                 def __str__(self):
(8)                     if len(self._path.vertices):
(12)                        s = "Polygon%d((%g, %g) ...)"
(12)                        return s % (len(self._path.vertices), *self._path.vertices[0])
(8)                     else:
(12)                        return "Polygon0()"
(4)                 @_docstring.dedent_interpd
(4)                 def __init__(self, xy, *, closed=True, **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     xy : (N, 2) array
(8)                     closed : bool, default: True
(12)                        Whether the polygon is closed (i.e., has identical start and end
(12)                        points).
(8)                     **kwargs
(12)                        %(Patch:kwdoc)s
(8)                     """
(8)                     super().__init__(**kwargs)
(8)                     self._closed = closed
(8)                     self.set_xy(xy)
(4)                 def get_path(self):
(8)                     """Get the `.Path` of the polygon."""
(8)                     return self._path
(4)                 def get_closed(self):
(8)                     """Return whether the polygon is closed."""
(8)                     return self._closed
(4)                 def set_closed(self, closed):
(8)                     """
(8)                     Set whether the polygon is closed.
(8)                     Parameters
(8)                     ----------
(8)                     closed : bool
(12)                        True if the polygon is closed
(8)                     """
(8)                     if self._closed == bool(closed):
(12)                        return
(8)                     self._closed = bool(closed)
(8)                     self.set_xy(self.get_xy())
(8)                     self.stale = True
(4)                 def get_xy(self):
(8)                     """
(8)                     Get the vertices of the path.
(8)                     Returns
(8)                     -------
(8)                     (N, 2) array
(12)                        The coordinates of the vertices.
(8)                     """
(8)                     return self._path.vertices
(4)                 def set_xy(self, xy):
```

```
(8)                     """
(8)                     Set the vertices of the polygon.
(8)                     Parameters
(8)                     ----------
(8)                     xy : (N, 2) array-like
(12)                        The coordinates of the vertices.
(8)                     Notes
(8)                     -----
(8)                     Unlike `.Path`, we do not ignore the last input vertex. If the
(8)                     polygon is meant to be closed, and the last point of the polygon is
not
(8)                     equal to the first, we assume that the user has not explicitly passed
a
(8)                     ``CLOSEPOLY`` vertex, and add it ourselves.
(8)                     """
(8)                     xy = np.asarray(xy)
(8)                     nverts, _ = xy.shape
(8)                     if self._closed:
(12)                        if nverts == 1 or nverts > 1 and (xy[0] != xy[-1]).any():
(16)                            xy = np.concatenate([xy, [xy[0]]])
(8)                     else:
(12)                        if nverts > 2 and (xy[0] == xy[-1]).all():
(16)                            xy = xy[:-1]
(8)                     self._path = Path(xy, closed=self._closed)
(8)                     self.stale = True
(4)                 xy = property(get_xy, set_xy,
(18)                              doc='The vertices of the path as a (N, 2) array.')
(0)             class Wedge(Patch):
(4)                 """Wedge shaped patch."""
(4)                 def __str__(self):
(8)                     pars = (self.center[0], self.center[1], self.r,
(16)                            self.theta1, self.theta2, self.width)
(8)                     fmt = "Wedge(center=(%g, %g), r=%g, theta1=%g, theta2=%g, width=%s)"
(8)                     return fmt % pars
(4)                 @_docstring.dedent_interpd
(4)                 def __init__(self, center, r, theta1, theta2, *, width=None, **kwargs):
(8)                     """
(8)                     A wedge centered at *x*, *y* center with radius *r* that
(8)                     sweeps *theta1* to *theta2* (in degrees).  If *width* is given,
(8)                     then a partial wedge is drawn from inner radius *r* - *width*
(8)                     to outer radius *r*.
(8)                     Valid keyword arguments are:
(8)                     %(Patch:kwdoc)s
(8)                     """
(8)                     super().__init__(**kwargs)
(8)                     self.center = center
(8)                     self.r, self.width = r, width
(8)                     self.theta1, self.theta2 = theta1, theta2
(8)                     self._patch_transform = transforms.IdentityTransform()
(8)                     self._recompute_path()
(4)                 def _recompute_path(self):
(8)                     if abs((self.theta2 - self.theta1) - 360) <= 1e-12:
(12)                        theta1, theta2 = 0, 360
(12)                        connector = Path.MOVETO
(8)                     else:
(12)                        theta1, theta2 = self.theta1, self.theta2
(12)                        connector = Path.LINETO
(8)                     arc = Path.arc(theta1, theta2)
(8)                     if self.width is not None:
(12)                        v1 = arc.vertices
(12)                        v2 = arc.vertices[::-1] * (self.r - self.width) / self.r
(12)                        v = np.concatenate([v1, v2, [(0, 0)]])
(12)                        c = [*arc.codes, connector, *arc.codes[1:], Path.CLOSEPOLY]
(8)                     else:
(12)                        v = np.concatenate([arc.vertices, [(0, 0), (0, 0)]])
(12)                        c = [*arc.codes, connector, Path.CLOSEPOLY]
(8)                     self._path = Path(v * self.r + self.center, c)
(4)                 def set_center(self, center):
(8)                     self._path = None
```

```
(8)                 self.center = center
(8)                 self.stale = True
(4)             def set_radius(self, radius):
(8)                 self._path = None
(8)                 self.r = radius
(8)                 self.stale = True
(4)             def set_theta1(self, theta1):
(8)                 self._path = None
(8)                 self.theta1 = theta1
(8)                 self.stale = True
(4)             def set_theta2(self, theta2):
(8)                 self._path = None
(8)                 self.theta2 = theta2
(8)                 self.stale = True
(4)             def set_width(self, width):
(8)                 self._path = None
(8)                 self.width = width
(8)                 self.stale = True
(4)             def get_path(self):
(8)                 if self._path is None:
(12)                    self._recompute_path()
(8)                 return self._path
(0)         class Arrow(Patch):
(4)             """An arrow patch."""
(4)             def __str__(self):
(8)                 return "Arrow()"
(4)             _path = Path._create_closed([
(8)                 [0.0, 0.1], [0.0, -0.1], [0.8, -0.1], [0.8, -0.3], [1.0, 0.0],
(8)                 [0.8, 0.3], [0.8, 0.1]])
(4)             @_docstring.dedent_interpd
(4)             def __init__(self, x, y, dx, dy, *, width=1.0, **kwargs):
(8)                 """
(8)                 Draws an arrow from (*x*, *y*) to (*x* + *dx*, *y* + *dy*).
(8)                 The width of the arrow is scaled by *width*.
(8)                 Parameters
(8)                 ----------
(8)                 x : float
(12)                    x coordinate of the arrow tail.
(8)                 y : float
(12)                    y coordinate of the arrow tail.
(8)                 dx : float
(12)                    Arrow length in the x direction.
(8)                 dy : float
(12)                    Arrow length in the y direction.
(8)                 width : float, default: 1
(12)                    Scale factor for the width of the arrow. With a default value of
1,
(12)                    the tail width is 0.2 and head width is 0.6.
(8)                 **kwargs
(12)                    Keyword arguments control the `Patch` properties:
(12)                    %(Patch:kwdoc)s
(8)                 See Also
(8)                 --------
(8)                 FancyArrow
(12)                    Patch that allows independent control of the head and tail
(12)                    properties.
(8)                 """
(8)                 super().__init__(**kwargs)
(8)                 self.set_data(x, y, dx, dy, width)
(4)             def get_path(self):
(8)                 return self._path
(4)             def get_patch_transform(self):
(8)                 return self._patch_transform
(4)             def set_data(self, x=None, y=None, dx=None, dy=None, width=None):
(8)                 """
(8)                 Set `.Arrow` x, y, dx, dy and width.
(8)                 Values left as None will not be updated.
(8)                 Parameters
(8)                 ----------
```

```
(8)                 x, y : float or None, default: None
(12)                    The x and y coordinates of the arrow base.
(8)                 dx, dy : float or None, default: None
(12)                    The length of the arrow along x and y direction.
(8)                 width : float or None, default: None
(12)                    Width of full arrow tail.
(8)                 """
(8)                 if x is not None:
(12)                    self._x = x
(8)                 if y is not None:
(12)                    self._y = y
(8)                 if dx is not None:
(12)                    self._dx = dx
(8)                 if dy is not None:
(12)                    self._dy = dy
(8)                 if width is not None:
(12)                    self._width = width
(8)                 self._patch_transform = (
(12)                    transforms.Affine2D()
(12)                    .scale(np.hypot(self._dx, self._dy), self._width)
(12)                    .rotate(np.arctan2(self._dy, self._dx))
(12)                    .translate(self._x, self._y)
(12)                    .frozen())
(0)         class FancyArrow(Polygon):
(4)             """
(4)             Like Arrow, but lets you set head width and head height independently.
(4)             """
(4)             _edge_default = True
(4)             def __str__(self):
(8)                 return "FancyArrow()"
(4)             @_docstring.dedent_interpd
(4)             def __init__(self, x, y, dx, dy, *,
(17)                         width=0.001, length_includes_head=False, head_width=None,
(17)                         head_length=None, shape='full', overhang=0,
(17)                         head_starts_at_zero=False, **kwargs):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 x, y : float
(12)                    The x and y coordinates of the arrow base.
(8)                 dx, dy : float
(12)                    The length of the arrow along x and y direction.
(8)                 width : float, default: 0.001
(12)                    Width of full arrow tail.
(8)                 length_includes_head : bool, default: False
(12)                    True if head is to be counted in calculating the length.
(8)                 head_width : float or None, default: 3*width
(12)                    Total width of the full arrow head.
(8)                 head_length : float or None, default: 1.5*head_width
(12)                    Length of arrow head.
(8)                 shape : {'full', 'left', 'right'}, default: 'full'
(12)                    Draw the left-half, right-half, or full arrow.
(8)                 overhang : float, default: 0
(12)                    Fraction that the arrow is swept back (0 overhang means
(12)                    triangular shape). Can be negative or greater than one.
(8)                 head_starts_at_zero : bool, default: False
(12)                    If True, the head starts being drawn at coordinate 0
(12)                    instead of ending at coordinate 0.
(8)                 **kwargs
(12)                    `.Patch` properties:
(12)                    %(Patch:kwdoc)s
(8)                 """
(8)                 self._x = x
(8)                 self._y = y
(8)                 self._dx = dx
(8)                 self._dy = dy
(8)                 self._width = width
(8)                 self._length_includes_head = length_includes_head
(8)                 self._head_width = head_width
```

```
(8)                 self._head_length = head_length
(8)                 self._shape = shape
(8)                 self._overhang = overhang
(8)                 self._head_starts_at_zero = head_starts_at_zero
(8)                 self._make_verts()
(8)                 super().__init__(self.verts, closed=True, **kwargs)
(4)             def set_data(self, *, x=None, y=None, dx=None, dy=None, width=None,
(17)                     head_width=None, head_length=None):
(8)                 """
(8)                 Set `.FancyArrow` x, y, dx, dy, width, head_with, and head_length.
(8)                 Values left as None will not be updated.
(8)                 Parameters
(8)                 ----------
(8)                 x, y : float or None, default: None
(12)                    The x and y coordinates of the arrow base.
(8)                 dx, dy : float or None, default: None
(12)                    The length of the arrow along x and y direction.
(8)                 width : float or None, default: None
(12)                    Width of full arrow tail.
(8)                 head_width : float or None, default: None
(12)                    Total width of the full arrow head.
(8)                 head_length : float or None, default: None
(12)                    Length of arrow head.
(8)                 """
(8)                 if x is not None:
(12)                    self._x = x
(8)                 if y is not None:
(12)                    self._y = y
(8)                 if dx is not None:
(12)                    self._dx = dx
(8)                 if dy is not None:
(12)                    self._dy = dy
(8)                 if width is not None:
(12)                    self._width = width
(8)                 if head_width is not None:
(12)                    self._head_width = head_width
(8)                 if head_length is not None:
(12)                    self._head_length = head_length
(8)                 self._make_verts()
(8)                 self.set_xy(self.verts)
(4)             def _make_verts(self):
(8)                 if self._head_width is None:
(12)                    head_width = 3 * self._width
(8)                 else:
(12)                    head_width = self._head_width
(8)                 if self._head_length is None:
(12)                    head_length = 1.5 * head_width
(8)                 else:
(12)                    head_length = self._head_length
(8)                 distance = np.hypot(self._dx, self._dy)
(8)                 if self._length_includes_head:
(12)                    length = distance
(8)                 else:
(12)                    length = distance + head_length
(8)                 if not length:
(12)                    self.verts = np.empty([0, 2])  # display nothing if empty
(8)                 else:
(12)                    hw, hl = head_width, head_length
(12)                    hs, lw = self._overhang, self._width
(12)                    left_half_arrow = np.array([
(16)                        [0.0, 0.0],                 # tip
(16)                        [-hl, -hw / 2],             # leftmost
(16)                        [-hl * (1 - hs), -lw / 2],  # meets stem
(16)                        [-length, -lw / 2],         # bottom left
(16)                        [-length, 0],
(12)                    ])
(12)                    if not self._length_includes_head:
(16)                        left_half_arrow += [head_length, 0]
(12)                    if self._head_starts_at_zero:
```

```
(16)                        left_half_arrow += [head_length / 2, 0]
(12)                    if self._shape == 'left':
(16)                        coords = left_half_arrow
(12)                    else:
(16)                        right_half_arrow = left_half_arrow * [1, -1]
(16)                        if self._shape == 'right':
(20)                            coords = right_half_arrow
(16)                        elif self._shape == 'full':
(20)                            coords = np.concatenate([left_half_arrow[:-1],
(45)                                                     right_half_arrow[-2::-1]])
(16)                        else:
(20)                            raise ValueError(f"Got unknown shape: {self._shape!r}")
(12)                    if distance != 0:
(16)                        cx = self._dx / distance
(16)                        sx = self._dy / distance
(12)                    else:
(16)                        cx, sx = 0, 1
(12)                    M = [[cx, sx], [-sx, cx]]
(12)                    self.verts = np.dot(coords, M) + [
(16)                        self._x + self._dx,
(16)                        self._y + self._dy,
(12)                    ]
(0)         _docstring.interpd.update(
(4)             FancyArrow="\n".join(
(8)                 (inspect.getdoc(FancyArrow.__init__) or "").splitlines()[2:]))
(0)         class CirclePolygon(RegularPolygon):
(4)             """A polygon-approximation of a circle patch."""
(4)             def __str__(self):
(8)                 s = "CirclePolygon((%g, %g), radius=%g, resolution=%d)"
(8)                 return s % (self.xy[0], self.xy[1], self.radius, self.numvertices)
(4)             @_docstring.dedent_interpd
(4)             def __init__(self, xy, radius=5, *,
(17)                         resolution=20,  # the number of vertices
(17)                         ** kwargs):
(8)                 """
(8)                 Create a circle at *xy* = (*x*, *y*) with given *radius*.
(8)                 This circle is approximated by a regular polygon with *resolution*
(8)                 sides.  For a smoother circle drawn with splines, see `Circle`.
(8)                 Valid keyword arguments are:
(8)                 %(Patch:kwdoc)s
(8)                 """
(8)                 super().__init__(
(12)                    xy, resolution, radius=radius, orientation=0, **kwargs)
(0)         class Ellipse(Patch):
(4)             """A scale-free ellipse."""
(4)             def __str__(self):
(8)                 pars = (self._center[0], self._center[1],
(16)                        self.width, self.height, self.angle)
(8)                 fmt = "Ellipse(xy=(%s, %s), width=%s, height=%s, angle=%s)"
(8)                 return fmt % pars
(4)             @_docstring.dedent_interpd
(4)             def __init__(self, xy, width, height, *, angle=0, **kwargs):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 xy : (float, float)
(12)                    xy coordinates of ellipse centre.
(8)                 width : float
(12)                    Total length (diameter) of horizontal axis.
(8)                 height : float
(12)                    Total length (diameter) of vertical axis.
(8)                 angle : float, default: 0
(12)                    Rotation in degrees anti-clockwise.
(8)                 Notes
(8)                 -----
(8)                 Valid keyword arguments are:
(8)                 %(Patch:kwdoc)s
(8)                 """
(8)                 super().__init__(**kwargs)
```

```
(8)                 self._center = xy
(8)                 self._width, self._height = width, height
(8)                 self._angle = angle
(8)                 self._path = Path.unit_circle()
(8)                 self._aspect_ratio_correction = 1.0
(8)                 self._patch_transform = transforms.IdentityTransform()
(4)             def _recompute_transform(self):
(8)                 """
(8)                 Notes
(8)                 -----
(8)                 This cannot be called until after this has been added to an Axes,
(8)                 otherwise unit conversion will fail. This makes it very important to
(8)                 call the accessor method and not directly access the transformation
(8)                 member variable.
(8)                 """
(8)                 center = (self.convert_xunits(self._center[0]),
(18)                          self.convert_yunits(self._center[1]))
(8)                 width = self.convert_xunits(self._width)
(8)                 height = self.convert_yunits(self._height)
(8)                 self._patch_transform = transforms.Affine2D() \
(12)                    .scale(width * 0.5, height * 0.5 * self._aspect_ratio_correction)
\
(12)                    .rotate_deg(self.angle) \
(12)                    .scale(1, 1 / self._aspect_ratio_correction) \
(12)                    .translate(*center)
(4)             def get_path(self):
(8)                 """Return the path of the ellipse."""
(8)                 return self._path
(4)             def get_patch_transform(self):
(8)                 self._recompute_transform()
(8)                 return self._patch_transform
(4)             def set_center(self, xy):
(8)                 """
(8)                 Set the center of the ellipse.
(8)                 Parameters
(8)                 ----------
(8)                 xy : (float, float)
(8)                 """
(8)                 self._center = xy
(8)                 self.stale = True
(4)             def get_center(self):
(8)                 """Return the center of the ellipse."""
(8)                 return self._center
(4)             center = property(get_center, set_center)
(4)             def set_width(self, width):
(8)                 """
(8)                 Set the width of the ellipse.
(8)                 Parameters
(8)                 ----------
(8)                 width : float
(8)                 """
(8)                 self._width = width
(8)                 self.stale = True
(4)             def get_width(self):
(8)                 """
(8)                 Return the width of the ellipse.
(8)                 """
(8)                 return self._width
(4)             width = property(get_width, set_width)
(4)             def set_height(self, height):
(8)                 """
(8)                 Set the height of the ellipse.
(8)                 Parameters
(8)                 ----------
(8)                 height : float
(8)                 """
(8)                 self._height = height
(8)                 self.stale = True
(4)             def get_height(self):
```

```
1356: (8)                    """Return the height of the ellipse."""
1357: (8)                    return self._height
1358: (4)                height = property(get_height, set_height)
1359: (4)                def set_angle(self, angle):
1360: (8)                    """
1361: (8)                    Set the angle of the ellipse.
1362: (8)                    Parameters
1363: (8)                    ----------
1364: (8)                    angle : float
1365: (8)                    """
1366: (8)                    self._angle = angle
1367: (8)                    self.stale = True
1368: (4)                def get_angle(self):
1369: (8)                    """Return the angle of the ellipse."""
1370: (8)                    return self._angle
1371: (4)                angle = property(get_angle, set_angle)
1372: (4)                def get_corners(self):
1373: (8)                    """
1374: (8)                    Return the corners of the ellipse bounding box.
1375: (8)                    The bounding box orientation is moving anti-clockwise from the
1376: (8)                    lower left corner defined before rotation.
1377: (8)                    """
1378: (8)                    return self.get_patch_transform().transform(
1379: (12)                       [(-1, -1), (1, -1), (1, 1), (-1, 1)])
1380: (4)                def get_vertices(self):
1381: (8)                    """
1382: (8)                    Return the vertices coordinates of the ellipse.
1383: (8)                    The definition can be found `here
<https://en.wikipedia.org/wiki/Ellipse>`_
1384: (8)                    .. versionadded:: 3.8
1385: (8)                    """
1386: (8)                    if self.width < self.height:
1387: (12)                       ret = self.get_patch_transform().transform([(0, 1), (0, -1)])
1388: (8)                    else:
1389: (12)                       ret = self.get_patch_transform().transform([(1, 0), (-1, 0)])
1390: (8)                    return [tuple(x) for x in ret]
1391: (4)                def get_co_vertices(self):
1392: (8)                    """
1393: (8)                    Return the co-vertices coordinates of the ellipse.
1394: (8)                    The definition can be found `here
<https://en.wikipedia.org/wiki/Ellipse>`_
1395: (8)                    .. versionadded:: 3.8
1396: (8)                    """
1397: (8)                    if self.width < self.height:
1398: (12)                       ret = self.get_patch_transform().transform([(1, 0), (-1, 0)])
1399: (8)                    else:
1400: (12)                       ret = self.get_patch_transform().transform([(0, 1), (0, -1)])
1401: (8)                    return [tuple(x) for x in ret]
1402: (0)            class Annulus(Patch):
1403: (4)                """
1404: (4)                An elliptical annulus.
1405: (4)                """
1406: (4)                @_docstring.dedent_interpd
1407: (4)                def __init__(self, xy, r, width, angle=0.0, **kwargs):
1408: (8)                    """
1409: (8)                    Parameters
1410: (8)                    ----------
1411: (8)                    xy : (float, float)
1412: (12)                       xy coordinates of annulus centre.
1413: (8)                    r : float or (float, float)
1414: (12)                       The radius, or semi-axes:
1415: (12)                       - If float: radius of the outer circle.
1416: (12)                       - If two floats: semi-major and -minor axes of outer ellipse.
1417: (8)                    width : float
1418: (12)                       Width (thickness) of the annular ring. The width is measured
inward
1419: (12)                       from the outer ellipse so that for the inner ellipse the semi-axes
1420: (12)                       are given by ``r - width``. *width* must be less than or equal to
1421: (12)                       the semi-minor axis.
```

```
(8)                 angle : float, default: 0
(12)                    Rotation angle in degrees (anti-clockwise from the positive
(12)                    x-axis). Ignored for circular annuli (i.e., if *r* is a scalar).
(8)                 **kwargs
(12)                    Keyword arguments control the `Patch` properties:
(12)                    %(Patch:kwdoc)s
(8)                 """
(8)                 super().__init__(**kwargs)
(8)                 self.set_radii(r)
(8)                 self.center = xy
(8)                 self.width = width
(8)                 self.angle = angle
(8)                 self._path = None
(4)             def __str__(self):
(8)                 if self.a == self.b:
(12)                    r = self.a
(8)                 else:
(12)                    r = (self.a, self.b)
(8)                 return "Annulus(xy=(%s, %s), r=%s, width=%s, angle=%s)" % \
(16)                        (*self.center, r, self.width, self.angle)
(4)             def set_center(self, xy):
(8)                 """
(8)                 Set the center of the annulus.
(8)                 Parameters
(8)                 ----------
(8)                 xy : (float, float)
(8)                 """
(8)                 self._center = xy
(8)                 self._path = None
(8)                 self.stale = True
(4)             def get_center(self):
(8)                 """Return the center of the annulus."""
(8)                 return self._center
(4)             center = property(get_center, set_center)
(4)             def set_width(self, width):
(8)                 """
(8)                 Set the width (thickness) of the annulus ring.
(8)                 The width is measured inwards from the outer ellipse.
(8)                 Parameters
(8)                 ----------
(8)                 width : float
(8)                 """
(8)                 if width > min(self.a, self.b):
(12)                    raise ValueError(
(16)                        'Width of annulus must be less than or equal to semi-minor
axis')
(8)                 self._width = width
(8)                 self._path = None
(8)                 self.stale = True
(4)             def get_width(self):
(8)                 """Return the width (thickness) of the annulus ring."""
(8)                 return self._width
(4)             width = property(get_width, set_width)
(4)             def set_angle(self, angle):
(8)                 """
(8)                 Set the tilt angle of the annulus.
(8)                 Parameters
(8)                 ----------
(8)                 angle : float
(8)                 """
(8)                 self._angle = angle
(8)                 self._path = None
(8)                 self.stale = True
(4)             def get_angle(self):
(8)                 """Return the angle of the annulus."""
(8)                 return self._angle
(4)             angle = property(get_angle, set_angle)
(4)             def set_semimajor(self, a):
(8)                 """
```

```
(8)                 Set the semi-major axis *a* of the annulus.
(8)                 Parameters
(8)                 ----------
(8)                 a : float
(8)                 """
(8)                 self.a = float(a)
(8)                 self._path = None
(8)                 self.stale = True
(4)             def set_semiminor(self, b):
(8)                 """
(8)                 Set the semi-minor axis *b* of the annulus.
(8)                 Parameters
(8)                 ----------
(8)                 b : float
(8)                 """
(8)                 self.b = float(b)
(8)                 self._path = None
(8)                 self.stale = True
(4)             def set_radii(self, r):
(8)                 """
(8)                 Set the semi-major (*a*) and semi-minor radii (*b*) of the annulus.
(8)                 Parameters
(8)                 ----------
(8)                 r : float or (float, float)
(12)                    The radius, or semi-axes:
(12)                    - If float: radius of the outer circle.
(12)                    - If two floats: semi-major and -minor axes of outer ellipse.
(8)                 """
(8)                 if np.shape(r) == (2,):
(12)                    self.a, self.b = r
(8)                 elif np.shape(r) == ():
(12)                    self.a = self.b = float(r)
(8)                 else:
(12)                    raise ValueError("Parameter 'r' must be one or two floats.")
(8)                 self._path = None
(8)                 self.stale = True
(4)             def get_radii(self):
(8)                 """Return the semi-major and semi-minor radii of the annulus."""
(8)                 return self.a, self.b
(4)             radii = property(get_radii, set_radii)
(4)             def _transform_verts(self, verts, a, b):
(8)                 return transforms.Affine2D() \
(12)                    .scale(*self._convert_xy_units((a, b))) \
(12)                    .rotate_deg(self.angle) \
(12)                    .translate(*self._convert_xy_units(self.center)) \
(12)                    .transform(verts)
(4)             def _recompute_path(self):
(8)                 arc = Path.arc(0, 360)
(8)                 a, b, w = self.a, self.b, self.width
(8)                 v1 = self._transform_verts(arc.vertices, a, b)
(8)                 v2 = self._transform_verts(arc.vertices[::-1], a - w, b - w)
(8)                 v = np.vstack([v1, v2, v1[0, :], (0, 0)])
(8)                 c = np.hstack([arc.codes, Path.MOVETO,
(23)                               arc.codes[1:], Path.MOVETO,
(23)                               Path.CLOSEPOLY])
(8)                 self._path = Path(v, c)
(4)             def get_path(self):
(8)                 if self._path is None:
(12)                    self._recompute_path()
(8)                 return self._path
(0)         class Circle(Ellipse):
(4)             """
(4)             A circle patch.
(4)             """
(4)             def __str__(self):
(8)                 pars = self.center[0], self.center[1], self.radius
(8)                 fmt = "Circle(xy=(%g, %g), radius=%g)"
(8)                 return fmt % pars
(4)             @_docstring.dedent_interpd
```

```
(4)             def __init__(self, xy, radius=5, **kwargs):
(8)                 """
(8)                 Create a true circle at center *xy* = (*x*, *y*) with given *radius*.
(8)                 Unlike `CirclePolygon` which is a polygonal approximation, this uses
(8)                 Bezier splines and is much closer to a scale-free circle.
(8)                 Valid keyword arguments are:
(8)                 %(Patch:kwdoc)s
(8)                 """
(8)                 super().__init__(xy, radius * 2, radius * 2, **kwargs)
(8)                 self.radius = radius
(4)             def set_radius(self, radius):
(8)                 """
(8)                 Set the radius of the circle.
(8)                 Parameters
(8)                 ----------
(8)                 radius : float
(8)                 """
(8)                 self.width = self.height = 2 * radius
(8)                 self.stale = True
(4)             def get_radius(self):
(8)                 """Return the radius of the circle."""
(8)                 return self.width / 2.
(4)             radius = property(get_radius, set_radius)
(0)         class Arc(Ellipse):
(4)             """
(4)             An elliptical arc, i.e. a segment of an ellipse.
(4)             Due to internal optimizations, the arc cannot be filled.
(4)             """
(4)             def __str__(self):
(8)                 pars = (self.center[0], self.center[1], self.width,
(16)                        self.height, self.angle, self.theta1, self.theta2)
(8)                 fmt = ("Arc(xy=(%g, %g), width=%g, "
(15)                       "height=%g, angle=%g, theta1=%g, theta2=%g)")
(8)                 return fmt % pars
(4)             @_docstring.dedent_interpd
(4)             def __init__(self, xy, width, height, *,
(17)                         angle=0.0, theta1=0.0, theta2=360.0, **kwargs):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 xy : (float, float)
(12)                    The center of the ellipse.
(8)                 width : float
(12)                    The length of the horizontal axis.
(8)                 height : float
(12)                    The length of the vertical axis.
(8)                 angle : float
(12)                    Rotation of the ellipse in degrees (counterclockwise).
(8)                 theta1, theta2 : float, default: 0, 360
(12)                    Starting and ending angles of the arc in degrees. These values
(12)                    are relative to *angle*, e.g. if *angle* = 45 and *theta1* = 90
(12)                    the absolute starting angle is 135.
(12)                    Default *theta1* = 0, *theta2* = 360, i.e. a complete ellipse.
(12)                    The arc is drawn in the counterclockwise direction.
(12)                    Angles greater than or equal to 360, or smaller than 0, are
(12)                    represented by an equivalent angle in the range [0, 360), by
(12)                    taking the input value mod 360.
(8)                 Other Parameters
(8)                 ----------------
(8)                 **kwargs : `~matplotlib.patches.Patch` properties
(12)                    Most `.Patch` properties are supported as keyword arguments,
(12)                    except *fill* and *facecolor* because filling is not supported.
(8)                 %(Patch:kwdoc)s
(8)                 """
(8)                 fill = kwargs.setdefault('fill', False)
(8)                 if fill:
(12)                    raise ValueError("Arc objects cannot be filled")
(8)                 super().__init__(xy, width, height, angle=angle, **kwargs)
(8)                 self.theta1 = theta1
```

```
(8)                 self.theta2 = theta2
(8)                 (self._theta1, self._theta2, self._stretched_width,
(9)                  self._stretched_height) = self._theta_stretch()
(8)                 self._path = Path.arc(self._theta1, self._theta2)
(4)             @artist.allow_rasterization
(4)             def draw(self, renderer):
(8)                 """
(8)                 Draw the arc to the given *renderer*.
(8)                 Notes
(8)                 -----
(8)                 Ellipses are normally drawn using an approximation that uses
(8)                 eight cubic Bezier splines.  The error of this approximation
(8)                 is 1.89818e-6, according to this unverified source:
(10)                  Lancaster, Don.  *Approximating a Circle or an Ellipse Using
(10)                  Four Bezier Cubic Splines.*
(10)                  https://www.tinaja.com/glib/ellipse4.pdf
(8)                 There is a use case where very large ellipses must be drawn
(8)                 with very high accuracy, and it is too expensive to render the
(8)                 entire ellipse with enough segments (either splines or line
(8)                 segments).  Therefore, in the case where either radius of the
(8)                 ellipse is large enough that the error of the spline
(8)                 approximation will be visible (greater than one pixel offset
(8)                 from the ideal), a different technique is used.
(8)                 In that case, only the visible parts of the ellipse are drawn,
(8)                 with each visible arc using a fixed number of spline segments
(8)                 (8).  The algorithm proceeds as follows:
(8)                 1. The points where the ellipse intersects the axes (or figure)
(11)                   bounding box are located.  (This is done by performing an inverse
(11)                   transformation on the bbox such that it is relative to the unit
(11)                   circle -- this makes the intersection calculation much easier than
(11)                   doing rotated ellipse intersection directly.)
(11)                   This uses the "line intersecting a circle" algorithm from:
(15)                       Vince, John.  *Geometry for Computer Graphics: Formulae,
(15)                       Examples & Proofs.*  London: Springer-Verlag, 2005.
(8)                 2. The angles of each of the intersection points are calculated.
(8)                 3. Proceeding counterclockwise starting in the positive
(11)                   x-direction, each of the visible arc-segments between the
(11)                   pairs of vertices are drawn using the Bezier arc
(11)                   approximation technique implemented in `.Path.arc`.
(8)                 """
(8)                 if not self.get_visible():
(12)                    return
(8)                 self._recompute_transform()
(8)                 self._update_path()
(8)                 data_to_screen_trans = self.get_data_transform()
(8)                 pwidth, pheight = (
(12)                    data_to_screen_trans.transform((self._stretched_width,
(44)                                                    self._stretched_height)) -
(12)                    data_to_screen_trans.transform((0, 0)))
(8)                 inv_error = (1.0 / 1.89818e-6) * 0.5
(8)                 if pwidth < inv_error and pheight < inv_error:
(12)                    return Patch.draw(self, renderer)
(8)                 def line_circle_intersect(x0, y0, x1, y1):
(12)                    dx = x1 - x0
(12)                    dy = y1 - y0
(12)                    dr2 = dx * dx + dy * dy
(12)                    D = x0 * y1 - x1 * y0
(12)                    D2 = D * D
(12)                    discrim = dr2 - D2
(12)                    if discrim >= 0.0:
(16)                        sign_dy = np.copysign(1, dy)  # +/-1, never 0.
(16)                        sqrt_discrim = np.sqrt(discrim)
(16)                        return np.array(
(20)                            [[(D * dy + sign_dy * dx * sqrt_discrim) / dr2,
(22)                              (-D * dx + abs(dy) * sqrt_discrim) / dr2],
(21)                             [(D * dy - sign_dy * dx * sqrt_discrim) / dr2,
(22)                              (-D * dx - abs(dy) * sqrt_discrim) / dr2]])
(12)                    else:
(16)                        return np.empty((0, 2))
```

```
(8)                 def segment_circle_intersect(x0, y0, x1, y1):
(12)                    epsilon = 1e-9
(12)                    if x1 < x0:
(16)                        x0e, x1e = x1, x0
(12)                    else:
(16)                        x0e, x1e = x0, x1
(12)                    if y1 < y0:
(16)                        y0e, y1e = y1, y0
(12)                    else:
(16)                        y0e, y1e = y0, y1
(12)                    xys = line_circle_intersect(x0, y0, x1, y1)
(12)                    xs, ys = xys.T
(12)                    return xys[
(16)                        (x0e - epsilon < xs) & (xs < x1e + epsilon)
(16)                        & (y0e - epsilon < ys) & (ys < y1e + epsilon)
(12)                    ]
(8)                 box_path_transform = (
(12)                    transforms.BboxTransformTo((self.axes or self.figure).bbox)
(12)                    - self.get_transform())
(8)                 box_path = Path.unit_rectangle().transformed(box_path_transform)
(8)                 thetas = set()
(8)                 for p0, p1 in zip(box_path.vertices[:-1], box_path.vertices[1:]):
(12)                    xy = segment_circle_intersect(*p0, *p1)
(12)                    x, y = xy.T
(12)                    theta = (np.rad2deg(np.arctan2(y, x)) + 360) % 360
(12)                    thetas.update(
(16)                        theta[(self._theta1 < theta) & (theta < self._theta2)])
(8)                 thetas = sorted(thetas) + [self._theta2]
(8)                 last_theta = self._theta1
(8)                 theta1_rad = np.deg2rad(self._theta1)
(8)                 inside = box_path.contains_point(
(12)                    (np.cos(theta1_rad), np.sin(theta1_rad))
(8)                 )
(8)                 path_original = self._path
(8)                 for theta in thetas:
(12)                    if inside:
(16)                        self._path = Path.arc(last_theta, theta, 8)
(16)                        Patch.draw(self, renderer)
(16)                        inside = False
(12)                    else:
(16)                        inside = True
(12)                    last_theta = theta
(8)                 self._path = path_original
(4)             def _update_path(self):
(8)                 stretched = self._theta_stretch()
(8)                 if any(a != b for a, b in zip(
(16)                        stretched, (self._theta1, self._theta2, self._stretched_width,
(28)                                    self._stretched_height))):
(12)                    (self._theta1, self._theta2, self._stretched_width,
(13)                     self._stretched_height) = stretched
(12)                    self._path = Path.arc(self._theta1, self._theta2)
(4)             def _theta_stretch(self):
(8)                 def theta_stretch(theta, scale):
(12)                    theta = np.deg2rad(theta)
(12)                    x = np.cos(theta)
(12)                    y = np.sin(theta)
(12)                    stheta = np.rad2deg(np.arctan2(scale * y, x))
(12)                    return (stheta + 360) % 360
(8)                 width = self.convert_xunits(self.width)
(8)                 height = self.convert_yunits(self.height)
(8)                 if (
(12)                    width != height
(12)                    and not (self.theta1 != self.theta2 and
(21)                             self.theta1 % 360 == self.theta2 % 360)
(8)                 ):
(12)                    theta1 = theta_stretch(self.theta1, width / height)
(12)                    theta2 = theta_stretch(self.theta2, width / height)
(12)                    return theta1, theta2, width, height
(8)                 return self.theta1, self.theta2, width, height
```

```
(0)  def bbox_artist(artist, renderer, props=None, fill=True):
(4)      """
(4)      A debug function to draw a rectangle around the bounding
(4)      box returned by an artist's `.Artist.get_window_extent`
(4)      to test whether the artist is returning the correct bbox.
(4)      *props* is a dict of rectangle props with the additional property
(4)      'pad' that sets the padding around the bbox in points.
(4)      """
(4)      if props is None:
(8)          props = {}
(4)      props = props.copy()  # don't want to alter the pad externally
(4)      pad = props.pop('pad', 4)
(4)      pad = renderer.points_to_pixels(pad)
(4)      bbox = artist.get_window_extent(renderer)
(4)      r = Rectangle(
(8)          xy=(bbox.x0 - pad / 2, bbox.y0 - pad / 2),
(8)          width=bbox.width + pad, height=bbox.height + pad,
(8)          fill=fill, transform=transforms.IdentityTransform(), clip_on=False)
(4)      r.update(props)
(4)      r.draw(renderer)
(0)  def draw_bbox(bbox, renderer, color='k', trans=None):
(4)      """
(4)      A debug function to draw a rectangle around the bounding
(4)      box returned by an artist's `.Artist.get_window_extent`
(4)      to test whether the artist is returning the correct bbox.
(4)      """
(4)      r = Rectangle(xy=bbox.p0, width=bbox.width, height=bbox.height,
(18)                   edgecolor=color, fill=False, clip_on=False)
(4)      if trans is not None:
(8)          r.set_transform(trans)
(4)      r.draw(renderer)
(0)  class _Style:
(4)      """
(4)      A base class for the Styles. It is meant to be a container class,
(4)      where actual styles are declared as subclass of it, and it
(4)      provides some helper functions.
(4)      """
(4)      def __init_subclass__(cls):
(8)          _docstring.interpd.update({
(12)             f"{cls.__name__}:table": cls.pprint_styles(),
(12)             f"{cls.__name__}:table_and_accepts": (
(16)                 cls.pprint_styles()
(16)                 + "\n\n    .. ACCEPTS: ["
(16)                 + "|".join(map(" '{}' ".format, cls._style_list))
(16)                 + "]")
(8)          })
(4)      def __new__(cls, stylename, **kwargs):
(8)          """Return the instance of the subclass with the given style name."""
(8)          _list = stylename.replace(" ", "").split(",")
(8)          _name = _list[0].lower()
(8)          try:
(12)             _cls = cls._style_list[_name]
(8)          except KeyError as err:
(12)             raise ValueError(f"Unknown style: {stylename!r}") from err
(8)          try:
(12)             _args_pair = [cs.split("=") for cs in _list[1:]]
(12)             _args = {k: float(v) for k, v in _args_pair}
(8)          except ValueError as err:
(12)             raise ValueError(
(16)                 f"Incorrect style argument: {stylename!r}") from err
(8)          return _cls(**{**_args, **kwargs})
(4)      @classmethod
(4)      def get_styles(cls):
(8)          """Return a dictionary of available styles."""
(8)          return cls._style_list
(4)      @classmethod
(4)      def pprint_styles(cls):
(8)          """Return the available styles as pretty-printed string."""
(8)          table = [('Class', 'Name', 'Attrs'),
```

```
(17)                            *[(cls.__name__,
(20)                               f'``{name}``',
(20)                               str(inspect.signature(cls))[1:-1] or 'None')
(19)                             for name, cls in cls._style_list.items()]]
(8)                     col_len = [max(len(cell) for cell in column) for column in zip(*table)]
(8)                     table_formatstr = '  '.join('=' * cl for cl in col_len)
(8)                     rst_table = '\n'.join([
(12)                        '',
(12)                        table_formatstr,
(12)                        '  '.join(cell.ljust(cl) for cell, cl in zip(table[0], col_len)),
(12)                        table_formatstr,
(12)                        *['  '.join(cell.ljust(cl) for cell, cl in zip(row, col_len))
(14)                          for row in table[1:]],
(12)                        table_formatstr,
(8)                     ])
(8)                     return textwrap.indent(rst_table, prefix=' ' * 4)
(4)                 @classmethod
(4)                 def register(cls, name, style):
(8)                     """Register a new style."""
(8)                     if not issubclass(style, cls._Base):
(12)                        raise ValueError(f"{style} must be a subclass of {cls._Base}")
(8)                     cls._style_list[name] = style
(0)             def _register_style(style_list, cls=None, *, name=None):
(4)                 """Class decorator that stashes a class in a (style) dictionary."""
(4)                 if cls is None:
(8)                     return functools.partial(_register_style, style_list, name=name)
(4)                 style_list[name or cls.__name__.lower()] = cls
(4)                 return cls
(0)             @_docstring.dedent_interpd
(0)             class BoxStyle(_Style):
(4)                 """
(4)                 `BoxStyle` is a container class which defines several
(4)                 boxstyle classes, which are used for `FancyBboxPatch`.
(4)                 A style object can be created as::
(11)                       BoxStyle.Round(pad=0.2)
(4)                 or::
(11)                       BoxStyle("Round", pad=0.2)
(4)                 or::
(11)                       BoxStyle("Round, pad=0.2")
(4)                 The following boxstyle classes are defined.
(4)                 %(BoxStyle:table)s
(4)                 An instance of a boxstyle class is a callable object, with the signature ::
(7)                    __call__(self, x0, y0, width, height, mutation_size) -> Path
(4)                 *x0*, *y0*, *width* and *height* specify the location and size of the box
(4)                 to be drawn; *mutation_size* scales the outline properties such as padding.
(4)                 """
(4)                 _style_list = {}
(4)                 @_register_style(_style_list)
(4)                 class Square:
(8)                     """A square box."""
(8)                     def __init__(self, pad=0.3):
(12)                        """
(12)                        Parameters
(12)                        ----------
(12)                        pad : float, default: 0.3
(16)                            The amount of padding around the original box.
(12)                        """
(12)                        self.pad = pad
(8)                     def __call__(self, x0, y0, width, height, mutation_size):
(12)                        pad = mutation_size * self.pad
(12)                        width, height = width + 2 * pad, height + 2 * pad
(12)                        x0, y0 = x0 - pad, y0 - pad
(12)                        x1, y1 = x0 + width, y0 + height
(12)                        return Path._create_closed(
(16)                            [(x0, y0), (x1, y0), (x1, y1), (x0, y1)])
(4)                 @_register_style(_style_list)
```

```
(4)    class Circle:
(8)        """A circular box."""
(8)        def __init__(self, pad=0.3):
(12)           """
(12)           Parameters
(12)           ----------
(12)           pad : float, default: 0.3
(16)               The amount of padding around the original box.
(12)           """
(12)           self.pad = pad
(8)        def __call__(self, x0, y0, width, height, mutation_size):
(12)           pad = mutation_size * self.pad
(12)           width, height = width + 2 * pad, height + 2 * pad
(12)           x0, y0 = x0 - pad, y0 - pad
(12)           return Path.circle((x0 + width / 2, y0 + height / 2),
(32)                              max(width, height) / 2)
(4)    @_register_style(_style_list)
(4)    class Ellipse:
(8)        """
(8)        An elliptical box.
(8)        .. versionadded:: 3.7
(8)        """
(8)        def __init__(self, pad=0.3):
(12)           """
(12)           Parameters
(12)           ----------
(12)           pad : float, default: 0.3
(16)               The amount of padding around the original box.
(12)           """
(12)           self.pad = pad
(8)        def __call__(self, x0, y0, width, height, mutation_size):
(12)           pad = mutation_size * self.pad
(12)           width, height = width + 2 * pad, height + 2 * pad
(12)           x0, y0 = x0 - pad, y0 - pad
(12)           a = width / math.sqrt(2)
(12)           b = height / math.sqrt(2)
(12)           trans = Affine2D().scale(a, b).translate(x0 + width / 2,
(53)                                                    y0 + height / 2)
(12)           return trans.transform_path(Path.unit_circle())
(4)    @_register_style(_style_list)
(4)    class LArrow:
(8)        """A box in the shape of a left-pointing arrow."""
(8)        def __init__(self, pad=0.3):
(12)           """
(12)           Parameters
(12)           ----------
(12)           pad : float, default: 0.3
(16)               The amount of padding around the original box.
(12)           """
(12)           self.pad = pad
(8)        def __call__(self, x0, y0, width, height, mutation_size):
(12)           pad = mutation_size * self.pad
(12)           width, height = width + 2 * pad, height + 2 * pad
(12)           x0, y0 = x0 - pad, y0 - pad,
(12)           x1, y1 = x0 + width, y0 + height
(12)           dx = (y1 - y0) / 2
(12)           dxx = dx / 2
(12)           x0 = x0 + pad / 1.4  # adjust by ~sqrt(2)
(12)           return Path._create_closed(
(16)               [(x0 + dxx, y0), (x1, y0), (x1, y1), (x0 + dxx, y1),
(17)                (x0 + dxx, y1 + dxx), (x0 - dx, y0 + dx),
(17)                (x0 + dxx, y0 - dxx),  # arrow
(17)                (x0 + dxx, y0)])
(4)    @_register_style(_style_list)
(4)    class RArrow(LArrow):
(8)        """A box in the shape of a right-pointing arrow."""
(8)        def __call__(self, x0, y0, width, height, mutation_size):
(12)           p = BoxStyle.LArrow.__call__(
(16)               self, x0, y0, width, height, mutation_size)
```

```
(12)                        p.vertices[:, 0] = 2 * x0 + width - p.vertices[:, 0]
(12)                        return p
(4)                 @_register_style(_style_list)
(4)                 class DArrow:
(8)                     """A box in the shape of a two-way arrow."""
(8)                     def __init__(self, pad=0.3):
(12)                        """
(12)                        Parameters
(12)                        ----------
(12)                        pad : float, default: 0.3
(16)                            The amount of padding around the original box.
(12)                        """
(12)                        self.pad = pad
(8)                     def __call__(self, x0, y0, width, height, mutation_size):
(12)                        pad = mutation_size * self.pad
(12)                        height = height + 2 * pad
(12)                        x0, y0 = x0 - pad, y0 - pad
(12)                        x1, y1 = x0 + width, y0 + height
(12)                        dx = (y1 - y0) / 2
(12)                        dxx = dx / 2
(12)                        x0 = x0 + pad / 1.4  # adjust by ~sqrt(2)
(12)                        return Path._create_closed([
(16)                            (x0 + dxx, y0), (x1, y0),  # bot-segment
(16)                            (x1, y0 - dxx), (x1 + dx + dxx, y0 + dx),
(16)                            (x1, y1 + dxx),  # right-arrow
(16)                            (x1, y1), (x0 + dxx, y1),  # top-segment
(16)                            (x0 + dxx, y1 + dxx), (x0 - dx, y0 + dx),
(16)                            (x0 + dxx, y0 - dxx),  # left-arrow
(16)                            (x0 + dxx, y0)])
(4)                 @_register_style(_style_list)
(4)                 class Round:
(8)                     """A box with round corners."""
(8)                     def __init__(self, pad=0.3, rounding_size=None):
(12)                        """
(12)                        Parameters
(12)                        ----------
(12)                        pad : float, default: 0.3
(16)                            The amount of padding around the original box.
(12)                        rounding_size : float, default: *pad*
(16)                            Radius of the corners.
(12)                        """
(12)                        self.pad = pad
(12)                        self.rounding_size = rounding_size
(8)                     def __call__(self, x0, y0, width, height, mutation_size):
(12)                        pad = mutation_size * self.pad
(12)                        if self.rounding_size:
(16)                            dr = mutation_size * self.rounding_size
(12)                        else:
(16)                            dr = pad
(12)                        width, height = width + 2 * pad, height + 2 * pad
(12)                        x0, y0 = x0 - pad, y0 - pad,
(12)                        x1, y1 = x0 + width, y0 + height
(12)                        cp = [(x0 + dr, y0),
(18)                              (x1 - dr, y0),
(18)                              (x1, y0), (x1, y0 + dr),
(18)                              (x1, y1 - dr),
(18)                              (x1, y1), (x1 - dr, y1),
(18)                              (x0 + dr, y1),
(18)                              (x0, y1), (x0, y1 - dr),
(18)                              (x0, y0 + dr),
(18)                              (x0, y0), (x0 + dr, y0),
(18)                              (x0 + dr, y0)]
(12)                        com = [Path.MOVETO,
(19)                               Path.LINETO,
(19)                               Path.CURVE3, Path.CURVE3,
(19)                               Path.LINETO,
(19)                               Path.CURVE3, Path.CURVE3,
(19)                               Path.LINETO,
(19)                               Path.CURVE3, Path.CURVE3,
```

```
(19)                            Path.LINETO,
(19)                            Path.CURVE3, Path.CURVE3,
(19)                            Path.CLOSEPOLY]
(12)                    return Path(cp, com)
(4)             @_register_style(_style_list)
(4)             class Round4:
(8)                 """A box with rounded edges."""
(8)                 def __init__(self, pad=0.3, rounding_size=None):
(12)                    """
(12)                    Parameters
(12)                    ----------
(12)                    pad : float, default: 0.3
(16)                        The amount of padding around the original box.
(12)                    rounding_size : float, default: *pad*/2
(16)                        Rounding of edges.
(12)                    """
(12)                    self.pad = pad
(12)                    self.rounding_size = rounding_size
(8)                 def __call__(self, x0, y0, width, height, mutation_size):
(12)                    pad = mutation_size * self.pad
(12)                    if self.rounding_size:
(16)                        dr = mutation_size * self.rounding_size
(12)                    else:
(16)                        dr = pad / 2.
(12)                    width = width + 2 * pad - 2 * dr
(12)                    height = height + 2 * pad - 2 * dr
(12)                    x0, y0 = x0 - pad + dr, y0 - pad + dr,
(12)                    x1, y1 = x0 + width, y0 + height
(12)                    cp = [(x0, y0),
(18)                          (x0 + dr, y0 - dr), (x1 - dr, y0 - dr), (x1, y0),
(18)                          (x1 + dr, y0 + dr), (x1 + dr, y1 - dr), (x1, y1),
(18)                          (x1 - dr, y1 + dr), (x0 + dr, y1 + dr), (x0, y1),
(18)                          (x0 - dr, y1 - dr), (x0 - dr, y0 + dr), (x0, y0),
(18)                          (x0, y0)]
(12)                    com = [Path.MOVETO,
(19)                           Path.CURVE4, Path.CURVE4, Path.CURVE4,
(19)                           Path.CURVE4, Path.CURVE4, Path.CURVE4,
(19)                           Path.CURVE4, Path.CURVE4, Path.CURVE4,
(19)                           Path.CURVE4, Path.CURVE4, Path.CURVE4,
(19)                           Path.CLOSEPOLY]
(12)                    return Path(cp, com)
(4)             @_register_style(_style_list)
(4)             class Sawtooth:
(8)                 """A box with a sawtooth outline."""
(8)                 def __init__(self, pad=0.3, tooth_size=None):
(12)                    """
(12)                    Parameters
(12)                    ----------
(12)                    pad : float, default: 0.3
(16)                        The amount of padding around the original box.
(12)                    tooth_size : float, default: *pad*/2
(16)                        Size of the sawtooth.
(12)                    """
(12)                    self.pad = pad
(12)                    self.tooth_size = tooth_size
(8)                 def _get_sawtooth_vertices(self, x0, y0, width, height,
mutation_size):
(12)                    pad = mutation_size * self.pad
(12)                    if self.tooth_size is None:
(16)                        tooth_size = self.pad * .5 * mutation_size
(12)                    else:
(16)                        tooth_size = self.tooth_size * mutation_size
(12)                    hsz = tooth_size / 2
(12)                    width = width + 2 * pad - tooth_size
(12)                    height = height + 2 * pad - tooth_size
(12)                    dsx_n = round((width - tooth_size) / (tooth_size * 2)) * 2
(12)                    dsy_n = round((height - tooth_size) / (tooth_size * 2)) * 2
(12)                    x0, y0 = x0 - pad + hsz, y0 - pad + hsz
(12)                    x1, y1 = x0 + width, y0 + height
```

```
(12)                    xs = [
(16)                        x0, *np.linspace(x0 + hsz, x1 - hsz, 2 * dsx_n + 1),  # bottom
(16)                        *([x1, x1 + hsz, x1, x1 - hsz] * dsy_n)[:2*dsy_n+2],  # right
(16)                        x1, *np.linspace(x1 - hsz, x0 + hsz, 2 * dsx_n + 1),  # top
(16)                        *([x0, x0 - hsz, x0, x0 + hsz] * dsy_n)[:2*dsy_n+2],  # left
(12)                    ]
(12)                    ys = [
(16)                        *([y0, y0 - hsz, y0, y0 + hsz] * dsx_n)[:2*dsx_n+2],  # bottom
(16)                        y0, *np.linspace(y0 + hsz, y1 - hsz, 2 * dsy_n + 1),  # right
(16)                        *([y1, y1 + hsz, y1, y1 - hsz] * dsx_n)[:2*dsx_n+2],  # top
(16)                        y1, *np.linspace(y1 - hsz, y0 + hsz, 2 * dsy_n + 1),  # left
(12)                    ]
(12)                    return [*zip(xs, ys), (xs[0], ys[0])]
(8)                 def __call__(self, x0, y0, width, height, mutation_size):
(12)                    saw_vertices = self._get_sawtooth_vertices(x0, y0, width,
(55)                                                               height, mutation_size)
(12)                    return Path(saw_vertices, closed=True)
(4)             @_register_style(_style_list)
(4)             class Roundtooth(Sawtooth):
(8)                 """A box with a rounded sawtooth outline."""
(8)                 def __call__(self, x0, y0, width, height, mutation_size):
(12)                    saw_vertices = self._get_sawtooth_vertices(x0, y0,
(55)                                                               width, height,
(55)                                                               mutation_size)
(12)                    saw_vertices = np.concatenate([saw_vertices, [saw_vertices[0]]])
(12)                    codes = ([Path.MOVETO] +
(21)                             [Path.CURVE3, Path.CURVE3] * ((len(saw_vertices)-1)//2) +
(21)                             [Path.CLOSEPOLY])
(12)                    return Path(saw_vertices, codes)
(0)         @_docstring.dedent_interpd
(0)         class ConnectionStyle(_Style):
(4)             """
(4)             `ConnectionStyle` is a container class which defines
(4)             several connectionstyle classes, which is used to create a path
(4)             between two points.  These are mainly used with `FancyArrowPatch`.
(4)             A connectionstyle object can be either created as::
(11)                   ConnectionStyle.Arc3(rad=0.2)
(4)             or::
(11)                   ConnectionStyle("Arc3", rad=0.2)
(4)             or::
(11)                   ConnectionStyle("Arc3, rad=0.2")
(4)             The following classes are defined
(4)             %(ConnectionStyle:table)s
(4)             An instance of any connection style class is a callable object,
(4)             whose call signature is::
(8)                 __call__(self, posA, posB,
(17)                         patchA=None, patchB=None,
(17)                         shrinkA=2., shrinkB=2.)
(4)             and it returns a `.Path` instance. *posA* and *posB* are
(4)             tuples of (x, y) coordinates of the two points to be
(4)             connected. *patchA* (or *patchB*) is given, the returned path is
(4)             clipped so that it start (or end) from the boundary of the
(4)             patch. The path is further shrunk by *shrinkA* (or *shrinkB*)
(4)             which is given in points.
(4)             """
(4)             _style_list = {}
(4)             class _Base:
(8)                 """
(8)                 A base class for connectionstyle classes. The subclass needs
(8)                 to implement a *connect* method whose call signature is::
(10)                  connect(posA, posB)
(8)                 where posA and posB are tuples of x, y coordinates to be
(8)                 connected.  The method needs to return a path connecting two
(8)                 points. This base class defines a __call__ method, and a few
(8)                 helper methods.
(8)                 """
(8)                 def _in_patch(self, patch):
(12)                    """
(12)                    Return a predicate function testing whether a point *xy* is
```

```
(12)                        contained in *patch*.
(12)                        """
(12)                        return lambda xy: patch.contains(
(16)                            SimpleNamespace(x=xy[0], y=xy[1]))[0]
(8)                     def _clip(self, path, in_start, in_stop):
(12)                        """
(12)                        Clip *path* at its start by the region where *in_start* returns
(12)                        True, and at its stop by the region where *in_stop* returns True.
(12)                        The original path is assumed to start in the *in_start* region and
(12)                        to stop in the *in_stop* region.
(12)                        """
(12)                        if in_start:
(16)                            try:
(20)                                _, path = split_path_inout(path, in_start)
(16)                            except ValueError:
(20)                                pass
(12)                        if in_stop:
(16)                            try:
(20)                                path, _ = split_path_inout(path, in_stop)
(16)                            except ValueError:
(20)                                pass
(12)                        return path
(8)                     def __call__(self, posA, posB,
(21)                                 shrinkA=2., shrinkB=2., patchA=None, patchB=None):
(12)                        """
(12)                        Call the *connect* method to create a path between *posA* and
(12)                        *posB*; then clip and shrink the path.
(12)                        """
(12)                        path = self.connect(posA, posB)
(12)                        path = self._clip(
(16)                            path,
(16)                            self._in_patch(patchA) if patchA else None,
(16)                            self._in_patch(patchB) if patchB else None,
(12)                        )
(12)                        path = self._clip(
(16)                            path,
(16)                            inside_circle(*path.vertices[0], shrinkA) if shrinkA else
None,
(16)                            inside_circle(*path.vertices[-1], shrinkB) if shrinkB else
None
(12)                        )
(12)                        return path
(4)                 @_register_style(_style_list)
(4)                 class Arc3(_Base):
(8)                     """
(8)                     Creates a simple quadratic Bézier curve between two
(8)                     points. The curve is created so that the middle control point
(8)                     (C1) is located at the same distance from the start (C0) and
(8)                     end points(C2) and the distance of the C1 to the line
(8)                     connecting C0-C2 is *rad* times the distance of C0-C2.
(8)                     """
(8)                     def __init__(self, rad=0.):
(12)                        """
(12)                        Parameters
(12)                        ----------
(12)                        rad : float
(14)                          Curvature of the curve.
(12)                        """
(12)                        self.rad = rad
(8)                     def connect(self, posA, posB):
(12)                        x1, y1 = posA
(12)                        x2, y2 = posB
(12)                        x12, y12 = (x1 + x2) / 2., (y1 + y2) / 2.
(12)                        dx, dy = x2 - x1, y2 - y1
(12)                        f = self.rad
(12)                        cx, cy = x12 + f * dy, y12 - f * dx
(12)                        vertices = [(x1, y1),
(24)                                    (cx, cy),
(24)                                    (x2, y2)]
```

```
(12)                        codes = [Path.MOVETO,
(21)                                 Path.CURVE3,
(21)                                 Path.CURVE3]
(12)                        return Path(vertices, codes)
(4)                 @_register_style(_style_list)
(4)                 class Angle3(_Base):
(8)                     """
(8)                     Creates a simple quadratic Bézier curve between two points. The middle
(8)                     control point is placed at the intersecting point of two lines which
(8)                     cross the start and end point, and have a slope of *angleA* and
(8)                     *angleB*, respectively.
(8)                     """
(8)                     def __init__(self, angleA=90, angleB=0):
(12)                        """
(12)                        Parameters
(12)                        ----------
(12)                        angleA : float
(14)                          Starting angle of the path.
(12)                        angleB : float
(14)                          Ending angle of the path.
(12)                        """
(12)                        self.angleA = angleA
(12)                        self.angleB = angleB
(8)                     def connect(self, posA, posB):
(12)                        x1, y1 = posA
(12)                        x2, y2 = posB
(12)                        cosA = math.cos(math.radians(self.angleA))
(12)                        sinA = math.sin(math.radians(self.angleA))
(12)                        cosB = math.cos(math.radians(self.angleB))
(12)                        sinB = math.sin(math.radians(self.angleB))
(12)                        cx, cy = get_intersection(x1, y1, cosA, sinA,
(38)                                                  x2, y2, cosB, sinB)
(12)                        vertices = [(x1, y1), (cx, cy), (x2, y2)]
(12)                        codes = [Path.MOVETO, Path.CURVE3, Path.CURVE3]
(12)                        return Path(vertices, codes)
(4)                 @_register_style(_style_list)
(4)                 class Angle(_Base):
(8)                     """
(8)                     Creates a piecewise continuous quadratic Bézier path between two
(8)                     points. The path has a one passing-through point placed at the
(8)                     intersecting point of two lines which cross the start and end point,
(8)                     and have a slope of *angleA* and *angleB*, respectively.
(8)                     The connecting edges are rounded with *rad*.
(8)                     """
(8)                     def __init__(self, angleA=90, angleB=0, rad=0.):
(12)                        """
(12)                        Parameters
(12)                        ----------
(12)                        angleA : float
(14)                          Starting angle of the path.
(12)                        angleB : float
(14)                          Ending angle of the path.
(12)                        rad : float
(14)                          Rounding radius of the edge.
(12)                        """
(12)                        self.angleA = angleA
(12)                        self.angleB = angleB
(12)                        self.rad = rad
(8)                     def connect(self, posA, posB):
(12)                        x1, y1 = posA
(12)                        x2, y2 = posB
(12)                        cosA = math.cos(math.radians(self.angleA))
(12)                        sinA = math.sin(math.radians(self.angleA))
(12)                        cosB = math.cos(math.radians(self.angleB))
(12)                        sinB = math.sin(math.radians(self.angleB))
(12)                        cx, cy = get_intersection(x1, y1, cosA, sinA,
(38)                                                  x2, y2, cosB, sinB)
(12)                        vertices = [(x1, y1)]
(12)                        codes = [Path.MOVETO]
```

```
(12)                        if self.rad == 0.:
(16)                            vertices.append((cx, cy))
(16)                            codes.append(Path.LINETO)
(12)                        else:
(16)                            dx1, dy1 = x1 - cx, y1 - cy
(16)                            d1 = np.hypot(dx1, dy1)
(16)                            f1 = self.rad / d1
(16)                            dx2, dy2 = x2 - cx, y2 - cy
(16)                            d2 = np.hypot(dx2, dy2)
(16)                            f2 = self.rad / d2
(16)                            vertices.extend([(cx + dx1 * f1, cy + dy1 * f1),
(33)                                             (cx, cy),
(33)                                             (cx + dx2 * f2, cy + dy2 * f2)])
(16)                            codes.extend([Path.LINETO, Path.CURVE3, Path.CURVE3])
(12)                        vertices.append((x2, y2))
(12)                        codes.append(Path.LINETO)
(12)                        return Path(vertices, codes)
(4)                 @_register_style(_style_list)
(4)                 class Arc(_Base):
(8)                     """
(8)                     Creates a piecewise continuous quadratic Bézier path between two
(8)                     points. The path can have two passing-through points, a
(8)                     point placed at the distance of *armA* and angle of *angleA* from
(8)                     point A, another point with respect to point B. The edges are
(8)                     rounded with *rad*.
(8)                     """
(8)                     def __init__(self, angleA=0, angleB=0, armA=None, armB=None, rad=0.):
(12)                        """
(12)                        Parameters
(12)                        ----------
(12)                        angleA : float
(14)                          Starting angle of the path.
(12)                        angleB : float
(14)                          Ending angle of the path.
(12)                        armA : float or None
(14)                          Length of the starting arm.
(12)                        armB : float or None
(14)                          Length of the ending arm.
(12)                        rad : float
(14)                          Rounding radius of the edges.
(12)                        """
(12)                        self.angleA = angleA
(12)                        self.angleB = angleB
(12)                        self.armA = armA
(12)                        self.armB = armB
(12)                        self.rad = rad
(8)                     def connect(self, posA, posB):
(12)                        x1, y1 = posA
(12)                        x2, y2 = posB
(12)                        vertices = [(x1, y1)]
(12)                        rounded = []
(12)                        codes = [Path.MOVETO]
(12)                        if self.armA:
(16)                            cosA = math.cos(math.radians(self.angleA))
(16)                            sinA = math.sin(math.radians(self.angleA))
(16)                            d = self.armA - self.rad
(16)                            rounded.append((x1 + d * cosA, y1 + d * sinA))
(16)                            d = self.armA
(16)                            rounded.append((x1 + d * cosA, y1 + d * sinA))
(12)                        if self.armB:
(16)                            cosB = math.cos(math.radians(self.angleB))
(16)                            sinB = math.sin(math.radians(self.angleB))
(16)                            x_armB, y_armB = x2 + self.armB * cosB, y2 + self.armB * sinB
(16)                            if rounded:
(20)                                xp, yp = rounded[-1]
(20)                                dx, dy = x_armB - xp, y_armB - yp
(20)                                dd = (dx * dx + dy * dy) ** .5
(20)                                rounded.append((xp + self.rad * dx / dd,
(36)                                                yp + self.rad * dy / dd))
```

```
2381: (20)                                vertices.extend(rounded)
2382: (20)                                codes.extend([Path.LINETO,
2383: (34)                                              Path.CURVE3,
2384: (34)                                              Path.CURVE3])
2385: (16)                            else:
2386: (20)                                xp, yp = vertices[-1]
2387: (20)                                dx, dy = x_armB - xp, y_armB - yp
2388: (20)                                dd = (dx * dx + dy * dy) ** .5
2389: (16)                            d = dd - self.rad
2390: (16)                            rounded = [(xp + d * dx / dd, yp + d * dy / dd),
2391: (27)                                       (x_armB, y_armB)]
2392: (12)                        if rounded:
2393: (16)                            xp, yp = rounded[-1]
2394: (16)                            dx, dy = x2 - xp, y2 - yp
2395: (16)                            dd = (dx * dx + dy * dy) ** .5
2396: (16)                            rounded.append((xp + self.rad * dx / dd,
2397: (32)                                            yp + self.rad * dy / dd))
2398: (16)                            vertices.extend(rounded)
2399: (16)                            codes.extend([Path.LINETO,
2400: (30)                                          Path.CURVE3,
2401: (30)                                          Path.CURVE3])
2402: (12)                        vertices.append((x2, y2))
2403: (12)                        codes.append(Path.LINETO)
2404: (12)                        return Path(vertices, codes)
2405: (4)                 @_register_style(_style_list)
2406: (4)                 class Bar(_Base):
2407: (8)                     """
2408: (8)                     A line with *angle* between A and B with *armA* and *armB*. One of the
2409: (8)                     arms is extended so that they are connected in a right angle. The
2410: (8)                     length of *armA* is determined by (*armA* + *fraction* x AB distance).
2411: (8)                     Same for *armB*.
2412: (8)                     """
2413: (8)                     def __init__(self, armA=0., armB=0., fraction=0.3, angle=None):
2414: (12)                        """
2415: (12)                        Parameters
2416: (12)                        ----------
2417: (12)                        armA : float
2418: (16)                            Minimum length of armA.
2419: (12)                        armB : float
2420: (16)                            Minimum length of armB.
2421: (12)                        fraction : float
2422: (16)                            A fraction of the distance between two points that will be
2423: (16)                            added to armA and armB.
2424: (12)                        angle : float or None
2425: (16)                            Angle of the connecting line (if None, parallel to A and B).
2426: (12)                        """
2427: (12)                        self.armA = armA
2428: (12)                        self.armB = armB
2429: (12)                        self.fraction = fraction
2430: (12)                        self.angle = angle
2431: (8)                     def connect(self, posA, posB):
2432: (12)                        x1, y1 = posA
2433: (12)                        x20, y20 = x2, y2 = posB
2434: (12)                        theta1 = math.atan2(y2 - y1, x2 - x1)
2435: (12)                        dx, dy = x2 - x1, y2 - y1
2436: (12)                        dd = (dx * dx + dy * dy) ** .5
2437: (12)                        ddx, ddy = dx / dd, dy / dd
2438: (12)                        armA, armB = self.armA, self.armB
2439: (12)                        if self.angle is not None:
2440: (16)                            theta0 = np.deg2rad(self.angle)
2441: (16)                            dtheta = theta1 - theta0
2442: (16)                            dl = dd * math.sin(dtheta)
2443: (16)                            dL = dd * math.cos(dtheta)
2444: (16)                            x2, y2 = x1 + dL * math.cos(theta0), y1 + dL *
math.sin(theta0)
2445: (16)                            armB = armB - dl
2446: (16)                            dx, dy = x2 - x1, y2 - y1
2447: (16)                            dd2 = (dx * dx + dy * dy) ** .5
2448: (16)                            ddx, ddy = dx / dd2, dy / dd2
```

```
(12)                    arm = max(armA, armB)
(12)                    f = self.fraction * dd + arm
(12)                    cx1, cy1 = x1 + f * ddy, y1 - f * ddx
(12)                    cx2, cy2 = x2 + f * ddy, y2 - f * ddx
(12)                    vertices = [(x1, y1),
(24)                                (cx1, cy1),
(24)                                (cx2, cy2),
(24)                                (x20, y20)]
(12)                    codes = [Path.MOVETO,
(21)                             Path.LINETO,
(21)                             Path.LINETO,
(21)                             Path.LINETO]
(12)                    return Path(vertices, codes)
(0)         def _point_along_a_line(x0, y0, x1, y1, d):
(4)             """
(4)             Return the point on the line connecting (*x0*, *y0*) -- (*x1*, *y1*) whose
(4)             distance from (*x0*, *y0*) is *d*.
(4)             """
(4)             dx, dy = x0 - x1, y0 - y1
(4)             ff = d / (dx * dx + dy * dy) ** .5
(4)             x2, y2 = x0 - ff * dx, y0 - ff * dy
(4)             return x2, y2
(0)         @_docstring.dedent_interpd
(0)         class ArrowStyle(_Style):
(4)             """
(4)             `ArrowStyle` is a container class which defines several
(4)             arrowstyle classes, which is used to create an arrow path along a
(4)             given path.  These are mainly used with `FancyArrowPatch`.
(4)             An arrowstyle object can be either created as::
(11)                   ArrowStyle.Fancy(head_length=.4, head_width=.4, tail_width=.4)
(4)             or::
(11)                   ArrowStyle("Fancy", head_length=.4, head_width=.4, tail_width=.4)
(4)             or::
(11)                   ArrowStyle("Fancy, head_length=.4, head_width=.4, tail_width=.4")
(4)             The following classes are defined
(4)             %(ArrowStyle:table)s
(4)             For an overview of the visual appearance, see
(4)             :doc:`/gallery/text_labels_and_annotations/fancyarrow_demo`.
(4)             An instance of any arrow style class is a callable object,
(4)             whose call signature is::
(8)                 __call__(self, path, mutation_size, linewidth, aspect_ratio=1.)
(4)             and it returns a tuple of a `.Path` instance and a boolean
(4)             value. *path* is a `.Path` instance along which the arrow
(4)             will be drawn. *mutation_size* and *aspect_ratio* have the same
(4)             meaning as in `BoxStyle`. *linewidth* is a line width to be
(4)             stroked. This is meant to be used to correct the location of the
(4)             head so that it does not overshoot the destination point, but not all
(4)             classes support it.
(4)             Notes
(4)             -----
(4)             *angleA* and *angleB* specify the orientation of the bracket, as either a
(4)             clockwise or counterclockwise angle depending on the arrow type. 0 degrees
(4)             means perpendicular to the line connecting the arrow's head and tail.
(4)             .. plot:: gallery/text_labels_and_annotations/angles_on_bracket_arrows.py
(4)             """
(4)             _style_list = {}
(4)             class _Base:
(8)                 """
(8)                 Arrow Transmuter Base class
(8)                 ArrowTransmuterBase and its derivatives are used to make a fancy
(8)                 arrow around a given path. The __call__ method returns a path
(8)                 (which will be used to create a PathPatch instance) and a boolean
(8)                 value indicating the path is open therefore is not fillable.  This
(8)                 class is not an artist and actual drawing of the fancy arrow is
(8)                 done by the FancyArrowPatch class.
(8)                 """
(8)                 @staticmethod
(8)                 def ensure_quadratic_bezier(path):
(12)                    """
```

```
(12)                        Some ArrowStyle classes only works with a simple quadratic
(12)                        Bézier curve (created with `.ConnectionStyle.Arc3` or
(12)                        `.ConnectionStyle.Angle3`). This static method checks if the
(12)                        provided path is a simple quadratic Bézier curve and returns its
(12)                        control points if true.
(12)                        """
(12)                        segments = list(path.iter_segments())
(12)                        if (len(segments) != 2 or segments[0][1] != Path.MOVETO or
(20)                                segments[1][1] != Path.CURVE3):
(16)                            raise ValueError(
(20)                                "'path' is not a valid quadratic Bezier curve")
(12)                        return [*segments[0][0], *segments[1][0]]
(8)                     def transmute(self, path, mutation_size, linewidth):
(12)                        """
(12)                        The transmute method is the very core of the ArrowStyle class and
(12)                        must be overridden in the subclasses. It receives the *path*
(12)                        object along which the arrow will be drawn, and the
(12)                        *mutation_size*, with which the arrow head etc. will be scaled.
(12)                        The *linewidth* may be used to adjust the path so that it does not
(12)                        pass beyond the given points. It returns a tuple of a `.Path`
(12)                        instance and a boolean. The boolean value indicate whether the
(12)                        path can be filled or not. The return value can also be a list of
(12)                        paths and list of booleans of the same length.
(12)                        """
(12)                        raise NotImplementedError('Derived must override')
(8)                     def __call__(self, path, mutation_size, linewidth,
(21)                                 aspect_ratio=1.):
(12)                        """
(12)                        The __call__ method is a thin wrapper around the transmute method
(12)                        and takes care of the aspect ratio.
(12)                        """
(12)                        if aspect_ratio is not None:
(16)                            vertices = path.vertices / [1, aspect_ratio]
(16)                            path_shrunk = Path(vertices, path.codes)
(16)                            path_mutated, fillable = self.transmute(path_shrunk,
(56)                                                                    mutation_size,
(56)                                                                    linewidth)
(16)                            if np.iterable(fillable):
(20)                                path_list = [Path(p.vertices * [1, aspect_ratio], p.codes)
(33)                                             for p in path_mutated]
(20)                                return path_list, fillable
(16)                            else:
(20)                                return path_mutated, fillable
(12)                        else:
(16)                            return self.transmute(path, mutation_size, linewidth)
(4)                 class _Curve(_Base):
(8)                     """
(8)                     A simple arrow which will work with any path instance. The
(8)                     returned path is the concatenation of the original path, and at
(8)                     most two paths representing the arrow head or bracket at the start
(8)                     point and at the end point. The arrow heads can be either open
(8)                     or closed.
(8)                     """
(8)                     arrow = "-"
(8)                     fillbegin = fillend = False  # Whether arrows are filled.
(8)                     def __init__(self, head_length=.4, head_width=.2, widthA=1.,
widthB=1.,
(21)                                 lengthA=0.2, lengthB=0.2, angleA=0, angleB=0,
scaleA=None,
(21)                                 scaleB=None):
(12)                        """
(12)                        Parameters
(12)                        ----------
(12)                        head_length : float, default: 0.4
(16)                            Length of the arrow head, relative to *mutation_size*.
(12)                        head_width : float, default: 0.2
(16)                            Width of the arrow head, relative to *mutation_size*.
(12)                        widthA, widthB : float, default: 1.0
(16)                            Width of the bracket.
```

```
(12)                        lengthA, lengthB : float, default: 0.2
(16)                            Length of the bracket.
(12)                        angleA, angleB : float, default: 0
(16)                            Orientation of the bracket, as a counterclockwise angle.
(16)                            0 degrees means perpendicular to the line.
(12)                        scaleA, scaleB : float, default: *mutation_size*
(16)                            The scale of the brackets.
(12)                        """
(12)                        self.head_length, self.head_width = head_length, head_width
(12)                        self.widthA, self.widthB = widthA, widthB
(12)                        self.lengthA, self.lengthB = lengthA, lengthB
(12)                        self.angleA, self.angleB = angleA, angleB
(12)                        self.scaleA, self.scaleB = scaleA, scaleB
(12)                        self._beginarrow_head = False
(12)                        self._beginarrow_bracket = False
(12)                        self._endarrow_head = False
(12)                        self._endarrow_bracket = False
(12)                        if "-" not in self.arrow:
(16)                            raise ValueError("arrow must have the '-' between "
(33)                                             "the two heads")
(12)                        beginarrow, endarrow = self.arrow.split("-", 1)
(12)                        if beginarrow == "<":
(16)                            self._beginarrow_head = True
(16)                            self._beginarrow_bracket = False
(12)                        elif beginarrow == "<|":
(16)                            self._beginarrow_head = True
(16)                            self._beginarrow_bracket = False
(16)                            self.fillbegin = True
(12)                        elif beginarrow in ("]", "|"):
(16)                            self._beginarrow_head = False
(16)                            self._beginarrow_bracket = True
(12)                        if endarrow == ">":
(16)                            self._endarrow_head = True
(16)                            self._endarrow_bracket = False
(12)                        elif endarrow == "|>":
(16)                            self._endarrow_head = True
(16)                            self._endarrow_bracket = False
(16)                            self.fillend = True
(12)                        elif endarrow in ("[", "|"):
(16)                            self._endarrow_head = False
(16)                            self._endarrow_bracket = True
(12)                        super().__init__()
(8)                     def _get_arrow_wedge(self, x0, y0, x1, y1,
(29)                                         head_dist, cos_t, sin_t, linewidth):
(12)                        """
(12)                        Return the paths for arrow heads. Since arrow lines are
(12)                        drawn with capstyle=projected, The arrow goes beyond the
(12)                        desired point. This method also returns the amount of the path
(12)                        to be shrunken so that it does not overshoot.
(12)                        """
(12)                        dx, dy = x0 - x1, y0 - y1
(12)                        cp_distance = np.hypot(dx, dy)
(12)                        pad_projected = (.5 * linewidth / sin_t)
(12)                        if cp_distance == 0:
(16)                            cp_distance = 1
(12)                        ddx = pad_projected * dx / cp_distance
(12)                        ddy = pad_projected * dy / cp_distance
(12)                        dx = dx / cp_distance * head_dist
(12)                        dy = dy / cp_distance * head_dist
(12)                        dx1, dy1 = cos_t * dx + sin_t * dy, -sin_t * dx + cos_t * dy
(12)                        dx2, dy2 = cos_t * dx - sin_t * dy, sin_t * dx + cos_t * dy
(12)                        vertices_arrow = [(x1 + ddx + dx1, y1 + ddy + dy1),
(30)                                          (x1 + ddx, y1 + ddy),
(30)                                          (x1 + ddx + dx2, y1 + ddy + dy2)]
(12)                        codes_arrow = [Path.MOVETO,
(27)                                       Path.LINETO,
(27)                                       Path.LINETO]
(12)                        return vertices_arrow, codes_arrow, ddx, ddy
(8)                     def _get_bracket(self, x0, y0,
```

```
(25)                                 x1, y1, width, length, angle):
(12)                     cos_t, sin_t = get_cos_sin(x1, y1, x0, y0)
(12)                     from matplotlib.bezier import get_normal_points
(12)                     x1, y1, x2, y2 = get_normal_points(x0, y0, cos_t, sin_t, width)
(12)                     dx, dy = length * cos_t, length * sin_t
(12)                     vertices_arrow = [(x1 + dx, y1 + dy),
(30)                                       (x1, y1),
(30)                                       (x2, y2),
(30)                                       (x2 + dx, y2 + dy)]
(12)                     codes_arrow = [Path.MOVETO,
(27)                                    Path.LINETO,
(27)                                    Path.LINETO,
(27)                                    Path.LINETO]
(12)                     if angle:
(16)                         trans = transforms.Affine2D().rotate_deg_around(x0, y0, angle)
(16)                         vertices_arrow = trans.transform(vertices_arrow)
(12)                     return vertices_arrow, codes_arrow
(8)                 def transmute(self, path, mutation_size, linewidth):
(12)                     if self._beginarrow_head or self._endarrow_head:
(16)                         head_length = self.head_length * mutation_size
(16)                         head_width = self.head_width * mutation_size
(16)                         head_dist = np.hypot(head_length, head_width)
(16)                         cos_t, sin_t = head_length / head_dist, head_width / head_dist
(12)                     scaleA = mutation_size if self.scaleA is None else self.scaleA
(12)                     scaleB = mutation_size if self.scaleB is None else self.scaleB
(12)                     x0, y0 = path.vertices[0]
(12)                     x1, y1 = path.vertices[1]
(12)                     has_begin_arrow = self._beginarrow_head and (x0, y0) != (x1, y1)
(12)                     verticesA, codesA, ddxA, ddyA = (
(16)                         self._get_arrow_wedge(x1, y1, x0, y0,
(38)                                               head_dist, cos_t, sin_t, linewidth)
(16)                         if has_begin_arrow
(16)                         else ([], [], 0, 0)
(12)                     )
(12)                     x2, y2 = path.vertices[-2]
(12)                     x3, y3 = path.vertices[-1]
(12)                     has_end_arrow = self._endarrow_head and (x2, y2) != (x3, y3)
(12)                     verticesB, codesB, ddxB, ddyB = (
(16)                         self._get_arrow_wedge(x2, y2, x3, y3,
(38)                                               head_dist, cos_t, sin_t, linewidth)
(16)                         if has_end_arrow
(16)                         else ([], [], 0, 0)
(12)                     )
(12)                     paths = [Path(np.concatenate([[(x0 + ddxA, y0 + ddyA)],
(42)                                                   path.vertices[1:-1],
(42)                                                   [(x3 + ddxB, y3 + ddyB)]]),
(26)                                  path.codes)]
(12)                     fills = [False]
(12)                     if has_begin_arrow:
(16)                         if self.fillbegin:
(20)                             paths.append(
(24)                                 Path([*verticesA, (0, 0)], [*codesA, Path.CLOSEPOLY]))
(20)                             fills.append(True)
(16)                         else:
(20)                             paths.append(Path(verticesA, codesA))
(20)                             fills.append(False)
(12)                     elif self._beginarrow_bracket:
(16)                         x0, y0 = path.vertices[0]
(16)                         x1, y1 = path.vertices[1]
(16)                         verticesA, codesA = self._get_bracket(x0, y0, x1, y1,
(54)                                                               self.widthA * scaleA,
(54)                                                               self.lengthA * scaleA,
(54)                                                               self.angleA)
(16)                         paths.append(Path(verticesA, codesA))
(16)                         fills.append(False)
(12)                     if has_end_arrow:
(16)                         if self.fillend:
(20)                             fills.append(True)
(20)                             paths.append(
```

```
(24)                                Path([*verticesB, (0, 0)], [*codesB, Path.CLOSEPOLY]))
(16)                        else:
(20)                            fills.append(False)
(20)                            paths.append(Path(verticesB, codesB))
(12)                    elif self._endarrow_bracket:
(16)                        x0, y0 = path.vertices[-1]
(16)                        x1, y1 = path.vertices[-2]
(16)                        verticesB, codesB = self._get_bracket(x0, y0, x1, y1,
(54)                                                              self.widthB * scaleB,
(54)                                                              self.lengthB * scaleB,
(54)                                                              self.angleB)
(16)                        paths.append(Path(verticesB, codesB))
(16)                        fills.append(False)
(12)                    return paths, fills
(4)             @_register_style(_style_list, name="-")
(4)             class Curve(_Curve):
(8)                 """A simple curve without any arrow head."""
(8)                 def __init__(self):  # hide head_length, head_width
(12)                    super().__init__(head_length=.2, head_width=.1)
(4)             @_register_style(_style_list, name="<-")
(4)             class CurveA(_Curve):
(8)                 """An arrow with a head at its start point."""
(8)                 arrow = "<-"
(4)             @_register_style(_style_list, name="->")
(4)             class CurveB(_Curve):
(8)                 """An arrow with a head at its end point."""
(8)                 arrow = "->"
(4)             @_register_style(_style_list, name="<->")
(4)             class CurveAB(_Curve):
(8)                 """An arrow with heads both at the start and the end point."""
(8)                 arrow = "<->"
(4)             @_register_style(_style_list, name="<|-")
(4)             class CurveFilledA(_Curve):
(8)                 """An arrow with filled triangle head at the start."""
(8)                 arrow = "<|-"
(4)             @_register_style(_style_list, name="-|>")
(4)             class CurveFilledB(_Curve):
(8)                 """An arrow with filled triangle head at the end."""
(8)                 arrow = "-|>"
(4)             @_register_style(_style_list, name="<|-|>")
(4)             class CurveFilledAB(_Curve):
(8)                 """An arrow with filled triangle heads at both ends."""
(8)                 arrow = "<|-|>"
(4)             @_register_style(_style_list, name="]-")
(4)             class BracketA(_Curve):
(8)                 """An arrow with an outward square bracket at its start."""
(8)                 arrow = "]-"
(8)                 def __init__(self, widthA=1., lengthA=0.2, angleA=0):
(12)                    """
(12)                    Parameters
(12)                    ----------
(12)                    widthA : float, default: 1.0
(16)                        Width of the bracket.
(12)                    lengthA : float, default: 0.2
(16)                        Length of the bracket.
(12)                    angleA : float, default: 0 degrees
(16)                        Orientation of the bracket, as a counterclockwise angle.
(16)                        0 degrees means perpendicular to the line.
(12)                    """
(12)                    super().__init__(widthA=widthA, lengthA=lengthA, angleA=angleA)
(4)             @_register_style(_style_list, name="-[")
(4)             class BracketB(_Curve):
(8)                 """An arrow with an outward square bracket at its end."""
(8)                 arrow = "-["
(8)                 def __init__(self, widthB=1., lengthB=0.2, angleB=0):
(12)                    """
(12)                    Parameters
(12)                    ----------
(12)                    widthB : float, default: 1.0
```

```
(16)                        Width of the bracket.
(12)                    lengthB : float, default: 0.2
(16)                        Length of the bracket.
(12)                    angleB : float, default: 0 degrees
(16)                        Orientation of the bracket, as a counterclockwise angle.
(16)                        0 degrees means perpendicular to the line.
(12)                    """
(12)                    super().__init__(widthB=widthB, lengthB=lengthB, angleB=angleB)
(4)             @_register_style(_style_list, name="]-[")
(4)             class BracketAB(_Curve):
(8)                 """An arrow with outward square brackets at both ends."""
(8)                 arrow = "]-["
(8)                 def __init__(self,
(21)                             widthA=1., lengthA=0.2, angleA=0,
(21)                             widthB=1., lengthB=0.2, angleB=0):
(12)                    """
(12)                    Parameters
(12)                    ----------
(12)                    widthA, widthB : float, default: 1.0
(16)                        Width of the bracket.
(12)                    lengthA, lengthB : float, default: 0.2
(16)                        Length of the bracket.
(12)                    angleA, angleB : float, default: 0 degrees
(16)                        Orientation of the bracket, as a counterclockwise angle.
(16)                        0 degrees means perpendicular to the line.
(12)                    """
(12)                    super().__init__(widthA=widthA, lengthA=lengthA, angleA=angleA,
(29)                                     widthB=widthB, lengthB=lengthB, angleB=angleB)
(4)             @_register_style(_style_list, name="|-|")
(4)             class BarAB(_Curve):
(8)                 """An arrow with vertical bars ``|`` at both ends."""
(8)                 arrow = "|-|"
(8)                 def __init__(self, widthA=1., angleA=0, widthB=1., angleB=0):
(12)                    """
(12)                    Parameters
(12)                    ----------
(12)                    widthA, widthB : float, default: 1.0
(16)                        Width of the bracket.
(12)                    angleA, angleB : float, default: 0 degrees
(16)                        Orientation of the bracket, as a counterclockwise angle.
(16)                        0 degrees means perpendicular to the line.
(12)                    """
(12)                    super().__init__(widthA=widthA, lengthA=0, angleA=angleA,
(29)                                     widthB=widthB, lengthB=0, angleB=angleB)
(4)             @_register_style(_style_list, name=']->')
(4)             class BracketCurve(_Curve):
(8)                 """
(8)                 An arrow with an outward square bracket at its start and a head at
(8)                 the end.
(8)                 """
(8)                 arrow = "]->"
(8)                 def __init__(self, widthA=1., lengthA=0.2, angleA=None):
(12)                    """
(12)                    Parameters
(12)                    ----------
(12)                    widthA : float, default: 1.0
(16)                        Width of the bracket.
(12)                    lengthA : float, default: 0.2
(16)                        Length of the bracket.
(12)                    angleA : float, default: 0 degrees
(16)                        Orientation of the bracket, as a counterclockwise angle.
(16)                        0 degrees means perpendicular to the line.
(12)                    """
(12)                    super().__init__(widthA=widthA, lengthA=lengthA, angleA=angleA)
(4)             @_register_style(_style_list, name='<-[')
(4)             class CurveBracket(_Curve):
(8)                 """
(8)                 An arrow with an outward square bracket at its end and a head at
(8)                 the start.
```

```
(8)                 """
(8)                 arrow = "<-["
(8)                 def __init__(self, widthB=1., lengthB=0.2, angleB=None):
(12)                    """
(12)                    Parameters
(12)                    ----------
(12)                    widthB : float, default: 1.0
(16)                        Width of the bracket.
(12)                    lengthB : float, default: 0.2
(16)                        Length of the bracket.
(12)                    angleB : float, default: 0 degrees
(16)                        Orientation of the bracket, as a counterclockwise angle.
(16)                        0 degrees means perpendicular to the line.
(12)                    """
(12)                    super().__init__(widthB=widthB, lengthB=lengthB, angleB=angleB)
(4)             @_register_style(_style_list)
(4)             class Simple(_Base):
(8)                 """A simple arrow. Only works with a quadratic Bézier curve."""
(8)                 def __init__(self, head_length=.5, head_width=.5, tail_width=.2):
(12)                    """
(12)                    Parameters
(12)                    ----------
(12)                    head_length : float, default: 0.5
(16)                        Length of the arrow head.
(12)                    head_width : float, default: 0.5
(16)                        Width of the arrow head.
(12)                    tail_width : float, default: 0.2
(16)                        Width of the arrow tail.
(12)                    """
(12)                    self.head_length, self.head_width, self.tail_width = \
(16)                        head_length, head_width, tail_width
(12)                    super().__init__()
(8)                 def transmute(self, path, mutation_size, linewidth):
(12)                    x0, y0, x1, y1, x2, y2 = self.ensure_quadratic_bezier(path)
(12)                    head_length = self.head_length * mutation_size
(12)                    in_f = inside_circle(x2, y2, head_length)
(12)                    arrow_path = [(x0, y0), (x1, y1), (x2, y2)]
(12)                    try:
(16)                        arrow_out, arrow_in = \
(20)                            split_bezier_intersecting_with_closedpath(arrow_path,
in_f)
(12)                    except NonIntersectingPathException:
(16)                        x0, y0 = _point_along_a_line(x2, y2, x1, y1, head_length)
(16)                        x1n, y1n = 0.5 * (x0 + x2), 0.5 * (y0 + y2)
(16)                        arrow_in = [(x0, y0), (x1n, y1n), (x2, y2)]
(16)                        arrow_out = None
(12)                    head_width = self.head_width * mutation_size
(12)                    head_left, head_right = make_wedged_bezier2(arrow_in,
(56)                                                                head_width / 2.,
wm=.5)
(12)                    if arrow_out is not None:
(16)                        tail_width = self.tail_width * mutation_size
(16)                        tail_left, tail_right = get_parallels(arrow_out,
(54)                                                              tail_width / 2.)
(16)                        patch_path = [(Path.MOVETO, tail_right[0]),
(30)                                      (Path.CURVE3, tail_right[1]),
(30)                                      (Path.CURVE3, tail_right[2]),
(30)                                      (Path.LINETO, head_right[0]),
(30)                                      (Path.CURVE3, head_right[1]),
(30)                                      (Path.CURVE3, head_right[2]),
(30)                                      (Path.CURVE3, head_left[1]),
(30)                                      (Path.CURVE3, head_left[0]),
(30)                                      (Path.LINETO, tail_left[2]),
(30)                                      (Path.CURVE3, tail_left[1]),
(30)                                      (Path.CURVE3, tail_left[0]),
(30)                                      (Path.LINETO, tail_right[0]),
(30)                                      (Path.CLOSEPOLY, tail_right[0]),
(30)                                      ]
(12)                    else:
```

```
2928: (16)                          patch_path = [(Path.MOVETO, head_right[0]),
2929: (30)                                        (Path.CURVE3, head_right[1]),
2930: (30)                                        (Path.CURVE3, head_right[2]),
2931: (30)                                        (Path.CURVE3, head_left[1]),
2932: (30)                                        (Path.CURVE3, head_left[0]),
2933: (30)                                        (Path.CLOSEPOLY, head_left[0]),
2934: (30)                                        ]
2935: (12)                      path = Path([p for c, p in patch_path], [c for c, p in
patch_path])
2936: (12)                      return path, True
2937: (4)               @_register_style(_style_list)
2938: (4)               class Fancy(_Base):
2939: (8)                   """A fancy arrow. Only works with a quadratic Bézier curve."""
2940: (8)                   def __init__(self, head_length=.4, head_width=.4, tail_width=.4):
2941: (12)                      """
2942: (12)                      Parameters
2943: (12)                      ----------
2944: (12)                      head_length : float, default: 0.4
2945: (16)                          Length of the arrow head.
2946: (12)                      head_width : float, default: 0.4
2947: (16)                          Width of the arrow head.
2948: (12)                      tail_width : float, default: 0.4
2949: (16)                          Width of the arrow tail.
2950: (12)                      """
2951: (12)                      self.head_length, self.head_width, self.tail_width = \
2952: (16)                          head_length, head_width, tail_width
2953: (12)                      super().__init__()
2954: (8)                   def transmute(self, path, mutation_size, linewidth):
2955: (12)                      x0, y0, x1, y1, x2, y2 = self.ensure_quadratic_bezier(path)
2956: (12)                      head_length = self.head_length * mutation_size
2957: (12)                      arrow_path = [(x0, y0), (x1, y1), (x2, y2)]
2958: (12)                      in_f = inside_circle(x2, y2, head_length)
2959: (12)                      try:
2960: (16)                          path_out, path_in = split_bezier_intersecting_with_closedpath(
2961: (20)                              arrow_path, in_f)
2962: (12)                      except NonIntersectingPathException:
2963: (16)                          x0, y0 = _point_along_a_line(x2, y2, x1, y1, head_length)
2964: (16)                          x1n, y1n = 0.5 * (x0 + x2), 0.5 * (y0 + y2)
2965: (16)                          arrow_path = [(x0, y0), (x1n, y1n), (x2, y2)]
2966: (16)                          path_head = arrow_path
2967: (12)                      else:
2968: (16)                          path_head = path_in
2969: (12)                      in_f = inside_circle(x2, y2, head_length * .8)
2970: (12)                      path_out, path_in = split_bezier_intersecting_with_closedpath(
2971: (16)                          arrow_path, in_f)
2972: (12)                      path_tail = path_out
2973: (12)                      head_width = self.head_width * mutation_size
2974: (12)                      head_l, head_r = make_wedged_bezier2(path_head,
2975: (49)                                                           head_width / 2.,
2976: (49)                                                           wm=.6)
2977: (12)                      tail_width = self.tail_width * mutation_size
2978: (12)                      tail_left, tail_right = make_wedged_bezier2(path_tail,
2979: (56)                                                                  tail_width * .5,
2980: (56)                                                                  w1=1., wm=0.6, w2=0.3)
2981: (12)                      in_f = inside_circle(x0, y0, tail_width * .3)
2982: (12)                      path_in, path_out = split_bezier_intersecting_with_closedpath(
2983: (16)                          arrow_path, in_f)
2984: (12)                      tail_start = path_in[-1]
2985: (12)                      head_right, head_left = head_r, head_l
2986: (12)                      patch_path = [(Path.MOVETO, tail_start),
2987: (26)                                    (Path.LINETO, tail_right[0]),
2988: (26)                                    (Path.CURVE3, tail_right[1]),
2989: (26)                                    (Path.CURVE3, tail_right[2]),
2990: (26)                                    (Path.LINETO, head_right[0]),
2991: (26)                                    (Path.CURVE3, head_right[1]),
2992: (26)                                    (Path.CURVE3, head_right[2]),
2993: (26)                                    (Path.CURVE3, head_left[1]),
2994: (26)                                    (Path.CURVE3, head_left[0]),
2995: (26)                                    (Path.LINETO, tail_left[2]),
```

```
2996: (26)                                  (Path.CURVE3, tail_left[1]),
2997: (26)                                  (Path.CURVE3, tail_left[0]),
2998: (26)                                  (Path.LINETO, tail_start),
2999: (26)                                  (Path.CLOSEPOLY, tail_start),
3000: (26)                                  ]
3001: (12)                      path = Path([p for c, p in patch_path], [c for c, p in
patch_path])
3002: (12)                      return path, True
3003: (4)               @_register_style(_style_list)
3004: (4)               class Wedge(_Base):
3005: (8)                   """
3006: (8)                   Wedge(?) shape. Only works with a quadratic Bézier curve.  The
3007: (8)                   start point has a width of the *tail_width* and the end point has a
3008: (8)                   width of 0. At the middle, the width is *shrink_factor*x*tail_width*.
3009: (8)                   """
3010: (8)                   def __init__(self, tail_width=.3, shrink_factor=0.5):
3011: (12)                      """
3012: (12)                      Parameters
3013: (12)                      ----------
3014: (12)                      tail_width : float, default: 0.3
3015: (16)                          Width of the tail.
3016: (12)                      shrink_factor : float, default: 0.5
3017: (16)                          Fraction of the arrow width at the middle point.
3018: (12)                      """
3019: (12)                      self.tail_width = tail_width
3020: (12)                      self.shrink_factor = shrink_factor
3021: (12)                      super().__init__()
3022: (8)                   def transmute(self, path, mutation_size, linewidth):
3023: (12)                      x0, y0, x1, y1, x2, y2 = self.ensure_quadratic_bezier(path)
3024: (12)                      arrow_path = [(x0, y0), (x1, y1), (x2, y2)]
3025: (12)                      b_plus, b_minus = make_wedged_bezier2(
3026: (36)                                              arrow_path,
3027: (36)                                              self.tail_width * mutation_size / 2.,
3028: (36)                                              wm=self.shrink_factor)
3029: (12)                      patch_path = [(Path.MOVETO, b_plus[0]),
3030: (26)                                    (Path.CURVE3, b_plus[1]),
3031: (26)                                    (Path.CURVE3, b_plus[2]),
3032: (26)                                    (Path.LINETO, b_minus[2]),
3033: (26)                                    (Path.CURVE3, b_minus[1]),
3034: (26)                                    (Path.CURVE3, b_minus[0]),
3035: (26)                                    (Path.CLOSEPOLY, b_minus[0]),
3036: (26)                                    ]
3037: (12)                      path = Path([p for c, p in patch_path], [c for c, p in
patch_path])
3038: (12)                      return path, True
3039: (0)           class FancyBboxPatch(Patch):
3040: (4)               """
3041: (4)               A fancy box around a rectangle with lower left at *xy* = (*x*, *y*)
3042: (4)               with specified width and height.
3043: (4)               `.FancyBboxPatch` is similar to `.Rectangle`, but it draws a fancy box
3044: (4)               around the rectangle. The transformation of the rectangle box to the
3045: (4)               fancy box is delegated to the style classes defined in `.BoxStyle`.
3046: (4)               """
3047: (4)               _edge_default = True
3048: (4)               def __str__(self):
3049: (8)                   s = self.__class__.__name__ + "((%g, %g), width=%g, height=%g)"
3050: (8)                   return s % (self._x, self._y, self._width, self._height)
3051: (4)               @_docstring.dedent_interpd
3052: (4)               def __init__(self, xy, width, height, boxstyle="round", *,
3053: (17)                           mutation_scale=1, mutation_aspect=1, **kwargs):
3054: (8)                   """
3055: (8)                   Parameters
3056: (8)                   ----------
3057: (8)                   xy : (float, float)
3058: (10)                    The lower left corner of the box.
3059: (8)                   width : float
3060: (12)                      The width of the box.
3061: (8)                   height : float
3062: (12)                      The height of the box.
```

```
(8)                 boxstyle : str or `~matplotlib.patches.BoxStyle`
(12)                    The style of the fancy box. This can either be a `.BoxStyle`
(12)                    instance or a string of the style name and optionally comma
(12)                    separated attributes (e.g. "Round, pad=0.2"). This string is
(12)                    passed to `.BoxStyle` to construct a `.BoxStyle` object. See
(12)                    there for a full documentation.
(12)                    The following box styles are available:
(12)                    %(BoxStyle:table)s
(8)                 mutation_scale : float, default: 1
(12)                    Scaling factor applied to the attributes of the box style
(12)                    (e.g. pad or rounding_size).
(8)                 mutation_aspect : float, default: 1
(12)                    The height of the rectangle will be squeezed by this value before
(12)                    the mutation and the mutated box will be stretched by the inverse
(12)                    of it. For example, this allows different horizontal and vertical
(12)                    padding.
(8)                 Other Parameters
(8)                 ----------------
(8)                 **kwargs : `~matplotlib.patches.Patch` properties
(8)                 %(Patch:kwdoc)s
(8)                 """
(8)                 super().__init__(**kwargs)
(8)                 self._x, self._y = xy
(8)                 self._width = width
(8)                 self._height = height
(8)                 self.set_boxstyle(boxstyle)
(8)                 self._mutation_scale = mutation_scale
(8)                 self._mutation_aspect = mutation_aspect
(8)                 self.stale = True
(4)             @_docstring.dedent_interpd
(4)             def set_boxstyle(self, boxstyle=None, **kwargs):
(8)                 """
(8)                 Set the box style, possibly with further attributes.
(8)                 Attributes from the previous box style are not reused.
(8)                 Without argument (or with ``boxstyle=None``), the available box styles
(8)                 are returned as a human-readable string.
(8)                 Parameters
(8)                 ----------
(8)                 boxstyle : str or `~matplotlib.patches.BoxStyle`
(12)                    The style of the box: either a `.BoxStyle` instance, or a string,
(12)                    which is the style name and optionally comma separated attributes
(12)                    (e.g. "Round,pad=0.2"). Such a string is used to construct a
(12)                    `.BoxStyle` object, as documented in that class.
(12)                    The following box styles are available:
(12)                    %(BoxStyle:table_and_accepts)s
(8)                 **kwargs
(12)                    Additional attributes for the box style. See the table above for
(12)                    supported parameters.
(8)                 Examples
(8)                 --------
(8)                 ::
(12)                    set_boxstyle("Round,pad=0.2")
(12)                    set_boxstyle("round", pad=0.2)
(8)                 """
(8)                 if boxstyle is None:
(12)                    return BoxStyle.pprint_styles()
(8)                 self._bbox_transmuter = (
(12)                    BoxStyle(boxstyle, **kwargs)
(12)                    if isinstance(boxstyle, str) else boxstyle)
(8)                 self.stale = True
(4)             def get_boxstyle(self):
(8)                 """Return the boxstyle object."""
(8)                 return self._bbox_transmuter
(4)             def set_mutation_scale(self, scale):
(8)                 """
(8)                 Set the mutation scale.
(8)                 Parameters
(8)                 ----------
(8)                 scale : float
```

```
(8)                 """
(8)                 self._mutation_scale = scale
(8)                 self.stale = True
(4)             def get_mutation_scale(self):
(8)                 """Return the mutation scale."""
(8)                 return self._mutation_scale
(4)             def set_mutation_aspect(self, aspect):
(8)                 """
(8)                 Set the aspect ratio of the bbox mutation.
(8)                 Parameters
(8)                 ----------
(8)                 aspect : float
(8)                 """
(8)                 self._mutation_aspect = aspect
(8)                 self.stale = True
(4)             def get_mutation_aspect(self):
(8)                 """Return the aspect ratio of the bbox mutation."""
(8)                 return (self._mutation_aspect if self._mutation_aspect is not None
(16)                        else 1)  # backcompat.
(4)             def get_path(self):
(8)                 """Return the mutated path of the rectangle."""
(8)                 boxstyle = self.get_boxstyle()
(8)                 m_aspect = self.get_mutation_aspect()
(8)                 path = boxstyle(self._x, self._y / m_aspect,
(24)                                self._width, self._height / m_aspect,
(24)                                self.get_mutation_scale())
(8)                 return Path(path.vertices * [1, m_aspect], path.codes)  # Unsqueeze y.
(4)             def get_x(self):
(8)                 """Return the left coord of the rectangle."""
(8)                 return self._x
(4)             def get_y(self):
(8)                 """Return the bottom coord of the rectangle."""
(8)                 return self._y
(4)             def get_width(self):
(8)                 """Return the width of the rectangle."""
(8)                 return self._width
(4)             def get_height(self):
(8)                 """Return the height of the rectangle."""
(8)                 return self._height
(4)             def set_x(self, x):
(8)                 """
(8)                 Set the left coord of the rectangle.
(8)                 Parameters
(8)                 ----------
(8)                 x : float
(8)                 """
(8)                 self._x = x
(8)                 self.stale = True
(4)             def set_y(self, y):
(8)                 """
(8)                 Set the bottom coord of the rectangle.
(8)                 Parameters
(8)                 ----------
(8)                 y : float
(8)                 """
(8)                 self._y = y
(8)                 self.stale = True
(4)             def set_width(self, w):
(8)                 """
(8)                 Set the rectangle width.
(8)                 Parameters
(8)                 ----------
(8)                 w : float
(8)                 """
(8)                 self._width = w
(8)                 self.stale = True
(4)             def set_height(self, h):
(8)                 """
(8)                 Set the rectangle height.
```

```
(8)                 Parameters
(8)                 ----------
(8)                 h : float
(8)                 """
(8)                 self._height = h
(8)                 self.stale = True
(4)             def set_bounds(self, *args):
(8)                 """
(8)                 Set the bounds of the rectangle.
(8)                 Call signatures::
(12)                    set_bounds(left, bottom, width, height)
(12)                    set_bounds((left, bottom, width, height))
(8)                 Parameters
(8)                 ----------
(8)                 left, bottom : float
(12)                    The coordinates of the bottom left corner of the rectangle.
(8)                 width, height : float
(12)                    The width/height of the rectangle.
(8)                 """
(8)                 if len(args) == 1:
(12)                    l, b, w, h = args[0]
(8)                 else:
(12)                    l, b, w, h = args
(8)                 self._x = l
(8)                 self._y = b
(8)                 self._width = w
(8)                 self._height = h
(8)                 self.stale = True
(4)             def get_bbox(self):
(8)                 """Return the `.Bbox`."""
(8)                 return transforms.Bbox.from_bounds(self._x, self._y,
(43)                                                   self._width, self._height)
(0)         class FancyArrowPatch(Patch):
(4)             """
(4)             A fancy arrow patch.
(4)             It draws an arrow using the `ArrowStyle`. It is primarily used by the
(4)             `~.axes.Axes.annotate` method.  For most purposes, use the annotate method
for
(4)             drawing arrows.
(4)             The head and tail positions are fixed at the specified start and end
points
(4)             of the arrow, but the size and shape (in display coordinates) of the arrow
(4)             does not change when the axis is moved or zoomed.
(4)             """
(4)             _edge_default = True
(4)             def __str__(self):
(8)                 if self._posA_posB is not None:
(12)                    (x1, y1), (x2, y2) = self._posA_posB
(12)                    return f"{type(self).__name__}(({x1:g}, {y1:g})->({x2:g},
{y2:g}))"
(8)                 else:
(12)                    return f"{type(self).__name__}({self._path_original})"
(4)             @_docstring.dedent_interpd
(4)             def __init__(self, posA=None, posB=None, *,
(17)                         path=None, arrowstyle="simple", connectionstyle="arc3",
(17)                         patchA=None, patchB=None, shrinkA=2, shrinkB=2,
(17)                         mutation_scale=1, mutation_aspect=1, **kwargs):
(8)                 """
(8)                 There are two ways for defining an arrow:
(8)                 - If *posA* and *posB* are given, a path connecting two points is
(10)                  created according to *connectionstyle*. The path will be
(10)                  clipped with *patchA* and *patchB* and further shrunken by
(10)                  *shrinkA* and *shrinkB*. An arrow is drawn along this
(10)                  resulting path using the *arrowstyle* parameter.
(8)                 - Alternatively if *path* is provided, an arrow is drawn along this
(10)                  path and *patchA*, *patchB*, *shrinkA*, and *shrinkB* are ignored.
(8)                 Parameters
(8)                 ----------
(8)                 posA, posB : (float, float), default: None
```

```
(12)                        (x, y) coordinates of arrow tail and arrow head respectively.
(8)                     path : `~matplotlib.path.Path`, default: None
(12)                        If provided, an arrow is drawn along this path and *patchA*,
(12)                        *patchB*, *shrinkA*, and *shrinkB* are ignored.
(8)                     arrowstyle : str or `.ArrowStyle`, default: 'simple'
(12)                        The `.ArrowStyle` with which the fancy arrow is drawn.  If a
(12)                        string, it should be one of the available arrowstyle names, with
(12)                        optional comma-separated attributes.  The optional attributes are
(12)                        meant to be scaled with the *mutation_scale*.  The following arrow
(12)                        styles are available:
(12)                        %(ArrowStyle:table)s
(8)                     connectionstyle : str or `.ConnectionStyle` or None, optional, \
(0)             default: 'arc3'
(12)                        The `.ConnectionStyle` with which *posA* and *posB* are connected.
(12)                        If a string, it should be one of the available connectionstyle
(12)                        names, with optional comma-separated attributes.  The following
(12)                        connection styles are available:
(12)                        %(ConnectionStyle:table)s
(8)                     patchA, patchB : `~matplotlib.patches.Patch`, default: None
(12)                        Head and tail patches, respectively.
(8)                     shrinkA, shrinkB : float, default: 2
(12)                        Shrink amount, in points, of the tail and head of the arrow
respectively.
(8)                     mutation_scale : float, default: 1
(12)                        Value with which attributes of *arrowstyle* (e.g., *head_length*)
(12)                        will be scaled.
(8)                     mutation_aspect : None or float, default: None
(12)                        The height of the rectangle will be squeezed by this value before
(12)                        the mutation and the mutated box will be stretched by the inverse
(12)                        of it.
(8)                     Other Parameters
(8)                     ----------------
(8)                     **kwargs : `~matplotlib.patches.Patch` properties, optional
(12)                        Here is a list of available `.Patch` properties:
(8)                     %(Patch:kwdoc)s
(12)                        In contrast to other patches, the default ``capstyle`` and
(12)                        ``joinstyle`` for `FancyArrowPatch` are set to ``"round"``.
(8)                     """
(8)                     kwargs.setdefault("joinstyle", JoinStyle.round)
(8)                     kwargs.setdefault("capstyle", CapStyle.round)
(8)                     super().__init__(**kwargs)
(8)                     if posA is not None and posB is not None and path is None:
(12)                        self._posA_posB = [posA, posB]
(12)                        if connectionstyle is None:
(16)                            connectionstyle = "arc3"
(12)                        self.set_connectionstyle(connectionstyle)
(8)                     elif posA is None and posB is None and path is not None:
(12)                        self._posA_posB = None
(8)                     else:
(12)                        raise ValueError("Either posA and posB, or path need to provided")
(8)                     self.patchA = patchA
(8)                     self.patchB = patchB
(8)                     self.shrinkA = shrinkA
(8)                     self.shrinkB = shrinkB
(8)                     self._path_original = path
(8)                     self.set_arrowstyle(arrowstyle)
(8)                     self._mutation_scale = mutation_scale
(8)                     self._mutation_aspect = mutation_aspect
(8)                     self._dpi_cor = 1.0
(4)                 def set_positions(self, posA, posB):
(8)                     """
(8)                     Set the start and end positions of the connecting path.
(8)                     Parameters
(8)                     ----------
(8)                     posA, posB : None, tuple
(12)                        (x, y) coordinates of arrow tail and arrow head respectively. If
(12)                        `None` use current value.
(8)                     """
(8)                     if posA is not None:
```

```
(12)                        self._posA_posB[0] = posA
(8)                     if posB is not None:
(12)                        self._posA_posB[1] = posB
(8)                     self.stale = True
(4)                 def set_patchA(self, patchA):
(8)                     """
(8)                     Set the tail patch.
(8)                     Parameters
(8)                     ----------
(8)                     patchA : `.patches.Patch`
(8)                     """
(8)                     self.patchA = patchA
(8)                     self.stale = True
(4)                 def set_patchB(self, patchB):
(8)                     """
(8)                     Set the head patch.
(8)                     Parameters
(8)                     ----------
(8)                     patchB : `.patches.Patch`
(8)                     """
(8)                     self.patchB = patchB
(8)                     self.stale = True
(4)                 @_docstring.dedent_interpd
(4)                 def set_connectionstyle(self, connectionstyle=None, **kwargs):
(8)                     """
(8)                     Set the connection style, possibly with further attributes.
(8)                     Attributes from the previous connection style are not reused.
(8)                     Without argument (or with ``connectionstyle=None``), the available box
(8)                     styles are returned as a human-readable string.
(8)                     Parameters
(8)                     ----------
(8)                     connectionstyle : str or `~matplotlib.patches.ConnectionStyle`
(12)                        The style of the connection: either a `.ConnectionStyle` instance,
(12)                        or a string, which is the style name and optionally comma
separated
(12)                        attributes (e.g. "Arc,armA=30,rad=10"). Such a string is used to
(12)                        construct a `.ConnectionStyle` object, as documented in that
class.
(12)                        The following connection styles are available:
(12)                        %(ConnectionStyle:table_and_accepts)s
(8)                     **kwargs
(12)                        Additional attributes for the connection style. See the table
above
(12)                        for supported parameters.
(8)                     Examples
(8)                     --------
(8)                     ::
(12)                        set_connectionstyle("Arc,armA=30,rad=10")
(12)                        set_connectionstyle("arc", armA=30, rad=10)
(8)                     """
(8)                     if connectionstyle is None:
(12)                        return ConnectionStyle.pprint_styles()
(8)                     self._connector = (
(12)                        ConnectionStyle(connectionstyle, **kwargs)
(12)                        if isinstance(connectionstyle, str) else connectionstyle)
(8)                     self.stale = True
(4)                 def get_connectionstyle(self):
(8)                     """Return the `ConnectionStyle` used."""
(8)                     return self._connector
(4)                 def set_arrowstyle(self, arrowstyle=None, **kwargs):
(8)                     """
(8)                     Set the arrow style, possibly with further attributes.
(8)                     Attributes from the previous arrow style are not reused.
(8)                     Without argument (or with ``arrowstyle=None``), the available box
(8)                     styles are returned as a human-readable string.
(8)                     Parameters
(8)                     ----------
(8)                     arrowstyle : str or `~matplotlib.patches.ArrowStyle`
(12)                        The style of the arrow: either a `.ArrowStyle` instance, or a
```

```
(12)                        string, which is the style name and optionally comma separated
(12)                        attributes (e.g. "Fancy,head_length=0.2"). Such a string is used
to
(12)                        construct a `.ArrowStyle` object, as documented in that class.
(12)                        The following arrow styles are available:
(12)                        %(ArrowStyle:table_and_accepts)s
(8)                     **kwargs
(12)                        Additional attributes for the arrow style. See the table above for
(12)                        supported parameters.
(8)                     Examples
(8)                     --------
(8)                     ::
(12)                        set_arrowstyle("Fancy,head_length=0.2")
(12)                        set_arrowstyle("fancy", head_length=0.2)
(8)                     """
(8)                     if arrowstyle is None:
(12)                        return ArrowStyle.pprint_styles()
(8)                     self._arrow_transmuter = (
(12)                        ArrowStyle(arrowstyle, **kwargs)
(12)                        if isinstance(arrowstyle, str) else arrowstyle)
(8)                     self.stale = True
(4)                 def get_arrowstyle(self):
(8)                     """Return the arrowstyle object."""
(8)                     return self._arrow_transmuter
(4)                 def set_mutation_scale(self, scale):
(8)                     """
(8)                     Set the mutation scale.
(8)                     Parameters
(8)                     ----------
(8)                     scale : float
(8)                     """
(8)                     self._mutation_scale = scale
(8)                     self.stale = True
(4)                 def get_mutation_scale(self):
(8)                     """
(8)                     Return the mutation scale.
(8)                     Returns
(8)                     -------
(8)                     scalar
(8)                     """
(8)                     return self._mutation_scale
(4)                 def set_mutation_aspect(self, aspect):
(8)                     """
(8)                     Set the aspect ratio of the bbox mutation.
(8)                     Parameters
(8)                     ----------
(8)                     aspect : float
(8)                     """
(8)                     self._mutation_aspect = aspect
(8)                     self.stale = True
(4)                 def get_mutation_aspect(self):
(8)                     """Return the aspect ratio of the bbox mutation."""
(8)                     return (self._mutation_aspect if self._mutation_aspect is not None
(16)                            else 1)  # backcompat.
(4)                 def get_path(self):
(8)                     """Return the path of the arrow in the data coordinates."""
(8)                     _path, fillable = self._get_path_in_displaycoord()
(8)                     if np.iterable(fillable):
(12)                        _path = Path.make_compound_path(*_path)
(8)                     return self.get_transform().inverted().transform_path(_path)
(4)                 def _get_path_in_displaycoord(self):
(8)                     """Return the mutated path of the arrow in display coordinates."""
(8)                     dpi_cor = self._dpi_cor
(8)                     if self._posA_posB is not None:
(12)                        posA = self._convert_xy_units(self._posA_posB[0])
(12)                        posB = self._convert_xy_units(self._posA_posB[1])
(12)                        (posA, posB) = self.get_transform().transform((posA, posB))
(12)                        _path = self.get_connectionstyle()(posA, posB,
(47)                                                           patchA=self.patchA,
```

```
3469: (47)                                                          patchB=self.patchB,
3470: (47)                                                          shrinkA=self.shrinkA * dpi_cor,
3471: (47)                                                          shrinkB=self.shrinkB * dpi_cor
3472: (47)                                                          )
3473: (8)                   else:
3474: (12)                      _path = self.get_transform().transform_path(self._path_original)
3475: (8)                   _path, fillable = self.get_arrowstyle()(
3476: (12)                      _path,
3477: (12)                      self.get_mutation_scale() * dpi_cor,
3478: (12)                      self.get_linewidth() * dpi_cor,
3479: (12)                      self.get_mutation_aspect())
3480: (8)                   return _path, fillable
3481: (4)               def draw(self, renderer):
3482: (8)                   if not self.get_visible():
3483: (12)                      return
3484: (8)                   self._dpi_cor = renderer.points_to_pixels(1.)
3485: (8)                   path, fillable = self._get_path_in_displaycoord()
3486: (8)                   if not np.iterable(fillable):
3487: (12)                      path = [path]
3488: (12)                      fillable = [fillable]
3489: (8)                   affine = transforms.IdentityTransform()
3490: (8)                   self._draw_paths_with_artist_properties(
3491: (12)                      renderer,
3492: (12)                      [(p, affine, self._facecolor if f and self._facecolor[3] else
None)
3493: (13)                       for p, f in zip(path, fillable)])
3494: (0)           class ConnectionPatch(FancyArrowPatch):
3495: (4)               """A patch that connects two points (possibly in different Axes)."""
3496: (4)               def __str__(self):
3497: (8)                   return "ConnectionPatch((%g, %g), (%g, %g))" % \
3498: (15)                         (self.xy1[0], self.xy1[1], self.xy2[0], self.xy2[1])
3499: (4)               @_docstring.dedent_interpd
3500: (4)               def __init__(self, xyA, xyB, coordsA, coordsB=None, *,
3501: (17)                           axesA=None, axesB=None,
3502: (17)                           arrowstyle="-",
3503: (17)                           connectionstyle="arc3",
3504: (17)                           patchA=None,
3505: (17)                           patchB=None,
3506: (17)                           shrinkA=0.,
3507: (17)                           shrinkB=0.,
3508: (17)                           mutation_scale=10.,
3509: (17)                           mutation_aspect=None,
3510: (17)                           clip_on=False,
3511: (17)                           **kwargs):
3512: (8)                   """
3513: (8)                   Connect point *xyA* in *coordsA* with point *xyB* in *coordsB*.
3514: (8)                   Valid keys are
3515: (8)                   ===============
=====================================================
3516: (8)                   Key              Description
3517: (8)                   ===============
=====================================================
3518: (8)                   arrowstyle       the arrow style
3519: (8)                   connectionstyle  the connection style
3520: (8)                   relpos           default is (0.5, 0.5)
3521: (8)                   patchA           default is bounding box of the text
3522: (8)                   patchB           default is None
3523: (8)                   shrinkA          default is 2 points
3524: (8)                   shrinkB          default is 2 points
3525: (8)                   mutation_scale   default is text size (in points)
3526: (8)                   mutation_aspect  default is 1.
3527: (8)                   ?                any key for `matplotlib.patches.PathPatch`
3528: (8)                   ===============
=====================================================
3529: (8)                   *coordsA* and *coordsB* are strings that indicate the
3530: (8)                   coordinates of *xyA* and *xyB*.
3531: (8)                   ====================
=====================================================
3532: (8)                   Property             Description
```

```
(8)                     ====================
==================================================
(8)                     'figure points'      points from the lower left corner of the figure
(8)                     'figure pixels'      pixels from the lower left corner of the figure
(8)                     'figure fraction'    0, 0 is lower left of figure and 1, 1 is upper
(29)                                         right
(8)                     'subfigure points'   points from the lower left corner of the
subfigure
(8)                     'subfigure pixels'   pixels from the lower left corner of the
subfigure
(8)                     'subfigure fraction' fraction of the subfigure, 0, 0 is lower left.
(8)                     'axes points'        points from lower left corner of the Axes
(8)                     'axes pixels'        pixels from lower left corner of the Axes
(8)                     'axes fraction'      0, 0 is lower left of Axes and 1, 1 is upper
right
(8)                     'data'               use the coordinate system of the object being
(29)                                         annotated (default)
(8)                     'offset points'      offset (in points) from the *xy* value
(8)                     'polar'              you can specify *theta*, *r* for the annotation,
(29)                                         even in cartesian plots.  Note that if you are
(29)                                         using a polar Axes, you do not need to specify
(29)                                         polar for the coordinate system since that is the
(29)                                         native "data" coordinate system.
(8)                     ====================
==================================================
(8)                     Alternatively they can be set to any valid
(8)                     `~matplotlib.transforms.Transform`.
(8)                     Note that 'subfigure pixels' and 'figure pixels' are the same
(8)                     for the parent figure, so users who want code that is usable in
(8)                     a subfigure can use 'subfigure pixels'.
(8)                     .. note::
(11)                       Using `ConnectionPatch` across two `~.axes.Axes` instances
(11)                       is not directly compatible with :ref:`constrained layout
(11)                       <constrainedlayout_guide>`. Add the artist
(11)                       directly to the `.Figure` instead of adding it to a specific Axes,
(11)                       or exclude it from the layout using ``con.set_in_layout(False)``.
(11)                       .. code-block:: default
(14)                          fig, ax = plt.subplots(1, 2, constrained_layout=True)
(14)                          con = ConnectionPatch(..., axesA=ax[0], axesB=ax[1])
(14)                          fig.add_artist(con)
(8)                     """
(8)                     if coordsB is None:
(12)                        coordsB = coordsA
(8)                     self.xy1 = xyA
(8)                     self.xy2 = xyB
(8)                     self.coords1 = coordsA
(8)                     self.coords2 = coordsB
(8)                     self.axesA = axesA
(8)                     self.axesB = axesB
(8)                     super().__init__(posA=(0, 0), posB=(1, 1),
(25)                                     arrowstyle=arrowstyle,
(25)                                     connectionstyle=connectionstyle,
(25)                                     patchA=patchA, patchB=patchB,
(25)                                     shrinkA=shrinkA, shrinkB=shrinkB,
(25)                                     mutation_scale=mutation_scale,
(25)                                     mutation_aspect=mutation_aspect,
(25)                                     clip_on=clip_on,
(25)                                     **kwargs)
(8)                     self._annotation_clip = None
(4)                 def _get_xy(self, xy, s, axes=None):
(8)                     """Calculate the pixel position of given point."""
(8)                     s0 = s  # For the error message, if needed.
(8)                     if axes is None:
(12)                        axes = self.axes
(8)                     xy = np.array(xy)
(8)                     if s in ["figure points", "axes points"]:
(12)                        xy *= self.figure.dpi / 72
(12)                        s = s.replace("points", "pixels")
(8)                     elif s == "figure fraction":
```

```
(12)                        s = self.figure.transFigure
(8)                     elif s == "subfigure fraction":
(12)                        s = self.figure.transSubfigure
(8)                     elif s == "axes fraction":
(12)                        s = axes.transAxes
(8)                     x, y = xy
(8)                     if s == 'data':
(12)                        trans = axes.transData
(12)                        x = float(self.convert_xunits(x))
(12)                        y = float(self.convert_yunits(y))
(12)                        return trans.transform((x, y))
(8)                     elif s == 'offset points':
(12)                        if self.xycoords == 'offset points':  # prevent recursion
(16)                            return self._get_xy(self.xy, 'data')
(12)                        return (
(16)                            self._get_xy(self.xy, self.xycoords)  # converted data point
(16)                            + xy * self.figure.dpi / 72)  # converted offset
(8)                     elif s == 'polar':
(12)                        theta, r = x, y
(12)                        x = r * np.cos(theta)
(12)                        y = r * np.sin(theta)
(12)                        trans = axes.transData
(12)                        return trans.transform((x, y))
(8)                     elif s == 'figure pixels':
(12)                        bb = self.figure.figbbox
(12)                        x = bb.x0 + x if x >= 0 else bb.x1 + x
(12)                        y = bb.y0 + y if y >= 0 else bb.y1 + y
(12)                        return x, y
(8)                     elif s == 'subfigure pixels':
(12)                        bb = self.figure.bbox
(12)                        x = bb.x0 + x if x >= 0 else bb.x1 + x
(12)                        y = bb.y0 + y if y >= 0 else bb.y1 + y
(12)                        return x, y
(8)                     elif s == 'axes pixels':
(12)                        bb = axes.bbox
(12)                        x = bb.x0 + x if x >= 0 else bb.x1 + x
(12)                        y = bb.y0 + y if y >= 0 else bb.y1 + y
(12)                        return x, y
(8)                     elif isinstance(s, transforms.Transform):
(12)                        return s.transform(xy)
(8)                     else:
(12)                        raise ValueError(f"{s0} is not a valid coordinate transformation")
(4)                 def set_annotation_clip(self, b):
(8)                     """
(8)                     Set the annotation's clipping behavior.
(8)                     Parameters
(8)                     ----------
(8)                     b : bool or None
(12)                        - True: The annotation will be clipped when ``self.xy`` is
(14)                          outside the Axes.
(12)                        - False: The annotation will always be drawn.
(12)                        - None: The annotation will be clipped when ``self.xy`` is
(14)                          outside the Axes and ``self.xycoords == "data"``.
(8)                     """
(8)                     self._annotation_clip = b
(8)                     self.stale = True
(4)                 def get_annotation_clip(self):
(8)                     """
(8)                     Return the clipping behavior.
(8)                     See `.set_annotation_clip` for the meaning of the return value.
(8)                     """
(8)                     return self._annotation_clip
(4)                 def _get_path_in_displaycoord(self):
(8)                     """Return the mutated path of the arrow in display coordinates."""
(8)                     dpi_cor = self._dpi_cor
(8)                     posA = self._get_xy(self.xy1, self.coords1, self.axesA)
(8)                     posB = self._get_xy(self.xy2, self.coords2, self.axesB)
(8)                     path = self.get_connectionstyle()(
(12)                        posA, posB,
```

```
3666: (12)                      patchA=self.patchA, patchB=self.patchB,
3667: (12)                      shrinkA=self.shrinkA * dpi_cor, shrinkB=self.shrinkB * dpi_cor,
3668: (8)                   )
3669: (8)                   path, fillable = self.get_arrowstyle()(
3670: (12)                      path,
3671: (12)                      self.get_mutation_scale() * dpi_cor,
3672: (12)                      self.get_linewidth() * dpi_cor,
3673: (12)                      self.get_mutation_aspect()
3674: (8)                   )
3675: (8)                   return path, fillable
3676: (4)               def _check_xy(self, renderer):
3677: (8)                   """Check whether the annotation needs to be drawn."""
3678: (8)                   b = self.get_annotation_clip()
3679: (8)                   if b or (b is None and self.coords1 == "data"):
3680: (12)                      xy_pixel = self._get_xy(self.xy1, self.coords1, self.axesA)
3681: (12)                      if self.axesA is None:
3682: (16)                          axes = self.axes
3683: (12)                      else:
3684: (16)                          axes = self.axesA
3685: (12)                      if not axes.contains_point(xy_pixel):
3686: (16)                          return False
3687: (8)                   if b or (b is None and self.coords2 == "data"):
3688: (12)                      xy_pixel = self._get_xy(self.xy2, self.coords2, self.axesB)
3689: (12)                      if self.axesB is None:
3690: (16)                          axes = self.axes
3691: (12)                      else:
3692: (16)                          axes = self.axesB
3693: (12)                      if not axes.contains_point(xy_pixel):
3694: (16)                          return False
3695: (8)                   return True
3696: (4)               def draw(self, renderer):
3697: (8)                   if not self.get_visible() or not self._check_xy(renderer):
3698: (12)                      return
3699: (8)                   super().draw(renderer)
```

File 52 - patheffects.py:

```
1: (0)              """
2: (0)              Defines classes for path effects. The path effects are supported in `.Text`,
3: (0)              `.Line2D` and `.Patch`.
4: (0)              .. seealso::
5: (3)                 :ref:`patheffects_guide`
6: (0)              """
7: (0)              from matplotlib.backend_bases import RendererBase
8: (0)              from matplotlib import colors as mcolors
9: (0)              from matplotlib import patches as mpatches
10: (0)             from matplotlib import transforms as mtransforms
11: (0)             from matplotlib.path import Path
12: (0)             import numpy as np
13: (0)             class AbstractPathEffect:
14: (4)                 """
15: (4)                 A base class for path effects.
16: (4)                 Subclasses should override the ``draw_path`` method to add effect
17: (4)                 functionality.
18: (4)                 """
19: (4)                 def __init__(self, offset=(0., 0.)):
20: (8)                     """
21: (8)                     Parameters
22: (8)                     ----------
23: (8)                     offset : (float, float), default: (0, 0)
24: (12)                        The (x, y) offset to apply to the path, measured in points.
25: (8)                     """
26: (8)                     self._offset = offset
27: (4)                 def _offset_transform(self, renderer):
28: (8)                     """Apply the offset to the given transform."""
29: (8)                     return mtransforms.Affine2D().translate(
30: (12)                        *map(renderer.points_to_pixels, self._offset))
```

```
 31: (4)                 def _update_gc(self, gc, new_gc_dict):
 32: (8)                     """
 33: (8)                     Update the given GraphicsContext with the given dict of properties.
 34: (8)                     The keys in the dictionary are used to identify the appropriate
 35: (8)                     ``set_`` method on the *gc*.
 36: (8)                     """
 37: (8)                     new_gc_dict = new_gc_dict.copy()
 38: (8)                     dashes = new_gc_dict.pop("dashes", None)
 39: (8)                     if dashes:
 40: (12)                        gc.set_dashes(**dashes)
 41: (8)                     for k, v in new_gc_dict.items():
 42: (12)                        set_method = getattr(gc, 'set_' + k, None)
 43: (12)                        if not callable(set_method):
 44: (16)                            raise AttributeError(f'Unknown property {k}')
 45: (12)                        set_method(v)
 46: (8)                     return gc
 47: (4)                 def draw_path(self, renderer, gc, tpath, affine, rgbFace=None):
 48: (8)                     """
 49: (8)                     Derived should override this method. The arguments are the same
 50: (8)                     as :meth:`matplotlib.backend_bases.RendererBase.draw_path`
 51: (8)                     except the first argument is a renderer.
 52: (8)                     """
 53: (8)                     if isinstance(renderer, PathEffectRenderer):
 54: (12)                        renderer = renderer._renderer
 55: (8)                     return renderer.draw_path(gc, tpath, affine, rgbFace)
 56: (0)             class PathEffectRenderer(RendererBase):
 57: (4)                 """
 58: (4)                 Implements a Renderer which contains another renderer.
 59: (4)                 This proxy then intercepts draw calls, calling the appropriate
 60: (4)                 :class:`AbstractPathEffect` draw method.
 61: (4)                 .. note::
 62: (8)                     Not all methods have been overridden on this RendererBase subclass.
 63: (8)                     It may be necessary to add further methods to extend the PathEffects
 64: (8)                     capabilities further.
 65: (4)                 """
 66: (4)                 def __init__(self, path_effects, renderer):
 67: (8)                     """
 68: (8)                     Parameters
 69: (8)                     ----------
 70: (8)                     path_effects : iterable of :class:`AbstractPathEffect`
 71: (12)                        The path effects which this renderer represents.
 72: (8)                     renderer : `~matplotlib.backend_bases.RendererBase` subclass
 73: (8)                     """
 74: (8)                     self._path_effects = path_effects
 75: (8)                     self._renderer = renderer
 76: (4)                 def copy_with_path_effect(self, path_effects):
 77: (8)                     return self.__class__(path_effects, self._renderer)
 78: (4)                 def draw_path(self, gc, tpath, affine, rgbFace=None):
 79: (8)                     for path_effect in self._path_effects:
 80: (12)                        path_effect.draw_path(self._renderer, gc, tpath, affine,
 81: (34)                                              rgbFace)
 82: (4)                 def draw_markers(
 83: (12)                        self, gc, marker_path, marker_trans, path, *args, **kwargs):
 84: (8)                     if len(self._path_effects) == 1:
 85: (12)                        return super().draw_markers(gc, marker_path, marker_trans, path,
 86: (40)                                                    *args, **kwargs)
 87: (8)                     for path_effect in self._path_effects:
 88: (12)                        renderer = self.copy_with_path_effect([path_effect])
 89: (12)                        renderer.draw_markers(gc, marker_path, marker_trans, path,
 90: (34)                                              *args, **kwargs)
 91: (4)                 def draw_path_collection(self, gc, master_transform, paths, *args,
 92: (29)                                         **kwargs):
 93: (8)                     if len(self._path_effects) == 1:
 94: (12)                        return super().draw_path_collection(gc, master_transform, paths,
 95: (48)                                                            *args, **kwargs)
 96: (8)                     for path_effect in self._path_effects:
 97: (12)                        renderer = self.copy_with_path_effect([path_effect])
 98: (12)                        renderer.draw_path_collection(gc, master_transform, paths,
 99: (42)                                                      *args, **kwargs)
```

```
(4)     def _draw_text_as_path(self, gc, x, y, s, prop, angle, ismath):
(8)         path, transform = self._get_text_path_transform(x, y, s, prop,
(56)                                                         angle, ismath)
(8)         color = gc.get_rgb()
(8)         gc.set_linewidth(0.0)
(8)         self.draw_path(gc, path, transform, rgbFace=color)
(4)     def __getattribute__(self, name):
(8)         if name in ['flipy', 'get_canvas_width_height', 'new_gc',
(20)                     'points_to_pixels', '_text2path', 'height', 'width']:
(12)            return getattr(self._renderer, name)
(8)         else:
(12)            return object.__getattribute__(self, name)
(4)     def open_group(self, s, gid=None):
(8)         return self._renderer.open_group(s, gid)
(4)     def close_group(self, s):
(8)         return self._renderer.close_group(s)
(0) class Normal(AbstractPathEffect):
(4)     """
(4)     The "identity" PathEffect.
(4)     The Normal PathEffect's sole purpose is to draw the original artist with
(4)     no special path effect.
(4)     """
(0) def _subclass_with_normal(effect_class):
(4)     """
(4)     Create a PathEffect class combining *effect_class* and a normal draw.
(4)     """
(4)     class withEffect(effect_class):
(8)         def draw_path(self, renderer, gc, tpath, affine, rgbFace):
(12)            super().draw_path(renderer, gc, tpath, affine, rgbFace)
(12)            renderer.draw_path(gc, tpath, affine, rgbFace)
(4)     withEffect.__name__ = f"with{effect_class.__name__}"
(4)     withEffect.__qualname__ = f"with{effect_class.__name__}"
(4)     withEffect.__doc__ = f"""
(4)     A shortcut PathEffect for applying `.{effect_class.__name__}` and then
(4)     drawing the original Artist.
(4)     With this class you can use ::
(8)         artist.set_path_effects([patheffects.with{effect_class.__name__}()])
(4)     as a shortcut for ::
(8)         artist.set_path_effects([patheffects.{effect_class.__name__}(),
(33)                                  patheffects.Normal()])
(4)     """
(4)     withEffect.draw_path.__doc__ = effect_class.draw_path.__doc__
(4)     return withEffect
(0) class Stroke(AbstractPathEffect):
(4)     """A line based PathEffect which re-draws a stroke."""
(4)     def __init__(self, offset=(0, 0), **kwargs):
(8)         """
(8)         The path will be stroked with its gc updated with the given
(8)         keyword arguments, i.e., the keyword arguments should be valid
(8)         gc parameter values.
(8)         """
(8)         super().__init__(offset)
(8)         self._gc = kwargs
(4)     def draw_path(self, renderer, gc, tpath, affine, rgbFace):
(8)         """Draw the path with updated gc."""
(8)         gc0 = renderer.new_gc()  # Don't modify gc, but a copy!
(8)         gc0.copy_properties(gc)
(8)         gc0 = self._update_gc(gc0, self._gc)
(8)         renderer.draw_path(
(12)            gc0, tpath, affine + self._offset_transform(renderer), rgbFace)
(8)         gc0.restore()
(0) withStroke = _subclass_with_normal(effect_class=Stroke)
(0) class SimplePatchShadow(AbstractPathEffect):
(4)     """A simple shadow via a filled patch."""
(4)     def __init__(self, offset=(2, -2),
(17)                 shadow_rgbFace=None, alpha=None,
(17)                 rho=0.3, **kwargs):
(8)         """
(8)         Parameters
```

```
(8)                     ----------
(8)                     offset : (float, float), default: (2, -2)
(12)                        The (x, y) offset of the shadow in points.
(8)                     shadow_rgbFace : :mpltype:`color`
(12)                        The shadow color.
(8)                     alpha : float, default: 0.3
(12)                        The alpha transparency of the created shadow patch.
(8)                     rho : float, default: 0.3
(12)                        A scale factor to apply to the rgbFace color if *shadow_rgbFace*
(12)                        is not specified.
(8)                     **kwargs
(12)                        Extra keywords are stored and passed through to
(12)                        :meth:`AbstractPathEffect._update_gc`.
(8)                     """
(8)                     super().__init__(offset)
(8)                     if shadow_rgbFace is None:
(12)                        self._shadow_rgbFace = shadow_rgbFace
(8)                     else:
(12)                        self._shadow_rgbFace = mcolors.to_rgba(shadow_rgbFace)
(8)                     if alpha is None:
(12)                        alpha = 0.3
(8)                     self._alpha = alpha
(8)                     self._rho = rho
(8)                     self._gc = kwargs
(4)                 def draw_path(self, renderer, gc, tpath, affine, rgbFace):
(8)                     """
(8)                     Overrides the standard draw_path to add the shadow offset and
(8)                     necessary color changes for the shadow.
(8)                     """
(8)                     gc0 = renderer.new_gc()  # Don't modify gc, but a copy!
(8)                     gc0.copy_properties(gc)
(8)                     if self._shadow_rgbFace is None:
(12)                        r, g, b = (rgbFace or (1., 1., 1.))[:3]
(12)                        shadow_rgbFace = (r * self._rho, g * self._rho, b * self._rho)
(8)                     else:
(12)                        shadow_rgbFace = self._shadow_rgbFace
(8)                     gc0.set_foreground("none")
(8)                     gc0.set_alpha(self._alpha)
(8)                     gc0.set_linewidth(0)
(8)                     gc0 = self._update_gc(gc0, self._gc)
(8)                     renderer.draw_path(
(12)                        gc0, tpath, affine + self._offset_transform(renderer),
(12)                        shadow_rgbFace)
(8)                     gc0.restore()
(0)             withSimplePatchShadow = _subclass_with_normal(effect_class=SimplePatchShadow)
(0)             class SimpleLineShadow(AbstractPathEffect):
(4)                 """A simple shadow via a line."""
(4)                 def __init__(self, offset=(2, -2),
(17)                            shadow_color='k', alpha=0.3, rho=0.3, **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     offset : (float, float), default: (2, -2)
(12)                        The (x, y) offset to apply to the path, in points.
(8)                     shadow_color : :mpltype:`color`, default: 'black'
(12)                        The shadow color.
(12)                        A value of ``None`` takes the original artist's color
(12)                        with a scale factor of *rho*.
(8)                     alpha : float, default: 0.3
(12)                        The alpha transparency of the created shadow patch.
(8)                     rho : float, default: 0.3
(12)                        A scale factor to apply to the rgbFace color if *shadow_color*
(12)                        is ``None``.
(8)                     **kwargs
(12)                        Extra keywords are stored and passed through to
(12)                        :meth:`AbstractPathEffect._update_gc`.
(8)                     """
(8)                     super().__init__(offset)
(8)                     if shadow_color is None:
```

```
(12)                        self._shadow_color = shadow_color
(8)                     else:
(12)                        self._shadow_color = mcolors.to_rgba(shadow_color)
(8)                     self._alpha = alpha
(8)                     self._rho = rho
(8)                     self._gc = kwargs
(4)                 def draw_path(self, renderer, gc, tpath, affine, rgbFace):
(8)                     """
(8)                     Overrides the standard draw_path to add the shadow offset and
(8)                     necessary color changes for the shadow.
(8)                     """
(8)                     gc0 = renderer.new_gc()  # Don't modify gc, but a copy!
(8)                     gc0.copy_properties(gc)
(8)                     if self._shadow_color is None:
(12)                        r, g, b = (gc0.get_foreground() or (1., 1., 1.))[:3]
(12)                        shadow_rgbFace = (r * self._rho, g * self._rho, b * self._rho)
(8)                     else:
(12)                        shadow_rgbFace = self._shadow_color
(8)                     gc0.set_foreground(shadow_rgbFace)
(8)                     gc0.set_alpha(self._alpha)
(8)                     gc0 = self._update_gc(gc0, self._gc)
(8)                     renderer.draw_path(
(12)                        gc0, tpath, affine + self._offset_transform(renderer))
(8)                     gc0.restore()
(0)             class PathPatchEffect(AbstractPathEffect):
(4)                 """
(4)                 Draws a `.PathPatch` instance whose Path comes from the original
(4)                 PathEffect artist.
(4)                 """
(4)                 def __init__(self, offset=(0, 0), **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     offset : (float, float), default: (0, 0)
(12)                        The (x, y) offset to apply to the path, in points.
(8)                     **kwargs
(12)                        All keyword arguments are passed through to the
(12)                        :class:`~matplotlib.patches.PathPatch` constructor. The
(12)                        properties which cannot be overridden are "path", "clip_box"
(12)                        "transform" and "clip_path".
(8)                     """
(8)                     super().__init__(offset=offset)
(8)                     self.patch = mpatches.PathPatch([], **kwargs)
(4)                 def draw_path(self, renderer, gc, tpath, affine, rgbFace):
(8)                     self.patch._path = tpath
(8)                     self.patch.set_transform(affine + self._offset_transform(renderer))
(8)                     self.patch.set_clip_box(gc.get_clip_rectangle())
(8)                     clip_path = gc.get_clip_path()
(8)                     if clip_path and self.patch.get_clip_path() is None:
(12)                        self.patch.set_clip_path(*clip_path)
(8)                     self.patch.draw(renderer)
(0)             class TickedStroke(AbstractPathEffect):
(4)                 """
(4)                 A line-based PathEffect which draws a path with a ticked style.
(4)                 This line style is frequently used to represent constraints in
(4)                 optimization.  The ticks may be used to indicate that one side
(4)                 of the line is invalid or to represent a closed boundary of a
(4)                 domain (i.e. a wall or the edge of a pipe).
(4)                 The spacing, length, and angle of ticks can be controlled.
(4)                 This line style is sometimes referred to as a hatched line.
(4)                 See also the :doc:`/gallery/misc/tickedstroke_demo` example.
(4)                 """
(4)                 def __init__(self, offset=(0, 0),
(17)                             spacing=10.0, angle=45.0, length=np.sqrt(2),
(17)                             **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     offset : (float, float), default: (0, 0)
```

```
(12)                        The (x, y) offset to apply to the path, in points.
(8)                     spacing : float, default: 10.0
(12)                        The spacing between ticks in points.
(8)                     angle : float, default: 45.0
(12)                        The angle between the path and the tick in degrees.  The angle
(12)                        is measured as if you were an ant walking along the curve, with
(12)                        zero degrees pointing directly ahead, 90 to your left, -90
(12)                        to your right, and 180 behind you. To change side of the ticks,
(12)                        change sign of the angle.
(8)                     length : float, default: 1.414
(12)                        The length of the tick relative to spacing.
(12)                        Recommended length = 1.414 (sqrt(2)) when angle=45, length=1.0
(12)                        when angle=90 and length=2.0 when angle=60.
(8)                     **kwargs
(12)                        Extra keywords are stored and passed through to
(12)                        :meth:`AbstractPathEffect._update_gc`.
(8)                     Examples
(8)                     --------
(8)                     See :doc:`/gallery/misc/tickedstroke_demo`.
(8)                     """
(8)                     super().__init__(offset)
(8)                     self._spacing = spacing
(8)                     self._angle = angle
(8)                     self._length = length
(8)                     self._gc = kwargs
(4)                 def draw_path(self, renderer, gc, tpath, affine, rgbFace):
(8)                     """Draw the path with updated gc."""
(8)                     gc0 = renderer.new_gc()
(8)                     gc0.copy_properties(gc)
(8)                     gc0 = self._update_gc(gc0, self._gc)
(8)                     trans = affine + self._offset_transform(renderer)
(8)                     theta = -np.radians(self._angle)
(8)                     trans_matrix = np.array([[np.cos(theta), -np.sin(theta)],
(33)                                             [np.sin(theta), np.cos(theta)]])
(8)                     spacing_px = renderer.points_to_pixels(self._spacing)
(8)                     transpath = affine.transform_path(tpath)
(8)                     polys = transpath.to_polygons(closed_only=False)
(8)                     for p in polys:
(12)                        x = p[:, 0]
(12)                        y = p[:, 1]
(12)                        if x.size < 2:
(16)                            continue
(12)                        ds = np.hypot(x[1:] - x[:-1], y[1:] - y[:-1])
(12)                        s = np.concatenate(([0.0], np.cumsum(ds)))
(12)                        s_total = s[-1]
(12)                        num = int(np.ceil(s_total / spacing_px)) - 1
(12)                        s_tick = np.linspace(spacing_px/2, s_total - spacing_px/2, num)
(12)                        x_tick = np.interp(s_tick, s, x)
(12)                        y_tick = np.interp(s_tick, s, y)
(12)                        delta_s = self._spacing * .001
(12)                        u = (np.interp(s_tick + delta_s, s, x) - x_tick) / delta_s
(12)                        v = (np.interp(s_tick + delta_s, s, y) - y_tick) / delta_s
(12)                        n = np.hypot(u, v)
(12)                        mask = n == 0
(12)                        n[mask] = 1.0
(12)                        uv = np.array([u / n, v / n]).T
(12)                        uv[mask] = np.array([0, 0]).T
(12)                        dxy = np.dot(uv, trans_matrix) * self._length * spacing_px
(12)                        x_end = x_tick + dxy[:, 0]
(12)                        y_end = y_tick + dxy[:, 1]
(12)                        xyt = np.empty((2 * num, 2), dtype=x_tick.dtype)
(12)                        xyt[0::2, 0] = x_tick
(12)                        xyt[1::2, 0] = x_end
(12)                        xyt[0::2, 1] = y_tick
(12)                        xyt[1::2, 1] = y_end
(12)                        codes = np.tile([Path.MOVETO, Path.LINETO], num)
(12)                        h = Path(xyt, codes)
(12)                        renderer.draw_path(gc0, h, affine.inverted() + trans, rgbFace)
(8)                     gc0.restore()
```

```
376: (0)            withTickedStroke = _subclass_with_normal(effect_class=TickedStroke)

----------------------------------------

File 53 - path.py:

1: (0)              r"""
2: (0)              A module for dealing with the polylines used throughout Matplotlib.
3: (0)              The primary class for polyline handling in Matplotlib is `Path`.  Almost all
4: (0)              vector drawing makes use of `Path`\s somewhere in the drawing pipeline.
5: (0)              Whilst a `Path` instance itself cannot be drawn, some `.Artist` subclasses,
6: (0)              such as `.PathPatch` and `.PathCollection`, can be used for convenient `Path`
7: (0)              visualisation.
8: (0)              """
9: (0)              import copy
10: (0)             from functools import lru_cache
11: (0)             from weakref import WeakValueDictionary
12: (0)             import numpy as np
13: (0)             import matplotlib as mpl
14: (0)             from . import _api, _path
15: (0)             from .cbook import _to_unmasked_float_array, simple_linear_interpolation
16: (0)             from .bezier import BezierSegment
17: (0)             class Path:
18: (4)                 """
19: (4)                 A series of possibly disconnected, possibly closed, line and curve
20: (4)                 segments.
21: (4)                 The underlying storage is made up of two parallel numpy arrays:
22: (4)                 - *vertices*: an (N, 2) float array of vertices
23: (4)                 - *codes*: an N-length `numpy.uint8` array of path codes, or None
24: (4)                 These two arrays always have the same length in the first
25: (4)                 dimension.  For example, to represent a cubic curve, you must
26: (4)                 provide three vertices and three `CURVE4` codes.
27: (4)                 The code types are:
28: (4)                 - `STOP`   :  1 vertex (ignored)
29: (8)                     A marker for the end of the entire path (currently not required and
30: (8)                     ignored)
31: (4)                 - `MOVETO` :  1 vertex
32: (8)                     Pick up the pen and move to the given vertex.
33: (4)                 - `LINETO` :  1 vertex
34: (8)                     Draw a line from the current position to the given vertex.
35: (4)                 - `CURVE3` :  1 control point, 1 endpoint
36: (8)                     Draw a quadratic Bézier curve from the current position, with the
given
37: (8)                     control point, to the given end point.
38: (4)                 - `CURVE4` :  2 control points, 1 endpoint
39: (8)                     Draw a cubic Bézier curve from the current position, with the given
40: (8)                     control points, to the given end point.
41: (4)                 - `CLOSEPOLY` : 1 vertex (ignored)
42: (8)                     Draw a line segment to the start point of the current polyline.
43: (4)                 If *codes* is None, it is interpreted as a `MOVETO` followed by a series
44: (4)                 of `LINETO`.
45: (4)                 Users of Path objects should not access the vertices and codes arrays
46: (4)                 directly.  Instead, they should use `iter_segments` or `cleaned` to get
the
47: (4)                 vertex/code pairs.  This helps, in particular, to consistently handle the
48: (4)                 case of *codes* being None.
49: (4)                 Some behavior of Path objects can be controlled by rcParams. See the
50: (4)                 rcParams whose keys start with 'path.'.
51: (4)                 .. note::
52: (8)                     The vertices and codes arrays should be treated as
53: (8)                     immutable -- there are a number of optimizations and assumptions
54: (8)                     made up front in the constructor that will not change when the
55: (8)                     data changes.
56: (4)                 """
57: (4)                 code_type = np.uint8
58: (4)                 STOP = code_type(0)         # 1 vertex
59: (4)                 MOVETO = code_type(1)       # 1 vertex
60: (4)                 LINETO = code_type(2)       # 1 vertex
61: (4)                 CURVE3 = code_type(3)       # 2 vertices
```

```
(4)                 CURVE4 = code_type(4)       # 3 vertices
(4)                 CLOSEPOLY = code_type(79)   # 1 vertex
(4)                 NUM_VERTICES_FOR_CODE = {STOP: 1,
(29)                                         MOVETO: 1,
(29)                                         LINETO: 1,
(29)                                         CURVE3: 2,
(29)                                         CURVE4: 3,
(29)                                         CLOSEPOLY: 1}
(4)                 def __init__(self, vertices, codes=None, _interpolation_steps=1,
(17)                             closed=False, readonly=False):
(8)                     """
(8)                     Create a new path with the given vertices and codes.
(8)                     Parameters
(8)                     ----------
(8)                     vertices : (N, 2) array-like
(12)                        The path vertices, as an array, masked array or sequence of pairs.
(12)                        Masked values, if any, will be converted to NaNs, which are then
(12)                        handled correctly by the Agg PathIterator and other consumers of
(12)                        path data, such as :meth:`iter_segments`.
(8)                     codes : array-like or None, optional
(12)                        N-length array of integers representing the codes of the path.
(12)                        If not None, codes must be the same length as vertices.
(12)                        If None, *vertices* will be treated as a series of line segments.
(8)                     _interpolation_steps : int, optional
(12)                        Used as a hint to certain projections, such as Polar, that this
(12)                        path should be linearly interpolated immediately before drawing.
(12)                        This attribute is primarily an implementation detail and is not
(12)                        intended for public use.
(8)                     closed : bool, optional
(12)                        If *codes* is None and closed is True, vertices will be treated as
(12)                        line segments of a closed polygon.  Note that the last vertex will
(12)                        then be ignored (as the corresponding code will be set to
(12)                        `CLOSEPOLY`).
(8)                     readonly : bool, optional
(12)                        Makes the path behave in an immutable way and sets the vertices
(12)                        and codes as read-only arrays.
(8)                     """
(8)                     vertices = _to_unmasked_float_array(vertices)
(8)                     _api.check_shape((None, 2), vertices=vertices)
(8)                     if codes is not None:
(12)                        codes = np.asarray(codes, self.code_type)
(12)                        if codes.ndim != 1 or len(codes) != len(vertices):
(16)                            raise ValueError("'codes' must be a 1D list or array with the
"
(33)                                             "same length of 'vertices'. "
(33)                                             f"Your vertices have shape {vertices.shape} "
(33)                                             f"but your codes have shape {codes.shape}")
(12)                        if len(codes) and codes[0] != self.MOVETO:
(16)                            raise ValueError("The first element of 'code' must be equal "
(33)                                             f"to 'MOVETO' ({self.MOVETO}).  "
(33)                                             f"Your first code is {codes[0]}")
(8)                     elif closed and len(vertices):
(12)                        codes = np.empty(len(vertices), dtype=self.code_type)
(12)                        codes[0] = self.MOVETO
(12)                        codes[1:-1] = self.LINETO
(12)                        codes[-1] = self.CLOSEPOLY
(8)                     self._vertices = vertices
(8)                     self._codes = codes
(8)                     self._interpolation_steps = _interpolation_steps
(8)                     self._update_values()
(8)                     if readonly:
(12)                        self._vertices.flags.writeable = False
(12)                        if self._codes is not None:
(16)                            self._codes.flags.writeable = False
(12)                        self._readonly = True
(8)                     else:
(12)                        self._readonly = False
(4)                 @classmethod
(4)                 def _fast_from_codes_and_verts(cls, verts, codes, internals_from=None):
```

```
(8)                 """
(8)                 Create a Path instance without the expense of calling the constructor.
(8)                 Parameters
(8)                 ----------
(8)                 verts : array-like
(8)                 codes : array
(8)                 internals_from : Path or None
(12)                    If not None, another `Path` from which the attributes
(12)                    ``should_simplify``, ``simplify_threshold``, and
(12)                    ``interpolation_steps`` will be copied.  Note that ``readonly`` is
(12)                    never copied, and always set to ``False`` by this constructor.
(8)                 """
(8)                 pth = cls.__new__(cls)
(8)                 pth._vertices = _to_unmasked_float_array(verts)
(8)                 pth._codes = codes
(8)                 pth._readonly = False
(8)                 if internals_from is not None:
(12)                    pth._should_simplify = internals_from._should_simplify
(12)                    pth._simplify_threshold = internals_from._simplify_threshold
(12)                    pth._interpolation_steps = internals_from._interpolation_steps
(8)                 else:
(12)                    pth._should_simplify = True
(12)                    pth._simplify_threshold = mpl.rcParams['path.simplify_threshold']
(12)                    pth._interpolation_steps = 1
(8)                 return pth
(4)             @classmethod
(4)             def _create_closed(cls, vertices):
(8)                 """
(8)                 Create a closed polygonal path going through *vertices*.
(8)                 Unlike ``Path(..., closed=True)``, *vertices* should **not** end with
(8)                 an entry for the CLOSEPATH; this entry is added by `._create_closed`.
(8)                 """
(8)                 v = _to_unmasked_float_array(vertices)
(8)                 return cls(np.concatenate([v, v[:1]]), closed=True)
(4)             def _update_values(self):
(8)                 self._simplify_threshold = mpl.rcParams['path.simplify_threshold']
(8)                 self._should_simplify = (
(12)                    self._simplify_threshold > 0 and
(12)                    mpl.rcParams['path.simplify'] and
(12)                    len(self._vertices) >= 128 and
(12)                    (self._codes is None or np.all(self._codes <= Path.LINETO))
(8)                 )
(4)             @property
(4)             def vertices(self):
(8)                 """The vertices of the `Path` as an (N, 2) array."""
(8)                 return self._vertices
(4)             @vertices.setter
(4)             def vertices(self, vertices):
(8)                 if self._readonly:
(12)                    raise AttributeError("Can't set vertices on a readonly Path")
(8)                 self._vertices = vertices
(8)                 self._update_values()
(4)             @property
(4)             def codes(self):
(8)                 """
(8)                 The list of codes in the `Path` as a 1D array.
(8)                 Each code is one of `STOP`, `MOVETO`, `LINETO`, `CURVE3`, `CURVE4` or
(8)                 `CLOSEPOLY`.  For codes that correspond to more than one vertex
(8)                 (`CURVE3` and `CURVE4`), that code will be repeated so that the length
(8)                 of `vertices` and `codes` is always the same.
(8)                 """
(8)                 return self._codes
(4)             @codes.setter
(4)             def codes(self, codes):
(8)                 if self._readonly:
(12)                    raise AttributeError("Can't set codes on a readonly Path")
(8)                 self._codes = codes
(8)                 self._update_values()
(4)             @property
```

```
    def simplify_threshold(self):
        """
        The fraction of a pixel difference below which vertices will
        be simplified out.
        """
        return self._simplify_threshold
    @simplify_threshold.setter
    def simplify_threshold(self, threshold):
        self._simplify_threshold = threshold
    @property
    def should_simplify(self):
        """
        `True` if the vertices array should be simplified.
        """
        return self._should_simplify
    @should_simplify.setter
    def should_simplify(self, should_simplify):
        self._should_simplify = should_simplify
    @property
    def readonly(self):
        """
        `True` if the `Path` is read-only.
        """
        return self._readonly
    def copy(self):
        """
        Return a shallow copy of the `Path`, which will share the
        vertices and codes with the source `Path`.
        """
        return copy.copy(self)
    def __deepcopy__(self, memo=None):
        """
        Return a deepcopy of the `Path`.  The `Path` will not be
        readonly, even if the source `Path` is.
        """
        p = copy.deepcopy(super(), memo)
        p._readonly = False
        return p
    deepcopy = __deepcopy__
    @classmethod
    def make_compound_path_from_polys(cls, XY):
        """
        Make a compound `Path` object to draw a number of polygons with equal
        numbers of sides.
        .. plot:: gallery/misc/histogram_path.py
        Parameters
        ----------
        XY : (numpolys, numsides, 2) array
        """
        numpolys, numsides, two = XY.shape
        if two != 2:
            raise ValueError("The third dimension of 'XY' must be 2")
        stride = numsides + 1
        nverts = numpolys * stride
        verts = np.zeros((nverts, 2))
        codes = np.full(nverts, cls.LINETO, dtype=cls.code_type)
        codes[0::stride] = cls.MOVETO
        codes[numsides::stride] = cls.CLOSEPOLY
        for i in range(numsides):
            verts[i::stride] = XY[:, i]
        return cls(verts, codes)
    @classmethod
    def make_compound_path(cls, *args):
        r"""
        Concatenate a list of `Path`\s into a single `Path`, removing all
`STOP`\s.
        """
        if not args:
            return Path(np.empty([0, 2], dtype=np.float32))
```

```
267: (8)                    vertices = np.concatenate([path.vertices for path in args])
268: (8)                    codes = np.empty(len(vertices), dtype=cls.code_type)
269: (8)                    i = 0
270: (8)                    for path in args:
271: (12)                       size = len(path.vertices)
272: (12)                       if path.codes is None:
273: (16)                           if size:
274: (20)                               codes[i] = cls.MOVETO
275: (20)                               codes[i+1:i+size] = cls.LINETO
276: (12)                       else:
277: (16)                           codes[i:i+size] = path.codes
278: (12)                       i += size
279: (8)                    not_stop_mask = codes != cls.STOP  # Remove STOPs, as internal STOPs
are a bug.
280: (8)                    return cls(vertices[not_stop_mask], codes[not_stop_mask])
281: (4)                def __repr__(self):
282: (8)                    return f"Path({self.vertices!r}, {self.codes!r})"
283: (4)                def __len__(self):
284: (8)                    return len(self.vertices)
285: (4)                def iter_segments(self, transform=None, remove_nans=True, clip=None,
286: (22)                                 snap=False, stroke_width=1.0, simplify=None,
287: (22)                                 curves=True, sketch=None):
288: (8)                    """
289: (8)                    Iterate over all curve segments in the path.
290: (8)                    Each iteration returns a pair ``(vertices, code)``, where ``vertices``
291: (8)                    is a sequence of 1-3 coordinate pairs, and ``code`` is a `Path` code.
292: (8)                    Additionally, this method can provide a number of standard cleanups
and
293: (8)                    conversions to the path.
294: (8)                    Parameters
295: (8)                    ----------
296: (8)                    transform : None or :class:`~matplotlib.transforms.Transform`
297: (12)                       If not None, the given affine transformation will be applied to
the
298: (12)                       path.
299: (8)                    remove_nans : bool, optional
300: (12)                       Whether to remove all NaNs from the path and skip over them using
301: (12)                       MOVETO commands.
302: (8)                    clip : None or (float, float, float, float), optional
303: (12)                       If not None, must be a four-tuple (x1, y1, x2, y2)
304: (12)                       defining a rectangle in which to clip the path.
305: (8)                    snap : None or bool, optional
306: (12)                       If True, snap all nodes to pixels; if False, don't snap them.
307: (12)                       If None, snap if the path contains only segments
308: (12)                       parallel to the x or y axes, and no more than 1024 of them.
309: (8)                    stroke_width : float, optional
310: (12)                       The width of the stroke being drawn (used for path snapping).
311: (8)                    simplify : None or bool, optional
312: (12)                       Whether to simplify the path by removing vertices
313: (12)                       that do not affect its appearance.  If None, use the
314: (12)                       :attr:`should_simplify` attribute.  See also :rc:`path.simplify`
315: (12)                       and :rc:`path.simplify_threshold`.
316: (8)                    curves : bool, optional
317: (12)                       If True, curve segments will be returned as curve segments.
318: (12)                       If False, all curves will be converted to line segments.
319: (8)                    sketch : None or sequence, optional
320: (12)                       If not None, must be a 3-tuple of the form
321: (12)                       (scale, length, randomness), representing the sketch parameters.
322: (8)                    """
323: (8)                    if not len(self):
324: (12)                       return
325: (8)                    cleaned = self.cleaned(transform=transform,
326: (31)                                          remove_nans=remove_nans, clip=clip,
327: (31)                                          snap=snap, stroke_width=stroke_width,
328: (31)                                          simplify=simplify, curves=curves,
329: (31)                                          sketch=sketch)
330: (8)                    NUM_VERTICES_FOR_CODE = self.NUM_VERTICES_FOR_CODE
331: (8)                    STOP = self.STOP
332: (8)                    vertices = iter(cleaned.vertices)
```

```
(8)                     codes = iter(cleaned.codes)
(8)                     for curr_vertices, code in zip(vertices, codes):
(12)                        if code == STOP:
(16)                            break
(12)                        extra_vertices = NUM_VERTICES_FOR_CODE[code] - 1
(12)                        if extra_vertices:
(16)                            for i in range(extra_vertices):
(20)                                next(codes)
(20)                                curr_vertices = np.append(curr_vertices, next(vertices))
(12)                        yield curr_vertices, code
(4)                 def iter_bezier(self, **kwargs):
(8)                     """
(8)                     Iterate over each Bézier curve (lines included) in a `Path`.
(8)                     Parameters
(8)                     ----------
(8)                     **kwargs
(12)                        Forwarded to `.iter_segments`.
(8)                     Yields
(8)                     ------
(8)                     B : `~matplotlib.bezier.BezierSegment`
(12)                        The Bézier curves that make up the current path. Note in particular
(12)                        that freestanding points are Bézier curves of order 0, and lines
(12)                        are Bézier curves of order 1 (with two control points).
(8)                     code : `~matplotlib.path.Path.code_type`
(12)                        The code describing what kind of curve is being returned.
(12)                        `MOVETO`, `LINETO`, `CURVE3`, and `CURVE4` correspond to
(12)                        Bézier curves with 1, 2, 3, and 4 control points (respectively).
(12)                        `CLOSEPOLY` is a `LINETO` with the control points correctly
(12)                        chosen based on the start/end points of the current stroke.
(8)                     """
(8)                     first_vert = None
(8)                     prev_vert = None
(8)                     for verts, code in self.iter_segments(**kwargs):
(12)                        if first_vert is None:
(16)                            if code != Path.MOVETO:
(20)                                raise ValueError("Malformed path, must start with MOVETO.")
(12)                        if code == Path.MOVETO:  # a point is like "CURVE1"
(16)                            first_vert = verts
(16)                            yield BezierSegment(np.array([first_vert])), code
(12)                        elif code == Path.LINETO:  # "CURVE2"
(16)                            yield BezierSegment(np.array([prev_vert, verts])), code
(12)                        elif code == Path.CURVE3:
(16)                            yield BezierSegment(np.array([prev_vert, verts[:2],
(46)                                                          verts[2:]])), code
(12)                        elif code == Path.CURVE4:
(16)                            yield BezierSegment(np.array([prev_vert, verts[:2],
(46)                                                          verts[2:4], verts[4:]])), code
(12)                        elif code == Path.CLOSEPOLY:
(16)                            yield BezierSegment(np.array([prev_vert, first_vert])), code
(12)                        elif code == Path.STOP:
(16)                            return
(12)                        else:
(16)                            raise ValueError(f"Invalid Path.code_type: {code}")
(12)                        prev_vert = verts[-2:]
(4)                 def _iter_connected_components(self):
(8)                     """Return subpaths split at MOVETOs."""
(8)                     if self.codes is None:
(12)                        yield self
(8)                     else:
(12)                        idxs = np.append((self.codes == Path.MOVETO).nonzero()[0], len(self.codes))
(12)                        for sl in map(slice, idxs, idxs[1:]):
(16)                            yield Path._fast_from_codes_and_verts(
(20)                                self.vertices[sl], self.codes[sl], self)
(4)                 def cleaned(self, transform=None, remove_nans=False, clip=None,
(16)                            *, simplify=False, curves=False,
(16)                            stroke_width=1.0, snap=False, sketch=None):
```

```
399: (8)                    """
400: (8)                    Return a new `Path` with vertices and codes cleaned according to the
401: (8)                    parameters.
402: (8)                    See Also
403: (8)                    --------
404: (8)                    Path.iter_segments : for details of the keyword arguments.
405: (8)                    """
406: (8)                    vertices, codes = _path.cleanup_path(
407: (12)                       self, transform, remove_nans, clip, snap, stroke_width, simplify,
408: (12)                       curves, sketch)
409: (8)                    pth = Path._fast_from_codes_and_verts(vertices, codes, self)
410: (8)                    if not simplify:
411: (12)                       pth._should_simplify = False
412: (8)                    return pth
413: (4)                def transformed(self, transform):
414: (8)                    """
415: (8)                    Return a transformed copy of the path.
416: (8)                    See Also
417: (8)                    --------
418: (8)                    matplotlib.transforms.TransformedPath
419: (12)                       A specialized path class that will cache the transformed result
and
420: (12)                       automatically update when the transform changes.
421: (8)                    """
422: (8)                    return Path(transform.transform(self.vertices), self.codes,
423: (20)                               self._interpolation_steps)
424: (4)                def contains_point(self, point, transform=None, radius=0.0):
425: (8)                    """
426: (8)                    Return whether the area enclosed by the path contains the given point.
427: (8)                    The path is always treated as closed; i.e. if the last code is not
428: (8)                    `CLOSEPOLY` an implicit segment connecting the last vertex to the
first
429: (8)                    vertex is assumed.
430: (8)                    Parameters
431: (8)                    ----------
432: (8)                    point : (float, float)
433: (12)                       The point (x, y) to check.
434: (8)                    transform : `~matplotlib.transforms.Transform`, optional
435: (12)                       If not ``None``, *point* will be compared to ``self`` transformed
436: (12)                       by *transform*; i.e. for a correct check, *transform* should
437: (12)                       transform the path into the coordinate system of *point*.
438: (8)                    radius : float, default: 0
439: (12)                       Additional margin on the path in coordinates of *point*.
440: (12)                       The path is extended tangentially by *radius/2*; i.e. if you would
441: (12)                       draw the path with a linewidth of *radius*, all points on the line
442: (12)                       would still be considered to be contained in the area. Conversely,
443: (12)                       negative values shrink the area: Points on the imaginary line
444: (12)                       will be considered outside the area.
445: (8)                    Returns
446: (8)                    -------
447: (8)                    bool
448: (8)                    Notes
449: (8)                    -----
450: (8)                    The current algorithm has some limitations:
451: (8)                    - The result is undefined for points exactly at the boundary
452: (10)                     (i.e. at the path shifted by *radius/2*).
453: (8)                    - The result is undefined if there is no enclosed area, i.e. all
454: (10)                     vertices are on a straight line.
455: (8)                    - If bounding lines start to cross each other due to *radius* shift,
456: (10)                     the result is not guaranteed to be correct.
457: (8)                    """
458: (8)                    if transform is not None:
459: (12)                       transform = transform.frozen()
460: (8)                    if transform and not transform.is_affine:
461: (12)                       self = transform.transform_path(self)
462: (12)                       transform = None
463: (8)                    return _path.point_in_path(point[0], point[1], radius, self,
transform)
464: (4)                def contains_points(self, points, transform=None, radius=0.0):
```

```
(8)                 """
(8)                 Return whether the area enclosed by the path contains the given
points.
(8)                 The path is always treated as closed; i.e. if the last code is not
(8)                 `CLOSEPOLY` an implicit segment connecting the last vertex to the
first
(8)                 vertex is assumed.
(8)                 Parameters
(8)                 ----------
(8)                 points : (N, 2) array
(12)                    The points to check. Columns contain x and y values.
(8)                 transform : `~matplotlib.transforms.Transform`, optional
(12)                    If not ``None``, *points* will be compared to ``self`` transformed
(12)                    by *transform*; i.e. for a correct check, *transform* should
(12)                    transform the path into the coordinate system of *points*.
(8)                 radius : float, default: 0
(12)                    Additional margin on the path in coordinates of *points*.
(12)                    The path is extended tangentially by *radius/2*; i.e. if you would
(12)                    draw the path with a linewidth of *radius*, all points on the line
(12)                    would still be considered to be contained in the area. Conversely,
(12)                    negative values shrink the area: Points on the imaginary line
(12)                    will be considered outside the area.
(8)                 Returns
(8)                 -------
(8)                 length-N bool array
(8)                 Notes
(8)                 -----
(8)                 The current algorithm has some limitations:
(8)                 - The result is undefined for points exactly at the boundary
(10)                  (i.e. at the path shifted by *radius/2*).
(8)                 - The result is undefined if there is no enclosed area, i.e. all
(10)                  vertices are on a straight line.
(8)                 - If bounding lines start to cross each other due to *radius* shift,
(10)                  the result is not guaranteed to be correct.
(8)                 """
(8)                 if transform is not None:
(12)                    transform = transform.frozen()
(8)                 result = _path.points_in_path(points, radius, self, transform)
(8)                 return result.astype('bool')
(4)             def contains_path(self, path, transform=None):
(8)                 """
(8)                 Return whether this (closed) path completely contains the given path.
(8)                 If *transform* is not ``None``, the path will be transformed before
(8)                 checking for containment.
(8)                 """
(8)                 if transform is not None:
(12)                    transform = transform.frozen()
(8)                 return _path.path_in_path(self, None, path, transform)
(4)             def get_extents(self, transform=None, **kwargs):
(8)                 """
(8)                 Get Bbox of the path.
(8)                 Parameters
(8)                 ----------
(8)                 transform : `~matplotlib.transforms.Transform`, optional
(12)                    Transform to apply to path before computing extents, if any.
(8)                 **kwargs
(12)                    Forwarded to `.iter_bezier`.
(8)                 Returns
(8)                 -------
(8)                 matplotlib.transforms.Bbox
(12)                    The extents of the path Bbox([[xmin, ymin], [xmax, ymax]])
(8)                 """
(8)                 from .transforms import Bbox
(8)                 if transform is not None:
(12)                    self = transform.transform_path(self)
(8)                 if self.codes is None:
(12)                    xys = self.vertices
(8)                 elif len(np.intersect1d(self.codes, [Path.CURVE3, Path.CURVE4])) == 0:
(12)                    xys = self.vertices[np.isin(self.codes,
```

```
532: (40)                                                   [Path.MOVETO, Path.LINETO])]
533: (8)                    else:
534: (12)                       xys = []
535: (12)                       for curve, code in self.iter_bezier(**kwargs):
536: (16)                           _, dzeros = curve.axis_aligned_extrema()
537: (16)                           xys.append(curve([0, *dzeros, 1]))
538: (12)                       xys = np.concatenate(xys)
539: (8)                    if len(xys):
540: (12)                       return Bbox([xys.min(axis=0), xys.max(axis=0)])
541: (8)                    else:
542: (12)                       return Bbox.null()
543: (4)                def intersects_path(self, other, filled=True):
544: (8)                    """
545: (8)                    Return whether if this path intersects another given path.
546: (8)                    If *filled* is True, then this also returns True if one path
completely
547: (8)                    encloses the other (i.e., the paths are treated as filled).
548: (8)                    """
549: (8)                    return _path.path_intersects_path(self, other, filled)
550: (4)                def intersects_bbox(self, bbox, filled=True):
551: (8)                    """
552: (8)                    Return whether this path intersects a given `~.transforms.Bbox`.
553: (8)                    If *filled* is True, then this also returns True if the path
completely
554: (8)                    encloses the `.Bbox` (i.e., the path is treated as filled).
555: (8)                    The bounding box is always considered filled.
556: (8)                    """
557: (8)                    return _path.path_intersects_rectangle(
558: (12)                       self, bbox.x0, bbox.y0, bbox.x1, bbox.y1, filled)
559: (4)                def interpolated(self, steps):
560: (8)                    """
561: (8)                    Return a new path resampled to length N x *steps*.
562: (8)                    Codes other than `LINETO` are not handled correctly.
563: (8)                    """
564: (8)                    if steps == 1:
565: (12)                       return self
566: (8)                    vertices = simple_linear_interpolation(self.vertices, steps)
567: (8)                    codes = self.codes
568: (8)                    if codes is not None:
569: (12)                       new_codes = np.full((len(codes) - 1) * steps + 1, Path.LINETO,
570: (32)                                           dtype=self.code_type)
571: (12)                       new_codes[0::steps] = codes
572: (8)                    else:
573: (12)                       new_codes = None
574: (8)                    return Path(vertices, new_codes)
575: (4)                def to_polygons(self, transform=None, width=0, height=0,
closed_only=True):
576: (8)                    """
577: (8)                    Convert this path to a list of polygons or polylines.  Each
578: (8)                    polygon/polyline is an (N, 2) array of vertices.  In other words,
579: (8)                    each polygon has no `MOVETO` instructions or curves.  This
580: (8)                    is useful for displaying in backends that do not support
581: (8)                    compound paths or Bézier curves.
582: (8)                    If *width* and *height* are both non-zero then the lines will
583: (8)                    be simplified so that vertices outside of (0, 0), (width,
584: (8)                    height) will be clipped.
585: (8)                    The resulting polygons will be simplified if the
586: (8)                    :attr:`Path.should_simplify` attribute of the path is `True`.
587: (8)                    If *closed_only* is `True` (default), only closed polygons,
588: (8)                    with the last point being the same as the first point, will be
589: (8)                    returned.  Any unclosed polylines in the path will be
590: (8)                    explicitly closed.  If *closed_only* is `False`, any unclosed
591: (8)                    polygons in the path will be returned as unclosed polygons,
592: (8)                    and the closed polygons will be returned explicitly closed by
593: (8)                    setting the last point to the same as the first point.
594: (8)                    """
595: (8)                    if len(self.vertices) == 0:
596: (12)                       return []
597: (8)                    if transform is not None:
```

```
598: (12)                        transform = transform.frozen()
599: (8)                    if self.codes is None and (width == 0 or height == 0):
600: (12)                        vertices = self.vertices
601: (12)                        if closed_only:
602: (16)                            if len(vertices) < 3:
603: (20)                                return []
604: (16)                            elif np.any(vertices[0] != vertices[-1]):
605: (20)                                vertices = [*vertices, vertices[0]]
606: (12)                        if transform is None:
607: (16)                            return [vertices]
608: (12)                        else:
609: (16)                            return [transform.transform(vertices)]
610: (8)                    return _path.convert_path_to_polygons(
611: (12)                        self, transform, width, height, closed_only)
612: (4)                _unit_rectangle = None
613: (4)                @classmethod
614: (4)                def unit_rectangle(cls):
615: (8)                    """
616: (8)                    Return a `Path` instance of the unit rectangle from (0, 0) to (1, 1).
617: (8)                    """
618: (8)                    if cls._unit_rectangle is None:
619: (12)                       cls._unit_rectangle = cls([[0, 0], [1, 0], [1, 1], [0, 1], [0,
0]],
620: (38)                                                 closed=True, readonly=True)
621: (8)                    return cls._unit_rectangle
622: (4)                _unit_regular_polygons = WeakValueDictionary()
623: (4)                @classmethod
624: (4)                def unit_regular_polygon(cls, numVertices):
625: (8)                    """
626: (8)                    Return a :class:`Path` instance for a unit regular polygon with the
627: (8)                    given *numVertices* such that the circumscribing circle has radius
1.0,
628: (8)                    centered at (0, 0).
629: (8)                    """
630: (8)                    if numVertices <= 16:
631: (12)                       path = cls._unit_regular_polygons.get(numVertices)
632: (8)                    else:
633: (12)                       path = None
634: (8)                    if path is None:
635: (12)                       theta = ((2 * np.pi / numVertices) * np.arange(numVertices + 1)
636: (21)                                + np.pi / 2)
637: (12)                       verts = np.column_stack((np.cos(theta), np.sin(theta)))
638: (12)                       path = cls(verts, closed=True, readonly=True)
639: (12)                       if numVertices <= 16:
640: (16)                           cls._unit_regular_polygons[numVertices] = path
641: (8)                    return path
642: (4)                _unit_regular_stars = WeakValueDictionary()
643: (4)                @classmethod
644: (4)                def unit_regular_star(cls, numVertices, innerCircle=0.5):
645: (8)                    """
646: (8)                    Return a :class:`Path` for a unit regular star with the given
647: (8)                    numVertices and radius of 1.0, centered at (0, 0).
648: (8)                    """
649: (8)                    if numVertices <= 16:
650: (12)                       path = cls._unit_regular_stars.get((numVertices, innerCircle))
651: (8)                    else:
652: (12)                       path = None
653: (8)                    if path is None:
654: (12)                       ns2 = numVertices * 2
655: (12)                       theta = (2*np.pi/ns2 * np.arange(ns2 + 1))
656: (12)                       theta += np.pi / 2.0
657: (12)                       r = np.ones(ns2 + 1)
658: (12)                       r[1::2] = innerCircle
659: (12)                       verts = (r * np.vstack((np.cos(theta), np.sin(theta)))).T
660: (12)                       path = cls(verts, closed=True, readonly=True)
661: (12)                       if numVertices <= 16:
662: (16)                           cls._unit_regular_stars[(numVertices, innerCircle)] = path
663: (8)                    return path
664: (4)                @classmethod
```

```
(4)     def unit_regular_asterisk(cls, numVertices):
(8)         """
(8)         Return a :class:`Path` for a unit regular asterisk with the given
(8)         numVertices and radius of 1.0, centered at (0, 0).
(8)         """
(8)         return cls.unit_regular_star(numVertices, 0.0)
(4)     _unit_circle = None
(4)     @classmethod
(4)     def unit_circle(cls):
(8)         """
(8)         Return the readonly :class:`Path` of the unit circle.
(8)         For most cases, :func:`Path.circle` will be what you want.
(8)         """
(8)         if cls._unit_circle is None:
(12)            cls._unit_circle = cls.circle(center=(0, 0), radius=1,
(42)                                          readonly=True)
(8)         return cls._unit_circle
(4)     @classmethod
(4)     def circle(cls, center=(0., 0.), radius=1., readonly=False):
(8)         """
(8)         Return a `Path` representing a circle of a given radius and center.
(8)         Parameters
(8)         ----------
(8)         center : (float, float), default: (0, 0)
(12)            The center of the circle.
(8)         radius : float, default: 1
(12)            The radius of the circle.
(8)         readonly : bool
(12)            Whether the created path should have the "readonly" argument
(12)            set when creating the Path instance.
(8)         Notes
(8)         -----
(8)         The circle is approximated using 8 cubic Bézier curves, as described
in
(10)          Lancaster, Don.  `Approximating a Circle or an Ellipse Using Four
(10)          Bezier Cubic Splines <https://www.tinaja.com/glib/ellipse4.pdf>`_.
(8)         """
(8)         MAGIC = 0.2652031
(8)         SQRTHALF = np.sqrt(0.5)
(8)         MAGIC45 = SQRTHALF * MAGIC
(8)         vertices = np.array([[0.0, -1.0],
(29)                             [MAGIC, -1.0],
(29)                             [SQRTHALF-MAGIC45, -SQRTHALF-MAGIC45],
(29)                             [SQRTHALF, -SQRTHALF],
(29)                             [SQRTHALF+MAGIC45, -SQRTHALF+MAGIC45],
(29)                             [1.0, -MAGIC],
(29)                             [1.0, 0.0],
(29)                             [1.0, MAGIC],
(29)                             [SQRTHALF+MAGIC45, SQRTHALF-MAGIC45],
(29)                             [SQRTHALF, SQRTHALF],
(29)                             [SQRTHALF-MAGIC45, SQRTHALF+MAGIC45],
(29)                             [MAGIC, 1.0],
(29)                             [0.0, 1.0],
(29)                             [-MAGIC, 1.0],
(29)                             [-SQRTHALF+MAGIC45, SQRTHALF+MAGIC45],
(29)                             [-SQRTHALF, SQRTHALF],
(29)                             [-SQRTHALF-MAGIC45, SQRTHALF-MAGIC45],
(29)                             [-1.0, MAGIC],
(29)                             [-1.0, 0.0],
(29)                             [-1.0, -MAGIC],
(29)                             [-SQRTHALF-MAGIC45, -SQRTHALF+MAGIC45],
(29)                             [-SQRTHALF, -SQRTHALF],
(29)                             [-SQRTHALF+MAGIC45, -SQRTHALF-MAGIC45],
(29)                             [-MAGIC, -1.0],
(29)                             [0.0, -1.0],
(29)                             [0.0, -1.0]],
(28)                            dtype=float)
(8)         codes = [cls.CURVE4] * 26
(8)         codes[0] = cls.MOVETO
```

```
(8)                 codes[-1] = cls.CLOSEPOLY
(8)                 return Path(vertices * radius + center, codes, readonly=readonly)
(4)             _unit_circle_righthalf = None
(4)             @classmethod
(4)             def unit_circle_righthalf(cls):
(8)                 """
(8)                 Return a `Path` of the right half of a unit circle.
(8)                 See `Path.circle` for the reference on the approximation used.
(8)                 """
(8)                 if cls._unit_circle_righthalf is None:
(12)                    MAGIC = 0.2652031
(12)                    SQRTHALF = np.sqrt(0.5)
(12)                    MAGIC45 = SQRTHALF * MAGIC
(12)                    vertices = np.array(
(16)                        [[0.0, -1.0],
(17)                         [MAGIC, -1.0],
(17)                         [SQRTHALF-MAGIC45, -SQRTHALF-MAGIC45],
(17)                         [SQRTHALF, -SQRTHALF],
(17)                         [SQRTHALF+MAGIC45, -SQRTHALF+MAGIC45],
(17)                         [1.0, -MAGIC],
(17)                         [1.0, 0.0],
(17)                         [1.0, MAGIC],
(17)                         [SQRTHALF+MAGIC45, SQRTHALF-MAGIC45],
(17)                         [SQRTHALF, SQRTHALF],
(17)                         [SQRTHALF-MAGIC45, SQRTHALF+MAGIC45],
(17)                         [MAGIC, 1.0],
(17)                         [0.0, 1.0],
(17)                         [0.0, -1.0]],
(16)                        float)
(12)                    codes = np.full(14, cls.CURVE4, dtype=cls.code_type)
(12)                    codes[0] = cls.MOVETO
(12)                    codes[-1] = cls.CLOSEPOLY
(12)                    cls._unit_circle_righthalf = cls(vertices, codes, readonly=True)
(8)                 return cls._unit_circle_righthalf
(4)             @classmethod
(4)             def arc(cls, theta1, theta2, n=None, is_wedge=False):
(8)                 """
(8)                 Return a `Path` for the unit circle arc from angles *theta1* to
(8)                 *theta2* (in degrees).
(8)                 *theta2* is unwrapped to produce the shortest arc within 360 degrees.
(8)                 That is, if *theta2* > *theta1* + 360, the arc will be from *theta1*
to
(8)                 *theta2* - 360 and not a full circle plus some extra overlap.
(8)                 If *n* is provided, it is the number of spline segments to make.
(8)                 If *n* is not provided, the number of spline segments is
(8)                 determined based on the delta between *theta1* and *theta2*.
(11)                   Masionobe, L.  2003.  `Drawing an elliptical arc using
(11)                   polylines, quadratic or cubic Bezier curves
(11)
<https://web.archive.org/web/20190318044212/http://www.spaceroots.org/documents/ellipse/index.html
>`_.
(8)                 """
(8)                 halfpi = np.pi * 0.5
(8)                 eta1 = theta1
(8)                 eta2 = theta2 - 360 * np.floor((theta2 - theta1) / 360)
(8)                 if theta2 != theta1 and eta2 <= eta1:
(12)                    eta2 += 360
(8)                 eta1, eta2 = np.deg2rad([eta1, eta2])
(8)                 if n is None:
(12)                    n = int(2 ** np.ceil((eta2 - eta1) / halfpi))
(8)                 if n < 1:
(12)                    raise ValueError("n must be >= 1 or None")
(8)                 deta = (eta2 - eta1) / n
(8)                 t = np.tan(0.5 * deta)
(8)                 alpha = np.sin(deta) * (np.sqrt(4.0 + 3.0 * t * t) - 1) / 3.0
(8)                 steps = np.linspace(eta1, eta2, n + 1, True)
(8)                 cos_eta = np.cos(steps)
(8)                 sin_eta = np.sin(steps)
(8)                 xA = cos_eta[:-1]
```

```
799: (8)                    yA = sin_eta[:-1]
800: (8)                    xA_dot = -yA
801: (8)                    yA_dot = xA
802: (8)                    xB = cos_eta[1:]
803: (8)                    yB = sin_eta[1:]
804: (8)                    xB_dot = -yB
805: (8)                    yB_dot = xB
806: (8)                    if is_wedge:
807: (12)                       length = n * 3 + 4
808: (12)                       vertices = np.zeros((length, 2), float)
809: (12)                       codes = np.full(length, cls.CURVE4, dtype=cls.code_type)
810: (12)                       vertices[1] = [xA[0], yA[0]]
811: (12)                       codes[0:2] = [cls.MOVETO, cls.LINETO]
812: (12)                       codes[-2:] = [cls.LINETO, cls.CLOSEPOLY]
813: (12)                       vertex_offset = 2
814: (12)                       end = length - 2
815: (8)                    else:
816: (12)                       length = n * 3 + 1
817: (12)                       vertices = np.empty((length, 2), float)
818: (12)                       codes = np.full(length, cls.CURVE4, dtype=cls.code_type)
819: (12)                       vertices[0] = [xA[0], yA[0]]
820: (12)                       codes[0] = cls.MOVETO
821: (12)                       vertex_offset = 1
822: (12)                       end = length
823: (8)                    vertices[vertex_offset:end:3, 0] = xA + alpha * xA_dot
824: (8)                    vertices[vertex_offset:end:3, 1] = yA + alpha * yA_dot
825: (8)                    vertices[vertex_offset+1:end:3, 0] = xB - alpha * xB_dot
826: (8)                    vertices[vertex_offset+1:end:3, 1] = yB - alpha * yB_dot
827: (8)                    vertices[vertex_offset+2:end:3, 0] = xB
828: (8)                    vertices[vertex_offset+2:end:3, 1] = yB
829: (8)                    return cls(vertices, codes, readonly=True)
830: (4)                @classmethod
831: (4)                def wedge(cls, theta1, theta2, n=None):
832: (8)                    """
833: (8)                    Return a `Path` for the unit circle wedge from angles *theta1* to
834: (8)                    *theta2* (in degrees).
835: (8)                    *theta2* is unwrapped to produce the shortest wedge within 360
degrees.
836: (8)                    That is, if *theta2* > *theta1* + 360, the wedge will be from *theta1*
837: (8)                    to *theta2* - 360 and not a full circle plus some extra overlap.
838: (8)                    If *n* is provided, it is the number of spline segments to make.
839: (8)                    If *n* is not provided, the number of spline segments is
840: (8)                    determined based on the delta between *theta1* and *theta2*.
841: (8)                    See `Path.arc` for the reference on the approximation used.
842: (8)                    """
843: (8)                    return cls.arc(theta1, theta2, n, True)
844: (4)                @staticmethod
845: (4)                @lru_cache(8)
846: (4)                def hatch(hatchpattern, density=6):
847: (8)                    """
848: (8)                    Given a hatch specifier, *hatchpattern*, generates a `Path` that
849: (8)                    can be used in a repeated hatching pattern.  *density* is the
850: (8)                    number of lines per unit square.
851: (8)                    """
852: (8)                    from matplotlib.hatch import get_path
853: (8)                    return (get_path(hatchpattern, density)
854: (16)                           if hatchpattern is not None else None)
855: (4)                def clip_to_bbox(self, bbox, inside=True):
856: (8)                    """
857: (8)                    Clip the path to the given bounding box.
858: (8)                    The path must be made up of one or more closed polygons.  This
859: (8)                    algorithm will not behave correctly for unclosed paths.
860: (8)                    If *inside* is `True`, clip to the inside of the box, otherwise
861: (8)                    to the outside of the box.
862: (8)                    """
863: (8)                    verts = _path.clip_path_to_rect(self, bbox, inside)
864: (8)                    paths = [Path(poly) for poly in verts]
865: (8)                    return self.make_compound_path(*paths)
866: (0)            def get_path_collection_extents(
```

```
867: (8)                    master_transform, paths, transforms, offsets, offset_transform):
868: (4)                r"""
869: (4)                Get bounding box of a `.PathCollection`\s internal objects.
870: (4)                That is, given a sequence of `Path`\s, `.Transform`\s objects, and
offsets, as found
871: (4)                in a `.PathCollection`, return the bounding box that encapsulates all of
them.
872: (4)                Parameters
873: (4)                ----------
874: (4)                master_transform : `~matplotlib.transforms.Transform`
875: (8)                    Global transformation applied to all paths.
876: (4)                paths : list of `Path`
877: (4)                transforms : list of `~matplotlib.transforms.Affine2DBase`
878: (8)                    If non-empty, this overrides *master_transform*.
879: (4)                offsets : (N, 2) array-like
880: (4)                offset_transform : `~matplotlib.transforms.Affine2DBase`
881: (8)                    Transform applied to the offsets before offsetting the path.
882: (4)                Notes
883: (4)                -----
884: (4)                The way that *paths*, *transforms* and *offsets* are combined follows the
same
885: (4)                method as for collections: each is iterated over independently, so if you
have 3
886: (4)                paths (A, B, C), 2 transforms (α, β) and 1 offset (O), their combinations
are as
887: (4)                follows:
888: (4)                - (A, α, O)
889: (4)                - (B, β, O)
890: (4)                - (C, α, O)
891: (4)                """
892: (4)                from .transforms import Bbox
893: (4)                if len(paths) == 0:
894: (8)                    raise ValueError("No paths provided")
895: (4)                if len(offsets) == 0:
896: (8)                    _api.warn_deprecated(
897: (12)                       "3.8", message="Calling get_path_collection_extents() with an"
898: (12)                       " empty offsets list is deprecated since %(since)s. Support will"
899: (12)                       " be removed %(removal)s.")
900: (4)                extents, minpos = _path.get_path_collection_extents(
901: (8)                    master_transform, paths, np.atleast_3d(transforms),
902: (8)                    offsets, offset_transform)
903: (4)                return Bbox.from_extents(*extents, minpos=minpos)
```

File 54 - pylab.py:

```
1: (0)              """
2: (0)              `pylab` is a historic interface and its use is strongly discouraged. The
equivalent
3: (0)              replacement is `matplotlib.pyplot`.  See :ref:`api_interfaces` for a full
overview
4: (0)              of Matplotlib interfaces.
5: (0)              `pylab` was designed to support a MATLAB-like way of working with all plotting
related
6: (0)              functions directly available in the global namespace. This was achieved
through a
7: (0)              wildcard import (``from pylab import *``).
8: (0)              .. warning::
9: (3)                 The use of `pylab` is discouraged for the following reasons:
10: (3)                ``from pylab import *`` imports all the functions from `matplotlib.pyplot`,
`numpy`,
11: (3)                `numpy.fft`, `numpy.linalg`, and `numpy.random`, and some additional
functions into
12: (3)                the global namespace.
13: (3)                Such a pattern is considered bad practice in modern python, as it clutters
the global
14: (3)                namespace. Even more severely, in the case of `pylab`, this will overwrite
some
```

```
15: (3)                 builtin functions (e.g. the builtin `sum` will be replaced by `numpy.sum`),
which
16: (3)                 can lead to unexpected behavior.
17: (0)             """
18: (0)             from matplotlib.cbook import flatten, silent_list
19: (0)             import matplotlib as mpl
20: (0)             from matplotlib.dates import (
21: (4)                 date2num, num2date, datestr2num, drange, DateFormatter, DateLocator,
22: (4)                 RRuleLocator, YearLocator, MonthLocator, WeekdayLocator, DayLocator,
23: (4)                 HourLocator, MinuteLocator, SecondLocator, rrule, MO, TU, WE, TH, FR,
24: (4)                 SA, SU, YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY, SECONDLY,
25: (4)                 relativedelta)
26: (0)             from matplotlib.mlab import (
27: (4)                 detrend, detrend_linear, detrend_mean, detrend_none, window_hanning,
28: (4)                 window_none)
29: (0)             from matplotlib import cbook, mlab, pyplot as plt
30: (0)             from matplotlib.pyplot import *
31: (0)             from numpy import *
32: (0)             from numpy.fft import *
33: (0)             from numpy.random import *
34: (0)             from numpy.linalg import *
35: (0)             import numpy as np
36: (0)             import numpy.ma as ma
37: (0)             import datetime
38: (0)             bytes = __import__("builtins").bytes
39: (0)             abs = __import__("builtins").abs
40: (0)             bool = __import__("builtins").bool
41: (0)             max = __import__("builtins").max
42: (0)             min = __import__("builtins").min
43: (0)             pow = __import__("builtins").pow
44: (0)             round = __import__("builtins").round

----------------------------------------

File 55 - pyplot.py:

1: (0)              """
2: (0)              `matplotlib.pyplot` is a state-based interface to matplotlib. It provides
3: (0)              an implicit,  MATLAB-like, way of plotting.  It also opens figures on your
4: (0)              screen, and acts as the figure GUI manager.
5: (0)              pyplot is mainly intended for interactive plots and simple cases of
6: (0)              programmatic plot generation::
7: (4)                  import numpy as np
8: (4)                  import matplotlib.pyplot as plt
9: (4)                  x = np.arange(0, 5, 0.1)
10: (4)                 y = np.sin(x)
11: (4)                 plt.plot(x, y)
12: (0)             The explicit object-oriented API is recommended for complex plots, though
13: (0)             pyplot is still usually used to create the figure and often the Axes in the
14: (0)             figure. See `.pyplot.figure`, `.pyplot.subplots`, and
15: (0)             `.pyplot.subplot_mosaic` to create figures, and
16: (0)             :doc:`Axes API </api/axes_api>` for the plotting methods on an Axes::
17: (4)                 import numpy as np
18: (4)                 import matplotlib.pyplot as plt
19: (4)                 x = np.arange(0, 5, 0.1)
20: (4)                 y = np.sin(x)
21: (4)                 fig, ax = plt.subplots()
22: (4)                 ax.plot(x, y)
23: (0)             See :ref:`api_interfaces` for an explanation of the tradeoffs between the
24: (0)             implicit and explicit interfaces.
25: (0)             """
26: (0)             from __future__ import annotations
27: (0)             from contextlib import AbstractContextManager, ExitStack
28: (0)             from enum import Enum
29: (0)             import functools
30: (0)             import importlib
31: (0)             import inspect
32: (0)             import logging
33: (0)             import sys
```

```
34: (0)             import threading
35: (0)             import time
36: (0)             from typing import TYPE_CHECKING, cast, overload
37: (0)             from cycler import cycler  # noqa: F401
38: (0)             import matplotlib
39: (0)             import matplotlib.colorbar
40: (0)             import matplotlib.image
41: (0)             from matplotlib import _api
42: (0)             from matplotlib import (  # noqa: F401 Re-exported for typing.
43: (4)                 cm as cm, get_backend as get_backend, rcParams as rcParams, style as
style)
44: (0)             from matplotlib import _pylab_helpers
45: (0)             from matplotlib import interactive  # noqa: F401
46: (0)             from matplotlib import cbook
47: (0)             from matplotlib import _docstring
48: (0)             from matplotlib.backend_bases import (
49: (4)                 FigureCanvasBase, FigureManagerBase, MouseButton)
50: (0)             from matplotlib.figure import Figure, FigureBase, figaspect
51: (0)             from matplotlib.gridspec import GridSpec, SubplotSpec
52: (0)             from matplotlib import rcsetup, rcParamsDefault, rcParamsOrig
53: (0)             from matplotlib.artist import Artist
54: (0)             from matplotlib.axes import Axes
55: (0)             from matplotlib.axes import Subplot  # noqa: F401
56: (0)             from matplotlib.backends import BackendFilter, backend_registry
57: (0)             from matplotlib.projections import PolarAxes
58: (0)             from matplotlib import mlab  # for detrend_none, window_hanning
59: (0)             from matplotlib.scale import get_scale_names  # noqa: F401
60: (0)             from matplotlib.cm import _colormaps
61: (0)             from matplotlib.colors import _color_sequences, Colormap
62: (0)             import numpy as np
63: (0)             if TYPE_CHECKING:
64: (4)                 from collections.abc import Callable, Hashable, Iterable, Sequence
65: (4)                 import datetime
66: (4)                 import pathlib
67: (4)                 import os
68: (4)                 from typing import Any, BinaryIO, Literal, TypeVar
69: (4)                 from typing_extensions import ParamSpec
70: (4)                 import PIL.Image
71: (4)                 from numpy.typing import ArrayLike
72: (4)                 import matplotlib.axes
73: (4)                 import matplotlib.artist
74: (4)                 import matplotlib.backend_bases
75: (4)                 from matplotlib.axis import Tick
76: (4)                 from matplotlib.axes._base import _AxesBase
77: (4)                 from matplotlib.backend_bases import RendererBase, Event
78: (4)                 from matplotlib.cm import ScalarMappable
79: (4)                 from matplotlib.contour import ContourSet, QuadContourSet
80: (4)                 from matplotlib.collections import (
81: (8)                     Collection,
82: (8)                     LineCollection,
83: (8)                     PolyCollection,
84: (8)                     PathCollection,
85: (8)                     EventCollection,
86: (8)                     QuadMesh,
87: (4)                 )
88: (4)                 from matplotlib.colorbar import Colorbar
89: (4)                 from matplotlib.container import (
90: (8)                     BarContainer,
91: (8)                     ErrorbarContainer,
92: (8)                     StemContainer,
93: (4)                 )
94: (4)                 from matplotlib.figure import SubFigure
95: (4)                 from matplotlib.legend import Legend
96: (4)                 from matplotlib.mlab import GaussianKDE
97: (4)                 from matplotlib.image import AxesImage, FigureImage
98: (4)                 from matplotlib.patches import FancyArrow, StepPatch, Wedge
99: (4)                 from matplotlib.quiver import Barbs, Quiver, QuiverKey
100: (4)                from matplotlib.scale import ScaleBase
101: (4)                from matplotlib.transforms import Transform, Bbox
```

```
102: (4)                from matplotlib.typing import ColorType, LineStyleType, MarkerType,
HashableList
103: (4)                from matplotlib.widgets import SubplotTool
104: (4)                _P = ParamSpec('_P')
105: (4)                _R = TypeVar('_R')
106: (4)                _T = TypeVar('_T')
107: (0)            from matplotlib.colors import Normalize
108: (0)            from matplotlib.lines import Line2D, AxLine
109: (0)            from matplotlib.text import Text, Annotation
110: (0)            from matplotlib.patches import Arrow, Circle, Rectangle  # noqa: F401
111: (0)            from matplotlib.patches import Polygon
112: (0)            from matplotlib.widgets import Button, Slider, Widget  # noqa: F401
113: (0)            from .ticker import (  # noqa: F401
114: (4)                TickHelper, Formatter, FixedFormatter, NullFormatter, FuncFormatter,
115: (4)                FormatStrFormatter, ScalarFormatter, LogFormatter, LogFormatterExponent,
116: (4)                LogFormatterMathtext, Locator, IndexLocator, FixedLocator, NullLocator,
117: (4)                LinearLocator, LogLocator, AutoLocator, MultipleLocator, MaxNLocator)
118: (0)            _log = logging.getLogger(__name__)
119: (0)            colormaps = _colormaps
120: (0)            color_sequences = _color_sequences
121: (0)            @overload
122: (0)            def _copy_docstring_and_deprecators(
123: (4)                method: Any,
124: (4)                func: Literal[None] = None
125: (0)            ) -> Callable[[Callable[_P, _R]], Callable[_P, _R]]: ...
126: (0)            @overload
127: (0)            def _copy_docstring_and_deprecators(
128: (4)                method: Any, func: Callable[_P, _R]) -> Callable[_P, _R]: ...
129: (0)            def _copy_docstring_and_deprecators(
130: (4)                method: Any,
131: (4)                func: Callable[_P, _R] | None = None
132: (0)            ) -> Callable[[Callable[_P, _R]], Callable[_P, _R]] | Callable[_P, _R]:
133: (4)                if func is None:
134: (8)                    return cast('Callable[[Callable[_P, _R]], Callable[_P, _R]]',
135: (20)                               functools.partial(_copy_docstring_and_deprecators,
method))
136: (4)                decorators: list[Callable[[Callable[_P, _R]], Callable[_P, _R]]] = [
137: (8)                    _docstring.copy(method)
138: (4)                ]
139: (4)                while hasattr(method, "__wrapped__"):
140: (8)                    potential_decorator = _api.deprecation.DECORATORS.get(method)
141: (8)                    if potential_decorator:
142: (12)                       decorators.append(potential_decorator)
143: (8)                    method = method.__wrapped__
144: (4)                for decorator in decorators[::-1]:
145: (8)                    func = decorator(func)
146: (4)                _add_pyplot_note(func, method)
147: (4)                return func
148: (0)            _NO_PYPLOT_NOTE = [
149: (4)                'FigureBase._gci',  # wrapped_func is private
150: (4)                '_AxesBase._sci',  # wrapped_func is private
151: (4)                'Artist.findobj',  # not a standard pyplot wrapper because it does not
operate
152: (0)            ]
153: (0)            def _add_pyplot_note(func, wrapped_func):
154: (4)                """
155: (4)                Add a note to the docstring of *func* that it is a pyplot wrapper.
156: (4)                The note is added to the "Notes" section of the docstring. If that does
157: (4)                not exist, a "Notes" section is created. In numpydoc, the "Notes"
158: (4)                section is the third last possible section, only potentially followed by
159: (4)                "References" and "Examples".
160: (4)                """
161: (4)                if not func.__doc__:
162: (8)                    return  # nothing to do
163: (4)                qualname = wrapped_func.__qualname__
164: (4)                if qualname in _NO_PYPLOT_NOTE:
165: (8)                    return
166: (4)                wrapped_func_is_method = True
167: (4)                if "." not in qualname:
```

```
168: (8)                    wrapped_func_is_method = False
169: (8)                    link = f"{wrapped_func.__module__}.{qualname}"
170: (4)                elif qualname.startswith("Axes."):  # e.g. "Axes.plot"
171: (8)                    link = ".axes." + qualname
172: (4)                elif qualname.startswith("_AxesBase."):  # e.g. "_AxesBase.set_xlabel"
173: (8)                    link = ".axes.Axes" + qualname[9:]
174: (4)                elif qualname.startswith("Figure."):  # e.g. "Figure.figimage"
175: (8)                    link = "." + qualname
176: (4)                elif qualname.startswith("FigureBase."):  # e.g. "FigureBase.gca"
177: (8)                    link = ".Figure" + qualname[10:]
178: (4)                elif qualname.startswith("FigureCanvasBase."):  #
"FigureBaseCanvas.mpl_connect"
179: (8)                    link = "." + qualname
180: (4)                else:
181: (8)                    raise RuntimeError(f"Wrapped method from unexpected class:
{qualname}")
182: (4)                if wrapped_func_is_method:
183: (8)                    message = f"This is the :ref:`pyplot wrapper <pyplot_interface>` for
`{link}`."
184: (4)                else:
185: (8)                    message = f"This is equivalent to `{link}`."
186: (4)                doc = inspect.cleandoc(func.__doc__)
187: (4)                if "\nNotes\n-----" in doc:
188: (8)                    before, after = doc.split("\nNotes\n-----", 1)
189: (4)                elif (index := doc.find("\nReferences\n----------")) != -1:
190: (8)                    before, after = doc[:index], doc[index:]
191: (4)                elif (index := doc.find("\nExamples\n--------")) != -1:
192: (8)                    before, after = doc[:index], doc[index:]
193: (4)                else:
194: (8)                    before = doc + "\n"
195: (8)                    after = ""
196: (4)                func.__doc__ = f"{before}\nNotes\n-----\n\n.. note::\n\n
{message}\n{after}"
197: (0)            _ReplDisplayHook = Enum("_ReplDisplayHook", ["NONE", "PLAIN", "IPYTHON"])
198: (0)            _REPL_DISPLAYHOOK = _ReplDisplayHook.NONE
199: (0)            def _draw_all_if_interactive() -> None:
200: (4)                if matplotlib.is_interactive():
201: (8)                    draw_all()
202: (0)            def install_repl_displayhook() -> None:
203: (4)                """
204: (4)                Connect to the display hook of the current shell.
205: (4)                The display hook gets called when the read-evaluate-print-loop (REPL) of
206: (4)                the shell has finished the execution of a command. We use this callback
207: (4)                to be able to automatically update a figure in interactive mode.
208: (4)                This works both with IPython and with vanilla python shells.
209: (4)                """
210: (4)                global _REPL_DISPLAYHOOK
211: (4)                if _REPL_DISPLAYHOOK is _ReplDisplayHook.IPYTHON:
212: (8)                    return
213: (4)                mod_ipython = sys.modules.get("IPython")
214: (4)                if not mod_ipython:
215: (8)                    _REPL_DISPLAYHOOK = _ReplDisplayHook.PLAIN
216: (8)                    return
217: (4)                ip = mod_ipython.get_ipython()
218: (4)                if not ip:
219: (8)                    _REPL_DISPLAYHOOK = _ReplDisplayHook.PLAIN
220: (8)                    return
221: (4)                ip.events.register("post_execute", _draw_all_if_interactive)
222: (4)                _REPL_DISPLAYHOOK = _ReplDisplayHook.IPYTHON
223: (4)                if mod_ipython.version_info[:2] < (8, 24):
224: (8)                    from IPython.core.pylabtools import backend2gui
225: (8)                    ipython_gui_name = backend2gui.get(get_backend())
226: (4)                else:
227: (8)                    _, ipython_gui_name = backend_registry.resolve_backend(get_backend())
228: (4)                if ipython_gui_name:
229: (8)                    ip.enable_gui(ipython_gui_name)
230: (0)            def uninstall_repl_displayhook() -> None:
231: (4)                """Disconnect from the display hook of the current shell."""
232: (4)                global _REPL_DISPLAYHOOK
```

```
(4)     if _REPL_DISPLAYHOOK is _ReplDisplayHook.IPYTHON:
(8)         from IPython import get_ipython
(8)         ip = get_ipython()
(8)         ip.events.unregister("post_execute", _draw_all_if_interactive)
(4)     _REPL_DISPLAYHOOK = _ReplDisplayHook.NONE
(0)draw_all = _pylab_helpers.Gcf.draw_all
(0)@_copy_docstring_and_deprecators(matplotlib.set_loglevel)
(0)def set_loglevel(*args, **kwargs) -> None:
(4)     return matplotlib.set_loglevel(*args, **kwargs)
(0)@_copy_docstring_and_deprecators(Artist.findobj)
(0)def findobj(
(4)     o: Artist | None = None,
(4)     match: Callable[[Artist], bool] | type[Artist] | None = None,
(4)     include_self: bool = True
(0)) -> list[Artist]:
(4)     if o is None:
(8)         o = gcf()
(4)     return o.findobj(match, include_self=include_self)
(0)_backend_mod: type[matplotlib.backend_bases._Backend] | None = None
(0)def _get_backend_mod() -> type[matplotlib.backend_bases._Backend]:
(4)     """
(4)     Ensure that a backend is selected and return it.
(4)     This is currently private, but may be made public in the future.
(4)     """
(4)     if _backend_mod is None:
(8)         switch_backend(rcParams._get("backend"))
(4)     return cast(type[matplotlib.backend_bases._Backend], _backend_mod)
(0)def switch_backend(newbackend: str) -> None:
(4)     """
(4)     Set the pyplot backend.
(4)     Switching to an interactive backend is possible only if no event loop for
(4)     another interactive backend has started.  Switching to and from
(4)     non-interactive backends is always possible.
(4)     If the new backend is different than the current backend then all open
(4)     Figures will be closed via ``plt.close('all')``.
(4)     Parameters
(4)     ----------
(4)     newbackend : str
(8)         The case-insensitive name of the backend to use.
(4)     """
(4)     global _backend_mod
(4)     import matplotlib.backends
(4)     if newbackend is rcsetup._auto_backend_sentinel:
(8)         current_framework = cbook._get_running_interactive_framework()
(8)         if (current_framework and
(16)                (backend := backend_registry.backend_for_gui_framework(
(20)                    current_framework))):
(12)            candidates = [backend]
(8)         else:
(12)            candidates = []
(8)         candidates += [
(12)            "macosx", "qtagg", "gtk4agg", "gtk3agg", "tkagg", "wxagg"]
(8)         for candidate in candidates:
(12)            try:
(16)                switch_backend(candidate)
(12)            except ImportError:
(16)                continue
(12)            else:
(16)                rcParamsOrig['backend'] = candidate
(16)                return
(8)         else:
(12)            switch_backend("agg")
(12)            rcParamsOrig["backend"] = "agg"
(12)            return
(4)     old_backend = dict.__getitem__(rcParams, 'backend')
(4)     module = backend_registry.load_backend_module(newbackend)
(4)     canvas_class = module.FigureCanvas
(4)     required_framework = canvas_class.required_interactive_framework
(4)     if required_framework is not None:
```

```
302: (8)                    current_framework = cbook._get_running_interactive_framework()
303: (8)                    if (current_framework and required_framework
304: (16)                           and current_framework != required_framework):
305: (12)                       raise ImportError(
306: (16)                           "Cannot load backend {!r} which requires the {!r} interactive
"
307: (16)                           "framework, as {!r} is currently running".format(
308: (20)                               newbackend, required_framework, current_framework))
309: (4)                new_figure_manager = getattr(module, "new_figure_manager", None)
310: (4)                show = getattr(module, "show", None)
311: (4)                class backend_mod(matplotlib.backend_bases._Backend):
312: (8)                    locals().update(vars(module))
313: (4)                if new_figure_manager is None:
314: (8)                    def new_figure_manager_given_figure(num, figure):
315: (12)                       return canvas_class.new_manager(figure, num)
316: (8)                    def new_figure_manager(num, *args, FigureClass=Figure, **kwargs):
317: (12)                       fig = FigureClass(*args, **kwargs)
318: (12)                       return new_figure_manager_given_figure(num, fig)
319: (8)                    def draw_if_interactive() -> None:
320: (12)                       if matplotlib.is_interactive():
321: (16)                           manager = _pylab_helpers.Gcf.get_active()
322: (16)                           if manager:
323: (20)                               manager.canvas.draw_idle()
324: (8)                    backend_mod.new_figure_manager_given_figure = (  # type:
ignore[method-assign]
325: (12)                       new_figure_manager_given_figure)
326: (8)                    backend_mod.new_figure_manager = (  # type: ignore[method-assign]
327: (12)                       new_figure_manager)
328: (8)                    backend_mod.draw_if_interactive = (  # type: ignore[method-assign]
329: (12)                       draw_if_interactive)
330: (4)                manager_class = getattr(canvas_class, "manager_class", None)
331: (4)                manager_pyplot_show = inspect.getattr_static(manager_class, "pyplot_show",
None)
332: (4)                base_pyplot_show = inspect.getattr_static(FigureManagerBase,
"pyplot_show", None)
333: (4)                if (show is None
334: (12)                       or (manager_pyplot_show is not None
335: (16)                           and manager_pyplot_show != base_pyplot_show)):
336: (8)                    if not manager_pyplot_show:
337: (12)                       raise ValueError(
338: (16)                           f"Backend {newbackend} defines neither
FigureCanvas.manager_class nor "
339: (16)                           f"a toplevel show function")
340: (8)                    _pyplot_show = cast('Any', manager_class).pyplot_show
341: (8)                    backend_mod.show = _pyplot_show  # type: ignore[method-assign]
342: (4)                _log.debug("Loaded backend %s version %s.",
343: (15)                          newbackend, backend_mod.backend_version)
344: (4)                if newbackend in ("ipympl", "widget"):
345: (8)                    import importlib.metadata as im
346: (8)                    from matplotlib import _parse_to_version_info  # type: ignore[attr-
defined]
347: (8)                    try:
348: (12)                       module_version = im.version("ipympl")
349: (12)                       if _parse_to_version_info(module_version) < (0, 9, 4):
350: (16)                           newbackend = "module://ipympl.backend_nbagg"
351: (8)                    except im.PackageNotFoundError:
352: (12)                       pass
353: (4)                rcParams['backend'] = rcParamsDefault['backend'] = newbackend
354: (4)                _backend_mod = backend_mod
355: (4)                for func_name in ["new_figure_manager", "draw_if_interactive", "show"]:
356: (8)                    globals()[func_name].__signature__ = inspect.signature(
357: (12)                       getattr(backend_mod, func_name))
358: (4)                matplotlib.backends.backend = newbackend  # type: ignore[attr-defined]
359: (4)                if not cbook._str_equal(old_backend, newbackend):
360: (8)                    if get_fignums():
361: (12)                       _api.warn_deprecated("3.8", message=(
362: (16)                           "Auto-close()ing of figures upon backend switching is
deprecated since "
363: (16)                           "%(since)s and will be removed %(removal)s.  To suppress this
```

```
warning, "
(16)                            "explicitly call plt.close('all') first."))
(8)                     close("all")
(4)                 install_repl_displayhook()
(0)             def _warn_if_gui_out_of_main_thread() -> None:
(4)                 warn = False
(4)                 canvas_class = cast(type[FigureCanvasBase],
_get_backend_mod().FigureCanvas)
(4)                 if canvas_class.required_interactive_framework:
(8)                     if hasattr(threading, 'get_native_id'):
(12)                        if threading.get_native_id() != threading.main_thread().native_id:
(16)                            warn = True
(8)                     else:
(12)                        if threading.current_thread() is not threading.main_thread():
(16)                            warn = True
(4)                 if warn:
(8)                     _api.warn_external(
(12)                        "Starting a Matplotlib GUI outside of the main thread will likely
"
(12)                        "fail.")
(0)             def new_figure_manager(*args, **kwargs):
(4)                 """Create a new figure manager instance."""
(4)                 _warn_if_gui_out_of_main_thread()
(4)                 return _get_backend_mod().new_figure_manager(*args, **kwargs)
(0)             def draw_if_interactive(*args, **kwargs):
(4)                 """
(4)                 Redraw the current figure if in interactive mode.
(4)                 .. warning::
(8)                     End users will typically not have to call this function because the
(8)                     the interactive mode takes care of this.
(4)                 """
(4)                 return _get_backend_mod().draw_if_interactive(*args, **kwargs)
(0)             def show(*args, **kwargs) -> None:
(4)                 """
(4)                 Display all open figures.
(4)                 Parameters
(4)                 ----------
(4)                 block : bool, optional
(8)                     Whether to wait for all figures to be closed before returning.
(8)                     If `True` block and run the GUI main loop until all figure windows
(8)                     are closed.
(8)                     If `False` ensure that all figure windows are displayed and return
(8)                     immediately.  In this case, you are responsible for ensuring
(8)                     that the event loop is running to have responsive figures.
(8)                     Defaults to True in non-interactive mode and to False in interactive
(8)                     mode (see `.pyplot.isinteractive`).
(4)                 See Also
(4)                 --------
(4)                 ion : Enable interactive mode, which shows / updates the figure after
(10)                      every plotting command, so that calling ``show()`` is not necessary.
(4)                 ioff : Disable interactive mode.
(4)                 savefig : Save the figure to an image file instead of showing it on
screen.
(4)                 Notes
(4)                 -----
(4)                 **Saving figures to file and showing a window at the same time**
(4)                 If you want an image file as well as a user interface window, use
(4)                 `.pyplot.savefig` before `.pyplot.show`. At the end of (a blocking)
(4)                 ``show()`` the figure is closed and thus unregistered from pyplot. Calling
(4)                 `.pyplot.savefig` afterwards would save a new and thus empty figure. This
(4)                 limitation of command order does not apply if the show is non-blocking or
(4)                 if you keep a reference to the figure and use `.Figure.savefig`.
(4)                 **Auto-show in jupyter notebooks**
(4)                 The jupyter backends (activated via ``%matplotlib inline``,
(4)                 ``%matplotlib notebook``, or ``%matplotlib widget``), call ``show()`` at
(4)                 the end of every cell by default. Thus, you usually don't have to call it
(4)                 explicitly there.
(4)                 """
(4)                 _warn_if_gui_out_of_main_thread()
```

```
(4)                 return _get_backend_mod().show(*args, **kwargs)
(0)             def isinteractive() -> bool:
(4)                 """
(4)                 Return whether plots are updated after every plotting command.
(4)                 The interactive mode is mainly useful if you build plots from the command
(4)                 line and want to see the effect of each command while you are building the
(4)                 figure.
(4)                 In interactive mode:
(4)                 - newly created figures will be shown immediately;
(4)                 - figures will automatically redraw on change;
(4)                 - `.pyplot.show` will not block by default.
(4)                 In non-interactive mode:
(4)                 - newly created figures and changes to figures will not be reflected until
(6)                   explicitly asked to be;
(4)                 - `.pyplot.show` will block by default.
(4)                 See Also
(4)                 --------
(4)                 ion : Enable interactive mode.
(4)                 ioff : Disable interactive mode.
(4)                 show : Show all figures (and maybe block).
(4)                 pause : Show all figures, and block for a time.
(4)                 """
(4)                 return matplotlib.is_interactive()
(0)             def ioff() -> AbstractContextManager:
(4)                 """
(4)                 Disable interactive mode.
(4)                 See `.pyplot.isinteractive` for more details.
(4)                 See Also
(4)                 --------
(4)                 ion : Enable interactive mode.
(4)                 isinteractive : Whether interactive mode is enabled.
(4)                 show : Show all figures (and maybe block).
(4)                 pause : Show all figures, and block for a time.
(4)                 Notes
(4)                 -----
(4)                 For a temporary change, this can be used as a context manager::
(8)                     plt.ion()
(8)                     fig = plt.figure()
(8)                     with plt.ioff():
(12)                        fig2 = plt.figure()
(4)                 To enable optional usage as a context manager, this function returns a
(4)                 context manager object, which is not intended to be stored or
(4)                 accessed by the user.
(4)                 """
(4)                 stack = ExitStack()
(4)                 stack.callback(ion if isinteractive() else ioff)
(4)                 matplotlib.interactive(False)
(4)                 uninstall_repl_displayhook()
(4)                 return stack
(0)             def ion() -> AbstractContextManager:
(4)                 """
(4)                 Enable interactive mode.
(4)                 See `.pyplot.isinteractive` for more details.
(4)                 See Also
(4)                 --------
(4)                 ioff : Disable interactive mode.
(4)                 isinteractive : Whether interactive mode is enabled.
(4)                 show : Show all figures (and maybe block).
(4)                 pause : Show all figures, and block for a time.
(4)                 Notes
(4)                 -----
(4)                 For a temporary change, this can be used as a context manager::
(8)                     plt.ioff()
(8)                     fig = plt.figure()
(8)                     with plt.ion():
(12)                        fig2 = plt.figure()
(4)                 To enable optional usage as a context manager, this function returns a
(4)                 context manager object, which is not intended to be stored or
(4)                 accessed by the user.
```

```
498: (4)                """
499: (4)                stack = ExitStack()
500: (4)                stack.callback(ion if isinteractive() else ioff)
501: (4)                matplotlib.interactive(True)
502: (4)                install_repl_displayhook()
503: (4)                return stack
504: (0)            def pause(interval: float) -> None:
505: (4)                """
506: (4)                Run the GUI event loop for *interval* seconds.
507: (4)                If there is an active figure, it will be updated and displayed before the
508: (4)                pause, and the GUI event loop (if any) will run during the pause.
509: (4)                This can be used for crude animation.  For more complex animation use
510: (4)                :mod:`matplotlib.animation`.
511: (4)                If there is no active figure, sleep for *interval* seconds instead.
512: (4)                See Also
513: (4)                --------
514: (4)                matplotlib.animation : Proper animations
515: (4)                show : Show all figures and optional block until all figures are closed.
516: (4)                """
517: (4)                manager = _pylab_helpers.Gcf.get_active()
518: (4)                if manager is not None:
519: (8)                    canvas = manager.canvas
520: (8)                    if canvas.figure.stale:
521: (12)                       canvas.draw_idle()
522: (8)                    show(block=False)
523: (8)                    canvas.start_event_loop(interval)
524: (4)                else:
525: (8)                    time.sleep(interval)
526: (0)            @_copy_docstring_and_deprecators(matplotlib.rc)
527: (0)            def rc(group: str, **kwargs) -> None:
528: (4)                matplotlib.rc(group, **kwargs)
529: (0)            @_copy_docstring_and_deprecators(matplotlib.rc_context)
530: (0)            def rc_context(
531: (4)                rc: dict[str, Any] | None = None,
532: (4)                fname: str | pathlib.Path | os.PathLike | None = None,
533: (0)            ) -> AbstractContextManager[None]:
534: (4)                return matplotlib.rc_context(rc, fname)
535: (0)            @_copy_docstring_and_deprecators(matplotlib.rcdefaults)
536: (0)            def rcdefaults() -> None:
537: (4)                matplotlib.rcdefaults()
538: (4)                if matplotlib.is_interactive():
539: (8)                    draw_all()
540: (0)            @_copy_docstring_and_deprecators(matplotlib.artist.getp)
541: (0)            def getp(obj, *args, **kwargs):
542: (4)                return matplotlib.artist.getp(obj, *args, **kwargs)
543: (0)            @_copy_docstring_and_deprecators(matplotlib.artist.get)
544: (0)            def get(obj, *args, **kwargs):
545: (4)                return matplotlib.artist.get(obj, *args, **kwargs)
546: (0)            @_copy_docstring_and_deprecators(matplotlib.artist.setp)
547: (0)            def setp(obj, *args, **kwargs):
548: (4)                return matplotlib.artist.setp(obj, *args, **kwargs)
549: (0)            def xkcd(
550: (4)                scale: float = 1, length: float = 100, randomness: float = 2
551: (0)            ) -> ExitStack:
552: (4)                """
553: (4)                Turn on `xkcd <https://xkcd.com/>`_ sketch-style drawing mode.
554: (4)                This will only have an effect on things drawn after this function is
called.
555: (4)                For best results, install the `xkcd script
<https://github.com/ipython/xkcd-font/>`_
556: (4)                font; xkcd fonts are not packaged with Matplotlib.
557: (4)                Parameters
558: (4)                ----------
559: (4)                scale : float, optional
560: (8)                    The amplitude of the wiggle perpendicular to the source line.
561: (4)                length : float, optional
562: (8)                    The length of the wiggle along the line.
563: (4)                randomness : float, optional
564: (8)                    The scale factor by which the length is shrunken or expanded.
```

```
565: (4)                Notes
566: (4)                -----
567: (4)                This function works by a number of rcParams, so it will probably
568: (4)                override others you have set before.
569: (4)                If you want the effects of this function to be temporary, it can
570: (4)                be used as a context manager, for example::
571: (8)                    with plt.xkcd():
572: (12)                       fig1 = plt.figure()
573: (8)                    fig2 = plt.figure()
574: (4)                """
575: (4)                if rcParams['text.usetex']:
576: (8)                    raise RuntimeError(
577: (12)                       "xkcd mode is not compatible with text.usetex = True")
578: (4)                stack = ExitStack()
579: (4)                stack.callback(dict.update, rcParams, rcParams.copy())  # type:
ignore[arg-type]
580: (4)                from matplotlib import patheffects
581: (4)                rcParams.update({
582: (8)                    'font.family': ['xkcd', 'xkcd Script', 'Comic Neue', 'Comic Sans MS'],
583: (8)                    'font.size': 14.0,
584: (8)                    'path.sketch': (scale, length, randomness),
585: (8)                    'path.effects': [
586: (12)                       patheffects.withStroke(linewidth=4, foreground="w")],
587: (8)                    'axes.linewidth': 1.5,
588: (8)                    'lines.linewidth': 2.0,
589: (8)                    'figure.facecolor': 'white',
590: (8)                    'grid.linewidth': 0.0,
591: (8)                    'axes.grid': False,
592: (8)                    'axes.unicode_minus': False,
593: (8)                    'axes.edgecolor': 'black',
594: (8)                    'xtick.major.size': 8,
595: (8)                    'xtick.major.width': 3,
596: (8)                    'ytick.major.size': 8,
597: (8)                    'ytick.major.width': 3,
598: (4)                })
599: (4)                return stack
600: (0)            def figure(
601: (4)                num: int | str | Figure | SubFigure | None = None,
602: (4)                figsize: tuple[float, float] | None = None,
603: (4)                dpi: float | None = None,
604: (4)                *,
605: (4)                facecolor: ColorType | None = None,
606: (4)                edgecolor: ColorType | None = None,
607: (4)                frameon: bool = True,
608: (4)                FigureClass: type[Figure] = Figure,
609: (4)                clear: bool = False,
610: (4)                **kwargs
611: (0)            ) -> Figure:
612: (4)                """
613: (4)                Create a new figure, or activate an existing figure.
614: (4)                Parameters
615: (4)                ----------
616: (4)                num : int or str or `.Figure` or `.SubFigure`, optional
617: (8)                    A unique identifier for the figure.
618: (8)                    If a figure with that identifier already exists, this figure is made
619: (8)                    active and returned. An integer refers to the ``Figure.number``
620: (8)                    attribute, a string refers to the figure label.
621: (8)                    If there is no figure with the identifier or *num* is not given, a new
622: (8)                    figure is created, made active and returned.  If *num* is an int, it
623: (8)                    will be used for the ``Figure.number`` attribute, otherwise, an
624: (8)                    auto-generated integer value is used (starting at 1 and incremented
625: (8)                    for each new figure). If *num* is a string, the figure label and the
626: (8)                    window title is set to this value.  If num is a ``SubFigure``, its
627: (8)                    parent ``Figure`` is activated.
628: (4)                figsize : (float, float), default: :rc:`figure.figsize`
629: (8)                    Width, height in inches.
630: (4)                dpi : float, default: :rc:`figure.dpi`
631: (8)                    The resolution of the figure in dots-per-inch.
632: (4)                facecolor : :mpltype:`color`, default: :rc:`figure.facecolor`
```

```
(8)                 The background color.
(4)             edgecolor : :mpltype:`color`, default: :rc:`figure.edgecolor`
(8)                 The border color.
(4)             frameon : bool, default: True
(8)                 If False, suppress drawing the figure frame.
(4)             FigureClass : subclass of `~matplotlib.figure.Figure`
(8)                 If set, an instance of this subclass will be created, rather than a
(8)                 plain `.Figure`.
(4)             clear : bool, default: False
(8)                 If True and the figure already exists, then it is cleared.
(4)             layout : {'constrained', 'compressed', 'tight', 'none', `.LayoutEngine`,
None}, \
(0)         default: None
(8)                 The layout mechanism for positioning of plot elements to avoid
(8)                 overlapping Axes decorations (labels, ticks, etc). Note that layout
(8)                 managers can measurably slow down figure display.
(8)                 - 'constrained': The constrained layout solver adjusts Axes sizes
(10)                  to avoid overlapping Axes decorations.  Can handle complex plot
(10)                  layouts and colorbars, and is thus recommended.
(10)                  See :ref:`constrainedlayout_guide`
(10)                  for examples.
(8)                 - 'compressed': uses the same algorithm as 'constrained', but
(10)                  removes extra space between fixed-aspect-ratio Axes.  Best for
(10)                  simple grids of Axes.
(8)                 - 'tight': Use the tight layout mechanism. This is a relatively
(10)                  simple algorithm that adjusts the subplot parameters so that
(10)                  decorations do not overlap. See `.Figure.set_tight_layout` for
(10)                  further details.
(8)                 - 'none': Do not use a layout engine.
(8)                 - A `.LayoutEngine` instance. Builtin layout classes are
(10)                  `.ConstrainedLayoutEngine` and `.TightLayoutEngine`, more easily
(10)                  accessible by 'constrained' and 'tight'.  Passing an instance
(10)                  allows third parties to provide their own layout engine.
(8)                 If not given, fall back to using the parameters *tight_layout* and
(8)                 *constrained_layout*, including their config defaults
(8)                 :rc:`figure.autolayout` and :rc:`figure.constrained_layout.use`.
(4)             **kwargs
(8)                 Additional keyword arguments are passed to the `.Figure` constructor.
(4)             Returns
(4)             -------
(4)             `~matplotlib.figure.Figure`
(4)             Notes
(4)             -----
(4)             A newly created figure is passed to the `~.FigureCanvasBase.new_manager`
(4)             method or the `new_figure_manager` function provided by the current
(4)             backend, which install a canvas and a manager on the figure.
(4)             Once this is done, :rc:`figure.hooks` are called, one at a time, on the
(4)             figure; these hooks allow arbitrary customization of the figure (e.g.,
(4)             attaching callbacks) or of associated elements (e.g., modifying the
(4)             toolbar).  See :doc:`/gallery/user_interfaces/mplcvd` for an example of
(4)             toolbar customization.
(4)             If you are creating many figures, make sure you explicitly call
(4)             `.pyplot.close` on the figures you are not using, because this will
(4)             enable pyplot to properly clean up the memory.
(4)             `~matplotlib.rcParams` defines the default values, which can be modified
(4)             in the matplotlibrc file.
(4)             """
(4)             if isinstance(num, FigureBase):
(8)                 if num.canvas.manager is None:
(12)                    raise ValueError("The passed figure is not managed by pyplot")
(8)                 _pylab_helpers.Gcf.set_active(num.canvas.manager)
(8)                 return num.figure
(4)             allnums = get_fignums()
(4)             next_num = max(allnums) + 1 if allnums else 1
(4)             fig_label = ''
(4)             if num is None:
(8)                 num = next_num
(4)             elif isinstance(num, str):
(8)                 fig_label = num
```

```
701: (8)                    all_labels = get_figlabels()
702: (8)                    if fig_label not in all_labels:
703: (12)                       if fig_label == 'all':
704: (16)                           _api.warn_external("close('all') closes all existing
figures.")
705: (12)                       num = next_num
706: (8)                    else:
707: (12)                       inum = all_labels.index(fig_label)
708: (12)                       num = allnums[inum]
709: (4)                else:
710: (8)                    num = int(num)  # crude validation of num argument
711: (4)                manager = _pylab_helpers.Gcf.get_fig_manager(num)  # type: ignore[arg-
type]
712: (4)                if manager is None:
713: (8)                    max_open_warning = rcParams['figure.max_open_warning']
714: (8)                    if len(allnums) == max_open_warning >= 1:
715: (12)                       _api.warn_external(
716: (16)                           f"More than {max_open_warning} figures have been opened. "
717: (16)                           f"Figures created through the pyplot interface "
718: (16)                           f"(`matplotlib.pyplot.figure`) are retained until explicitly "
719: (16)                           f"closed and may consume too much memory. (To control this "
720: (16)                           f"warning, see the rcParam `figure.max_open_warning`). "
721: (16)                           f"Consider using `matplotlib.pyplot.close()`.",
722: (16)                           RuntimeWarning)
723: (8)                    manager = new_figure_manager(
724: (12)                       num, figsize=figsize, dpi=dpi,
725: (12)                       facecolor=facecolor, edgecolor=edgecolor, frameon=frameon,
726: (12)                       FigureClass=FigureClass, **kwargs)
727: (8)                    fig = manager.canvas.figure
728: (8)                    if fig_label:
729: (12)                       fig.set_label(fig_label)
730: (8)                    for hookspecs in rcParams["figure.hooks"]:
731: (12)                       module_name, dotted_name = hookspecs.split(":")
732: (12)                       obj: Any = importlib.import_module(module_name)
733: (12)                       for part in dotted_name.split("."):
734: (16)                           obj = getattr(obj, part)
735: (12)                       obj(fig)
736: (8)                    _pylab_helpers.Gcf._set_new_active_manager(manager)
737: (8)                    draw_if_interactive()
738: (8)                    if _REPL_DISPLAYHOOK is _ReplDisplayHook.PLAIN:
739: (12)                       fig.stale_callback = _auto_draw_if_interactive
740: (4)                if clear:
741: (8)                    manager.canvas.figure.clear()
742: (4)                return manager.canvas.figure
743: (0)            def _auto_draw_if_interactive(fig, val):
744: (4)                """
745: (4)                An internal helper function for making sure that auto-redrawing
746: (4)                works as intended in the plain python repl.
747: (4)                Parameters
748: (4)                ----------
749: (4)                fig : Figure
750: (8)                    A figure object which is assumed to be associated with a canvas
751: (4)                """
752: (4)                if (val and matplotlib.is_interactive()
753: (12)                       and not fig.canvas.is_saving()
754: (12)                       and not fig.canvas._is_idle_drawing):
755: (8)                    with fig.canvas._idle_draw_cntx():
756: (12)                       fig.canvas.draw_idle()
757: (0)            def gcf() -> Figure:
758: (4)                """
759: (4)                Get the current figure.
760: (4)                If there is currently no figure on the pyplot figure stack, a new one is
761: (4)                created using `~.pyplot.figure()`.  (To test whether there is currently a
762: (4)                figure on the pyplot figure stack, check whether `~.pyplot.get_fignums()`
763: (4)                is empty.)
764: (4)                """
765: (4)                manager = _pylab_helpers.Gcf.get_active()
766: (4)                if manager is not None:
767: (8)                    return manager.canvas.figure
```

```
(4)         else:
(8)             return figure()
(0)     def fignum_exists(num: int | str) -> bool:
(4)         """
(4)         Return whether the figure with the given id exists.
(4)         Parameters
(4)         ----------
(4)         num : int or str
(8)             A figure identifier.
(4)         Returns
(4)         -------
(4)         bool
(8)             Whether or not a figure with id *num* exists.
(4)         """
(4)         return (
(8)             _pylab_helpers.Gcf.has_fignum(num)
(8)             if isinstance(num, int)
(8)             else num in get_figlabels()
(4)         )
(0)     def get_fignums() -> list[int]:
(4)         """Return a list of existing figure numbers."""
(4)         return sorted(_pylab_helpers.Gcf.figs)
(0)     def get_figlabels() -> list[Any]:
(4)         """Return a list of existing figure labels."""
(4)         managers = _pylab_helpers.Gcf.get_all_fig_managers()
(4)         managers.sort(key=lambda m: m.num)
(4)         return [m.canvas.figure.get_label() for m in managers]
(0)     def get_current_fig_manager() -> FigureManagerBase | None:
(4)         """
(4)         Return the figure manager of the current figure.
(4)         The figure manager is a container for the actual backend-depended window
(4)         that displays the figure on screen.
(4)         If no current figure exists, a new one is created, and its figure
(4)         manager is returned.
(4)         Returns
(4)         -------
(4)         `.FigureManagerBase` or backend-dependent subclass thereof
(4)         """
(4)         return gcf().canvas.manager
(0)     @_copy_docstring_and_deprecators(FigureCanvasBase.mpl_connect)
(0)     def connect(s: str, func: Callable[[Event], Any]) -> int:
(4)         return gcf().canvas.mpl_connect(s, func)
(0)     @_copy_docstring_and_deprecators(FigureCanvasBase.mpl_disconnect)
(0)     def disconnect(cid: int) -> None:
(4)         gcf().canvas.mpl_disconnect(cid)
(0)     def close(fig: None | int | str | Figure | Literal["all"] = None) -> None:
(4)         """
(4)         Close a figure window.
(4)         Parameters
(4)         ----------
(4)         fig : None or int or str or `.Figure`
(8)             The figure to close. There are a number of ways to specify this:
(8)             - *None*: the current figure
(8)             - `.Figure`: the given `.Figure` instance
(8)             - ``int``: a figure number
(8)             - ``str``: a figure name
(8)             - 'all': all figures
(4)         """
(4)         if fig is None:
(8)             manager = _pylab_helpers.Gcf.get_active()
(8)             if manager is None:
(12)                return
(8)             else:
(12)                _pylab_helpers.Gcf.destroy(manager)
(4)         elif fig == 'all':
(8)             _pylab_helpers.Gcf.destroy_all()
(4)         elif isinstance(fig, int):
(8)             _pylab_helpers.Gcf.destroy(fig)
(4)         elif hasattr(fig, 'int'):
```

```
(8)                     _pylab_helpers.Gcf.destroy(fig.int)
(4)                 elif isinstance(fig, str):
(8)                     all_labels = get_figlabels()
(8)                     if fig in all_labels:
(12)                        num = get_fignums()[all_labels.index(fig)]
(12)                        _pylab_helpers.Gcf.destroy(num)
(4)                 elif isinstance(fig, Figure):
(8)                     _pylab_helpers.Gcf.destroy_fig(fig)
(4)                 else:
(8)                     raise TypeError("close() argument must be a Figure, an int, a string,
"
(24)                                    "or None, not %s" % type(fig))
(0)             def clf() -> None:
(4)                 """Clear the current figure."""
(4)                 gcf().clear()
(0)             def draw() -> None:
(4)                 """
(4)                 Redraw the current figure.
(4)                 This is used to update a figure that has been altered, but not
(4)                 automatically re-drawn.  If interactive mode is on (via `.ion()`), this
(4)                 should be only rarely needed, but there may be ways to modify the state of
(4)                 a figure without marking it as "stale".  Please report these cases as
bugs.
(4)                 This is equivalent to calling ``fig.canvas.draw_idle()``, where ``fig`` is
(4)                 the current figure.
(4)                 See Also
(4)                 --------
(4)                 .FigureCanvasBase.draw_idle
(4)                 .FigureCanvasBase.draw
(4)                 """
(4)                 gcf().canvas.draw_idle()
(0)             @_copy_docstring_and_deprecators(Figure.savefig)
(0)             def savefig(*args, **kwargs) -> None:
(4)                 fig = gcf()
(4)                 res = fig.savefig(*args, **kwargs)  # type: ignore[func-returns-value]
(4)                 fig.canvas.draw_idle()  # Need this if 'transparent=True', to reset
colors.
(4)                 return res
(0)             def figlegend(*args, **kwargs) -> Legend:
(4)                 return gcf().legend(*args, **kwargs)
(0)             if Figure.legend.__doc__:
(4)                 figlegend.__doc__ = Figure.legend.__doc__ \
(8)                     .replace(" legend(", " figlegend(") \
(8)                     .replace("fig.legend(", "plt.figlegend(") \
(8)                     .replace("ax.plot(", "plt.plot(")
(0)             @_docstring.dedent_interpd
(0)             def axes(
(4)                 arg: None | tuple[float, float, float, float] = None,
(4)                 **kwargs
(0)             ) -> matplotlib.axes.Axes:
(4)                 """
(4)                 Add an Axes to the current figure and make it the current Axes.
(4)                 Call signatures::
(8)                     plt.axes()
(8)                     plt.axes(rect, projection=None, polar=False, **kwargs)
(8)                     plt.axes(ax)
(4)                 Parameters
(4)                 ----------
(4)                 arg : None or 4-tuple
(8)                     The exact behavior of this function depends on the type:
(8)                     - *None*: A new full window Axes is added using
(10)                      ``subplot(**kwargs)``.
(8)                     - 4-tuple of floats *rect* = ``(left, bottom, width, height)``.
(10)                      A new Axes is added with dimensions *rect* in normalized
(10)                      (0, 1) units using `~.Figure.add_axes` on the current figure.
(4)                 projection : {None, 'aitoff', 'hammer', 'lambert', 'mollweide', \
(0)             'polar', 'rectilinear', str}, optional
(8)                     The projection type of the `~.axes.Axes`. *str* is the name of
(8)                     a custom projection, see `~matplotlib.projections`. The default
```

```
(8)                 None results in a 'rectilinear' projection.
(4)             polar : bool, default: False
(8)                 If True, equivalent to projection='polar'.
(4)             sharex, sharey : `~matplotlib.axes.Axes`, optional
(8)                 Share the x or y `~matplotlib.axis` with sharex and/or sharey.
(8)                 The axis will have the same limits, ticks, and scale as the axis
(8)                 of the shared Axes.
(4)             label : str
(8)                 A label for the returned Axes.
(4)             Returns
(4)             -------
(4)             `~.axes.Axes`, or a subclass of `~.axes.Axes`
(8)                 The returned Axes class depends on the projection used. It is
(8)                 `~.axes.Axes` if rectilinear projection is used and
(8)                 `.projections.polar.PolarAxes` if polar projection is used.
(4)             Other Parameters
(4)             ----------------
(4)             **kwargs
(8)                 This method also takes the keyword arguments for
(8)                 the returned Axes class. The keyword arguments for the
(8)                 rectilinear Axes class `~.axes.Axes` can be found in
(8)                 the following table but there might also be other keyword
(8)                 arguments if another projection is used, see the actual Axes
(8)                 class.
(8)                 %(Axes:kwdoc)s
(4)             See Also
(4)             --------
(4)             .Figure.add_axes
(4)             .pyplot.subplot
(4)             .Figure.add_subplot
(4)             .Figure.subplots
(4)             .pyplot.subplots
(4)             Examples
(4)             --------
(4)             ::
(8)                 plt.axes()
(8)                 plt.axes((left, bottom, width, height), facecolor='grey')
(4)             """
(4)             fig = gcf()
(4)             pos = kwargs.pop('position', None)
(4)             if arg is None:
(8)                 if pos is None:
(12)                    return fig.add_subplot(**kwargs)
(8)                 else:
(12)                    return fig.add_axes(pos, **kwargs)
(4)             else:
(8)                 return fig.add_axes(arg, **kwargs)
(0)         def delaxes(ax: matplotlib.axes.Axes | None = None) -> None:
(4)             """
(4)             Remove an `~.axes.Axes` (defaulting to the current Axes) from its figure.
(4)             """
(4)             if ax is None:
(8)                 ax = gca()
(4)             ax.remove()
(0)         def sca(ax: Axes) -> None:
(4)             """
(4)             Set the current Axes to *ax* and the current Figure to the parent of *ax*.
(4)             """
(4)             figure(ax.figure)  # type: ignore[arg-type]
(4)             ax.figure.sca(ax)  # type: ignore[union-attr]
(0)         def cla() -> None:
(4)             """Clear the current Axes."""
(4)             return gca().cla()
(0)         @_docstring.dedent_interpd
(0)         def subplot(*args, **kwargs) -> Axes:
(4)             """
(4)             Add an Axes to the current figure or retrieve an existing Axes.
(4)             This is a wrapper of `.Figure.add_subplot` which provides additional
(4)             behavior when working with the implicit API (see the notes section).
```

```
(4)                 Call signatures::
(7)                    subplot(nrows, ncols, index, **kwargs)
(7)                    subplot(pos, **kwargs)
(7)                    subplot(**kwargs)
(7)                    subplot(ax)
(4)                 Parameters
(4)                 ----------
(4)                 *args : int, (int, int, *index*), or `.SubplotSpec`, default: (1, 1, 1)
(8)                     The position of the subplot described by one of
(8)                     - Three integers (*nrows*, *ncols*, *index*). The subplot will take
the
(10)                      *index* position on a grid with *nrows* rows and *ncols* columns.
(10)                      *index* starts at 1 in the upper left corner and increases to the
(10)                      right. *index* can also be a two-tuple specifying the (*first*,
(10)                      *last*) indices (1-based, and including *last*) of the subplot,
e.g.,
(10)                      ``fig.add_subplot(3, 1, (1, 2))`` makes a subplot that spans the
(10)                      upper 2/3 of the figure.
(8)                     - A 3-digit integer. The digits are interpreted as if given separately
(10)                      as three single-digit integers, i.e. ``fig.add_subplot(235)`` is the
(10)                      same as ``fig.add_subplot(2, 3, 5)``. Note that this can only be
used
(10)                      if there are no more than 9 subplots.
(8)                     - A `.SubplotSpec`.
(4)                 projection : {None, 'aitoff', 'hammer', 'lambert', 'mollweide', \
(0)             'polar', 'rectilinear', str}, optional
(8)                     The projection type of the subplot (`~.axes.Axes`). *str* is the name
(8)                     of a custom projection, see `~matplotlib.projections`. The default
(8)                     None results in a 'rectilinear' projection.
(4)                 polar : bool, default: False
(8)                     If True, equivalent to projection='polar'.
(4)                 sharex, sharey : `~matplotlib.axes.Axes`, optional
(8)                     Share the x or y `~matplotlib.axis` with sharex and/or sharey. The
(8)                     axis will have the same limits, ticks, and scale as the axis of the
(8)                     shared Axes.
(4)                 label : str
(8)                     A label for the returned Axes.
(4)                 Returns
(4)                 -------
(4)                 `~.axes.Axes`
(8)                     The Axes of the subplot. The returned Axes can actually be an instance
(8)                     of a subclass, such as `.projections.polar.PolarAxes` for polar
(8)                     projections.
(4)                 Other Parameters
(4)                 ----------------
(4)                 **kwargs
(8)                     This method also takes the keyword arguments for the returned Axes
(8)                     base class; except for the *figure* argument. The keyword arguments
(8)                     for the rectilinear base class `~.axes.Axes` can be found in
(8)                     the following table but there might also be other keyword
(8)                     arguments if another projection is used.
(8)                     %(Axes:kwdoc)s
(4)                 Notes
(4)                 -----
(4)                 Creating a new Axes will delete any preexisting Axes that
(4)                 overlaps with it beyond sharing a boundary::
(8)                     import matplotlib.pyplot as plt
(8)                     plt.plot([1, 2, 3])
(8)                     plt.subplot(211)
(4)                 If you do not want this behavior, use the `.Figure.add_subplot` method
(4)                 or the `.pyplot.axes` function instead.
(4)                 If no *kwargs* are passed and there exists an Axes in the location
(4)                 specified by *args* then that Axes will be returned rather than a new
(4)                 Axes being created.
(4)                 If *kwargs* are passed and there exists an Axes in the location
(4)                 specified by *args*, the projection type is the same, and the
(4)                 *kwargs* match with the existing Axes, then the existing Axes is
(4)                 returned.  Otherwise a new Axes is created with the specified
(4)                 parameters.  We save a reference to the *kwargs* which we use
```

```
(4)                 for this comparison.  If any of the values in *kwargs* are
(4)                 mutable we will not detect the case where they are mutated.
(4)                 In these cases we suggest using `.Figure.add_subplot` and the
(4)                 explicit Axes API rather than the implicit pyplot API.
(4)                 See Also
(4)                 --------
(4)                 .Figure.add_subplot
(4)                 .pyplot.subplots
(4)                 .pyplot.axes
(4)                 .Figure.subplots
(4)                 Examples
(4)                 --------
(4)                 ::
(8)                     plt.subplot(221)
(8)                     ax1 = plt.subplot(2, 2, 1)
(8)                     ax2 = plt.subplot(222, frameon=False)
(8)                     plt.subplot(223, projection='polar')
(8)                     plt.subplot(224, sharex=ax1, facecolor='red')
(8)                     plt.delaxes(ax2)
(8)                     plt.subplot(ax2)
(8)                     plt.subplot(221)
(4)                 """
(4)                 unset = object()
(4)                 projection = kwargs.get('projection', unset)
(4)                 polar = kwargs.pop('polar', unset)
(4)                 if polar is not unset and polar:
(8)                     if projection is not unset and projection != 'polar':
(12)                        raise ValueError(
(16)                            f"polar={polar}, yet projection={projection!r}. "
(16)                            "Only one of these arguments should be supplied."
(12)                        )
(8)                     kwargs['projection'] = projection = 'polar'
(4)                 if len(args) == 0:
(8)                     args = (1, 1, 1)
(4)                 if len(args) >= 3 and isinstance(args[2], bool):
(8)                     _api.warn_external("The subplot index argument to subplot() appears "
(27)                                       "to be a boolean. Did you intend to use "
(27)                                       "subplots()?")
(4)                 if 'nrows' in kwargs or 'ncols' in kwargs:
(8)                     raise TypeError("subplot() got an unexpected keyword argument 'ncols'
"
(24)                                    "and/or 'nrows'.  Did you intend to call subplots()?")
(4)                 fig = gcf()
(4)                 key = SubplotSpec._from_subplot_args(fig, args)
(4)                 for ax in fig.axes:
(8)                     if (ax.get_subplotspec() == key
(12)                        and (kwargs == {}
(17)                             or (ax._projection_init
(21)                                 == fig._process_projection_requirements(**kwargs)))):
(12)                        break
(4)                 else:
(8)                     ax = fig.add_subplot(*args, **kwargs)
(4)                 fig.sca(ax)
(4)                 return ax
(0)             @overload
(0)             def subplots(
(4)                 nrows: Literal[1] = ...,
(4)                 ncols: Literal[1] = ...,
(4)                 *,
(4)                 sharex: bool | Literal["none", "all", "row", "col"] = ...,
(4)                 sharey: bool | Literal["none", "all", "row", "col"] = ...,
(4)                 squeeze: Literal[True] = ...,
(4)                 width_ratios: Sequence[float] | None = ...,
(4)                 height_ratios: Sequence[float] | None = ...,
(4)                 subplot_kw: dict[str, Any] | None = ...,
(4)                 gridspec_kw: dict[str, Any] | None = ...,
(4)                 **fig_kw
(0)             ) -> tuple[Figure, Axes]:
(4)                 ...
```

```
(0)         @overload
(0)         def subplots(
(4)             nrows: int = ...,
(4)             ncols: int = ...,
(4)             *,
(4)             sharex: bool | Literal["none", "all", "row", "col"] = ...,
(4)             sharey: bool | Literal["none", "all", "row", "col"] = ...,
(4)             squeeze: Literal[False],
(4)             width_ratios: Sequence[float] | None = ...,
(4)             height_ratios: Sequence[float] | None = ...,
(4)             subplot_kw: dict[str, Any] | None = ...,
(4)             gridspec_kw: dict[str, Any] | None = ...,
(4)             **fig_kw
(0)         ) -> tuple[Figure, np.ndarray]:  # TODO numpy/numpy#24738
(4)             ...
(0)         @overload
(0)         def subplots(
(4)             nrows: int = ...,
(4)             ncols: int = ...,
(4)             *,
(4)             sharex: bool | Literal["none", "all", "row", "col"] = ...,
(4)             sharey: bool | Literal["none", "all", "row", "col"] = ...,
(4)             squeeze: bool = ...,
(4)             width_ratios: Sequence[float] | None = ...,
(4)             height_ratios: Sequence[float] | None = ...,
(4)             subplot_kw: dict[str, Any] | None = ...,
(4)             gridspec_kw: dict[str, Any] | None = ...,
(4)             **fig_kw
(0)         ) -> tuple[Figure, Any]:
(4)             ...
(0)         def subplots(
(4)             nrows: int = 1, ncols: int = 1, *,
(4)             sharex: bool | Literal["none", "all", "row", "col"] = False,
(4)             sharey: bool | Literal["none", "all", "row", "col"] = False,
(4)             squeeze: bool = True,
(4)             width_ratios: Sequence[float] | None = None,
(4)             height_ratios: Sequence[float] | None = None,
(4)             subplot_kw: dict[str, Any] | None = None,
(4)             gridspec_kw: dict[str, Any] | None = None,
(4)             **fig_kw
(0)         ) -> tuple[Figure, Any]:
(4)             """
(4)             Create a figure and a set of subplots.
(4)             This utility wrapper makes it convenient to create common layouts of
(4)             subplots, including the enclosing figure object, in a single call.
(4)             Parameters
(4)             ----------
(4)             nrows, ncols : int, default: 1
(8)                 Number of rows/columns of the subplot grid.
(4)             sharex, sharey : bool or {'none', 'all', 'row', 'col'}, default: False
(8)                 Controls sharing of properties among x (*sharex*) or y (*sharey*)
(8)                 axes:
(8)                 - True or 'all': x- or y-axis will be shared among all subplots.
(8)                 - False or 'none': each subplot x- or y-axis will be independent.
(8)                 - 'row': each subplot row will share an x- or y-axis.
(8)                 - 'col': each subplot column will share an x- or y-axis.
(8)                 When subplots have a shared x-axis along a column, only the x tick
(8)                 labels of the bottom subplot are created. Similarly, when subplots
(8)                 have a shared y-axis along a row, only the y tick labels of the first
(8)                 column subplot are created. To later turn other subplots' ticklabels
(8)                 on, use `~matplotlib.axes.Axes.tick_params`.
(8)                 When subplots have a shared axis that has units, calling
(8)                 `.Axis.set_units` will update each axis with the new units.
(8)                 Note that it is not possible to unshare axes.
(4)             squeeze : bool, default: True
(8)                 - If True, extra dimensions are squeezed out from the returned
(10)                  array of `~matplotlib.axes.Axes`:
(10)                  - if only one subplot is constructed (nrows=ncols=1), the
(12)                    resulting single Axes object is returned as a scalar.
```

```
(10)                    - for Nx1 or 1xM subplots, the returned object is a 1D numpy
(12)                      object array of Axes objects.
(10)                    - for NxM, subplots with N>1 and M>1 are returned as a 2D array.
(8)                   - If False, no squeezing at all is done: the returned Axes object is
(10)                    always a 2D array containing Axes instances, even if it ends up
(10)                    being 1x1.
(4)               width_ratios : array-like of length *ncols*, optional
(8)                   Defines the relative widths of the columns. Each column gets a
(8)                   relative width of ``width_ratios[i] / sum(width_ratios)``.
(8)                   If not given, all columns will have the same width.  Equivalent
(8)                   to ``gridspec_kw={'width_ratios': [...]}``.
(4)               height_ratios : array-like of length *nrows*, optional
(8)                   Defines the relative heights of the rows. Each row gets a
(8)                   relative height of ``height_ratios[i] / sum(height_ratios)``.
(8)                   If not given, all rows will have the same height. Convenience
(8)                   for ``gridspec_kw={'height_ratios': [...]}``.
(4)               subplot_kw : dict, optional
(8)                   Dict with keywords passed to the
(8)                   `~matplotlib.figure.Figure.add_subplot` call used to create each
(8)                   subplot.
(4)               gridspec_kw : dict, optional
(8)                   Dict with keywords passed to the `~matplotlib.gridspec.GridSpec`
(8)                   constructor used to create the grid the subplots are placed on.
(4)               **fig_kw
(8)                   All additional keyword arguments are passed to the
(8)                   `.pyplot.figure` call.
(4)               Returns
(4)               -------
(4)               fig : `.Figure`
(4)               ax : `~matplotlib.axes.Axes` or array of Axes
(8)                   *ax* can be either a single `~.axes.Axes` object, or an array of Axes
(8)                   objects if more than one subplot was created.  The dimensions of the
(8)                   resulting array can be controlled with the squeeze keyword, see above.
(8)                   Typical idioms for handling the return value are::
(12)                      fig, ax = plt.subplots()
(12)                      fig, axs = plt.subplots(2, 2)
(12)                      fig, (ax1, ax2) = plt.subplots(1, 2)
(12)                      fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
(8)                   The names ``ax`` and pluralized ``axs`` are preferred over ``axes``
(8)                   because for the latter it's not clear if it refers to a single
(8)                   `~.axes.Axes` instance or a collection of these.
(4)               See Also
(4)               --------
(4)               .pyplot.figure
(4)               .pyplot.subplot
(4)               .pyplot.axes
(4)               .Figure.subplots
(4)               .Figure.add_subplot
(4)               Examples
(4)               --------
(4)               ::
(8)                   x = np.linspace(0, 2*np.pi, 400)
(8)                   y = np.sin(x**2)
(8)                   fig, ax = plt.subplots()
(8)                   ax.plot(x, y)
(8)                   ax.set_title('Simple plot')
(8)                   f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
(8)                   ax1.plot(x, y)
(8)                   ax1.set_title('Sharing Y axis')
(8)                   ax2.scatter(x, y)
(8)                   fig, axs = plt.subplots(2, 2, subplot_kw=dict(projection="polar"))
(8)                   axs[0, 0].plot(x, y)
(8)                   axs[1, 1].scatter(x, y)
(8)                   plt.subplots(2, 2, sharex='col')
(8)                   plt.subplots(2, 2, sharey='row')
(8)                   plt.subplots(2, 2, sharex='all', sharey='all')
(8)                   plt.subplots(2, 2, sharex=True, sharey=True)
(8)                   fig, ax = plt.subplots(num=10, clear=True)
(4)               """
```

```
1244: (4)               fig = figure(**fig_kw)
1245: (4)               axs = fig.subplots(nrows=nrows, ncols=ncols, sharex=sharex, sharey=sharey,
1246: (23)                                 squeeze=squeeze, subplot_kw=subplot_kw,
1247: (23)                                 gridspec_kw=gridspec_kw, height_ratios=height_ratios,
1248: (23)                                 width_ratios=width_ratios)
1249: (4)               return fig, axs
1250: (0)           @overload
1251: (0)           def subplot_mosaic(
1252: (4)               mosaic: str,
1253: (4)               *,
1254: (4)               sharex: bool = ...,
1255: (4)               sharey: bool = ...,
1256: (4)               width_ratios: ArrayLike | None = ...,
1257: (4)               height_ratios: ArrayLike | None = ...,
1258: (4)               empty_sentinel: str = ...,
1259: (4)               subplot_kw: dict[str, Any] | None = ...,
1260: (4)               gridspec_kw: dict[str, Any] | None = ...,
1261: (4)               per_subplot_kw: dict[str | tuple[str, ...], dict[str, Any]] | None = ...,
1262: (4)               **fig_kw: Any
1263: (0)           ) -> tuple[Figure, dict[str, matplotlib.axes.Axes]]: ...
1264: (0)           @overload
1265: (0)           def subplot_mosaic(
1266: (4)               mosaic: list[HashableList[_T]],
1267: (4)               *,
1268: (4)               sharex: bool = ...,
1269: (4)               sharey: bool = ...,
1270: (4)               width_ratios: ArrayLike | None = ...,
1271: (4)               height_ratios: ArrayLike | None = ...,
1272: (4)               empty_sentinel: _T = ...,
1273: (4)               subplot_kw: dict[str, Any] | None = ...,
1274: (4)               gridspec_kw: dict[str, Any] | None = ...,
1275: (4)               per_subplot_kw: dict[_T | tuple[_T, ...], dict[str, Any]] | None = ...,
1276: (4)               **fig_kw: Any
1277: (0)           ) -> tuple[Figure, dict[_T, matplotlib.axes.Axes]]: ...
1278: (0)           @overload
1279: (0)           def subplot_mosaic(
1280: (4)               mosaic: list[HashableList[Hashable]],
1281: (4)               *,
1282: (4)               sharex: bool = ...,
1283: (4)               sharey: bool = ...,
1284: (4)               width_ratios: ArrayLike | None = ...,
1285: (4)               height_ratios: ArrayLike | None = ...,
1286: (4)               empty_sentinel: Any = ...,
1287: (4)               subplot_kw: dict[str, Any] | None = ...,
1288: (4)               gridspec_kw: dict[str, Any] | None = ...,
1289: (4)               per_subplot_kw: dict[Hashable | tuple[Hashable, ...], dict[str, Any]] |
None = ...,
1290: (4)               **fig_kw: Any
1291: (0)           ) -> tuple[Figure, dict[Hashable, matplotlib.axes.Axes]]: ...
1292: (0)           def subplot_mosaic(
1293: (4)               mosaic: str | list[HashableList[_T]] | list[HashableList[Hashable]],
1294: (4)               *,
1295: (4)               sharex: bool = False,
1296: (4)               sharey: bool = False,
1297: (4)               width_ratios: ArrayLike | None = None,
1298: (4)               height_ratios: ArrayLike | None = None,
1299: (4)               empty_sentinel: Any = '.',
1300: (4)               subplot_kw: dict[str, Any] | None = None,
1301: (4)               gridspec_kw: dict[str, Any] | None = None,
1302: (4)               per_subplot_kw: dict[str | tuple[str, ...], dict[str, Any]] |
1303: (20)                              dict[_T | tuple[_T, ...], dict[str, Any]] |
1304: (20)                              dict[Hashable | tuple[Hashable, ...], dict[str, Any]] |
None = None,
1305: (4)               **fig_kw: Any
1306: (0)           ) -> tuple[Figure, dict[str, matplotlib.axes.Axes]] | \
1307: (5)                tuple[Figure, dict[_T, matplotlib.axes.Axes]] | \
1308: (5)                tuple[Figure, dict[Hashable, matplotlib.axes.Axes]]:
1309: (4)               """
1310: (4)               Build a layout of Axes based on ASCII art or nested lists.
```

```
(4)                 This is a helper function to build complex GridSpec layouts visually.
(4)                 See :ref:`mosaic`
(4)                 for an example and full API documentation
(4)                 Parameters
(4)                 ----------
(4)                 mosaic : list of list of {hashable or nested} or str
(8)                     A visual layout of how you want your Axes to be arranged
(8)                     labeled as strings.  For example ::
(11)                       x = [['A panel', 'A panel', 'edge'],
(16)                            ['C panel', '.',       'edge']]
(8)                     produces 4 Axes:
(8)                     - 'A panel' which is 1 row high and spans the first two columns
(8)                     - 'edge' which is 2 rows high and is on the right edge
(8)                     - 'C panel' which in 1 row and 1 column wide in the bottom left
(8)                     - a blank space 1 row and 1 column wide in the bottom center
(8)                     Any of the entries in the layout can be a list of lists
(8)                     of the same form to create nested layouts.
(8)                     If input is a str, then it must be of the form ::
(10)                      '''
(10)                      AAE
(10)                      C.E
(10)                      '''
(8)                     where each character is a column and each line is a row.
(8)                     This only allows only single character Axes labels and does
(8)                     not allow nesting but is very terse.
(4)                 sharex, sharey : bool, default: False
(8)                     If True, the x-axis (*sharex*) or y-axis (*sharey*) will be shared
(8)                     among all subplots.  In that case, tick label visibility and axis
units
(8)                     behave as for `subplots`.  If False, each subplot's x- or y-axis will
(8)                     be independent.
(4)                 width_ratios : array-like of length *ncols*, optional
(8)                     Defines the relative widths of the columns. Each column gets a
(8)                     relative width of ``width_ratios[i] / sum(width_ratios)``.
(8)                     If not given, all columns will have the same width.  Convenience
(8)                     for ``gridspec_kw={'width_ratios': [...]}``.
(4)                 height_ratios : array-like of length *nrows*, optional
(8)                     Defines the relative heights of the rows. Each row gets a
(8)                     relative height of ``height_ratios[i] / sum(height_ratios)``.
(8)                     If not given, all rows will have the same height. Convenience
(8)                     for ``gridspec_kw={'height_ratios': [...]}``.
(4)                 empty_sentinel : object, optional
(8)                     Entry in the layout to mean "leave this space empty".  Defaults
(8)                     to ``'.'``. Note, if *layout* is a string, it is processed via
(8)                     `inspect.cleandoc` to remove leading white space, which may
(8)                     interfere with using white-space as the empty sentinel.
(4)                 subplot_kw : dict, optional
(8)                     Dictionary with keywords passed to the `.Figure.add_subplot` call
(8)                     used to create each subplot.  These values may be overridden by
(8)                     values in *per_subplot_kw*.
(4)                 per_subplot_kw : dict, optional
(8)                     A dictionary mapping the Axes identifiers or tuples of identifiers
(8)                     to a dictionary of keyword arguments to be passed to the
(8)                     `.Figure.add_subplot` call used to create each subplot.  The values
(8)                     in these dictionaries have precedence over the values in
(8)                     *subplot_kw*.
(8)                     If *mosaic* is a string, and thus all keys are single characters,
(8)                     it is possible to use a single string instead of a tuple as keys;
(8)                     i.e. ``"AB"`` is equivalent to ``("A", "B")``.
(8)                     .. versionadded:: 3.7
(4)                 gridspec_kw : dict, optional
(8)                     Dictionary with keywords passed to the `.GridSpec` constructor used
(8)                     to create the grid the subplots are placed on.
(4)                 **fig_kw
(8)                     All additional keyword arguments are passed to the
(8)                     `.pyplot.figure` call.
(4)                 Returns
(4)                 -------
(4)                 fig : `.Figure`
```

```
(7)                 The new figure
(4)             dict[label, Axes]
(7)                 A dictionary mapping the labels to the Axes objects.  The order of
(7)                 the Axes is left-to-right and top-to-bottom of their position in the
(7)                 total layout.
(4)             """
(4)             fig = figure(**fig_kw)
(4)             ax_dict = fig.subplot_mosaic(  # type: ignore[misc]
(8)                 mosaic,  # type: ignore[arg-type]
(8)                 sharex=sharex, sharey=sharey,
(8)                 height_ratios=height_ratios, width_ratios=width_ratios,
(8)                 subplot_kw=subplot_kw, gridspec_kw=gridspec_kw,
(8)                 empty_sentinel=empty_sentinel,
(8)                 per_subplot_kw=per_subplot_kw,  # type: ignore[arg-type]
(4)             )
(4)             return fig, ax_dict
(0)         def subplot2grid(
(4)             shape: tuple[int, int], loc: tuple[int, int],
(4)             rowspan: int = 1, colspan: int = 1,
(4)             fig: Figure | None = None,
(4)             **kwargs
(0)         ) -> matplotlib.axes.Axes:
(4)             """
(4)             Create a subplot at a specific location inside a regular grid.
(4)             Parameters
(4)             ----------
(4)             shape : (int, int)
(8)                 Number of rows and of columns of the grid in which to place axis.
(4)             loc : (int, int)
(8)                 Row number and column number of the axis location within the grid.
(4)             rowspan : int, default: 1
(8)                 Number of rows for the axis to span downwards.
(4)             colspan : int, default: 1
(8)                 Number of columns for the axis to span to the right.
(4)             fig : `.Figure`, optional
(8)                 Figure to place the subplot in. Defaults to the current figure.
(4)             **kwargs
(8)                 Additional keyword arguments are handed to `~.Figure.add_subplot`.
(4)             Returns
(4)             -------
(4)             `~.axes.Axes`
(8)                 The Axes of the subplot. The returned Axes can actually be an instance
(8)                 of a subclass, such as `.projections.polar.PolarAxes` for polar
(8)                 projections.
(4)             Notes
(4)             -----
(4)             The following call ::
(8)                 ax = subplot2grid((nrows, ncols), (row, col), rowspan, colspan)
(4)             is identical to ::
(8)                 fig = gcf()
(8)                 gs = fig.add_gridspec(nrows, ncols)
(8)                 ax = fig.add_subplot(gs[row:row+rowspan, col:col+colspan])
(4)             """
(4)             if fig is None:
(8)                 fig = gcf()
(4)             rows, cols = shape
(4)             gs = GridSpec._check_gridspec_exists(fig, rows, cols)
(4)             subplotspec = gs.new_subplotspec(loc, rowspan=rowspan, colspan=colspan)
(4)             return fig.add_subplot(subplotspec, **kwargs)
(0)         def twinx(ax: matplotlib.axes.Axes | None = None) -> _AxesBase:
(4)             """
(4)             Make and return a second Axes that shares the *x*-axis.  The new Axes will
(4)             overlay *ax* (or the current Axes if *ax* is *None*), and its ticks will
be
(4)             on the right.
(4)             Examples
(4)             --------
(4)             :doc:`/gallery/subplots_axes_and_figures/two_scales`
(4)             """
```

```
1447: (4)               if ax is None:
1448: (8)                   ax = gca()
1449: (4)               ax1 = ax.twinx()
1450: (4)               return ax1
1451: (0)           def twiny(ax: matplotlib.axes.Axes | None = None) -> _AxesBase:
1452: (4)               """
1453: (4)               Make and return a second Axes that shares the *y*-axis.  The new Axes will
1454: (4)               overlay *ax* (or the current Axes if *ax* is *None*), and its ticks will
be
1455: (4)               on the top.
1456: (4)               Examples
1457: (4)               --------
1458: (4)               :doc:`/gallery/subplots_axes_and_figures/two_scales`
1459: (4)               """
1460: (4)               if ax is None:
1461: (8)                   ax = gca()
1462: (4)               ax1 = ax.twiny()
1463: (4)               return ax1
1464: (0)           def subplot_tool(targetfig: Figure | None = None) -> SubplotTool | None:
1465: (4)               """
1466: (4)               Launch a subplot tool window for a figure.
1467: (4)               Returns
1468: (4)               -------
1469: (4)               `matplotlib.widgets.SubplotTool`
1470: (4)               """
1471: (4)               if targetfig is None:
1472: (8)                   targetfig = gcf()
1473: (4)               tb = targetfig.canvas.manager.toolbar  # type: ignore[union-attr]
1474: (4)               if hasattr(tb, "configure_subplots"):  # toolbar2
1475: (8)                   from matplotlib.backend_bases import NavigationToolbar2
1476: (8)                   return cast(NavigationToolbar2, tb).configure_subplots()
1477: (4)               elif hasattr(tb, "trigger_tool"):  # toolmanager
1478: (8)                   from matplotlib.backend_bases import ToolContainerBase
1479: (8)                   cast(ToolContainerBase, tb).trigger_tool("subplots")
1480: (8)                   return None
1481: (4)               else:
1482: (8)                   raise ValueError("subplot_tool can only be launched for figures with "
1483: (25)                                   "an associated toolbar")
1484: (0)           def box(on: bool | None = None) -> None:
1485: (4)               """
1486: (4)               Turn the Axes box on or off on the current Axes.
1487: (4)               Parameters
1488: (4)               ----------
1489: (4)               on : bool or None
1490: (8)                   The new `~matplotlib.axes.Axes` box state. If ``None``, toggle
1491: (8)                   the state.
1492: (4)               See Also
1493: (4)               --------
1494: (4)               :meth:`matplotlib.axes.Axes.set_frame_on`
1495: (4)               :meth:`matplotlib.axes.Axes.get_frame_on`
1496: (4)               """
1497: (4)               ax = gca()
1498: (4)               if on is None:
1499: (8)                   on = not ax.get_frame_on()
1500: (4)               ax.set_frame_on(on)
1501: (0)           def xlim(*args, **kwargs) -> tuple[float, float]:
1502: (4)               """
1503: (4)               Get or set the x limits of the current Axes.
1504: (4)               Call signatures::
1505: (8)                   left, right = xlim()  # return the current xlim
1506: (8)                   xlim((left, right))   # set the xlim to left, right
1507: (8)                   xlim(left, right)     # set the xlim to left, right
1508: (4)               If you do not specify args, you can pass *left* or *right* as kwargs,
1509: (4)               i.e.::
1510: (8)                   xlim(right=3)  # adjust the right leaving left unchanged
1511: (8)                   xlim(left=1)  # adjust the left leaving right unchanged
1512: (4)               Setting limits turns autoscaling off for the x-axis.
1513: (4)               Returns
1514: (4)               -------
```

```
(4)             left, right
(8)                 A tuple of the new x-axis limits.
(4)             Notes
(4)             -----
(4)             Calling this function with no arguments (e.g. ``xlim()``) is the pyplot
(4)             equivalent of calling `~.Axes.get_xlim` on the current Axes.
(4)             Calling this function with arguments is the pyplot equivalent of calling
(4)             `~.Axes.set_xlim` on the current Axes. All arguments are passed though.
(4)             """
(4)             ax = gca()
(4)             if not args and not kwargs:
(8)                 return ax.get_xlim()
(4)             ret = ax.set_xlim(*args, **kwargs)
(4)             return ret
(0)         def ylim(*args, **kwargs) -> tuple[float, float]:
(4)             """
(4)             Get or set the y-limits of the current Axes.
(4)             Call signatures::
(8)                 bottom, top = ylim()  # return the current ylim
(8)                 ylim((bottom, top))   # set the ylim to bottom, top
(8)                 ylim(bottom, top)     # set the ylim to bottom, top
(4)             If you do not specify args, you can alternatively pass *bottom* or
(4)             *top* as kwargs, i.e.::
(8)                 ylim(top=3)  # adjust the top leaving bottom unchanged
(8)                 ylim(bottom=1)  # adjust the bottom leaving top unchanged
(4)             Setting limits turns autoscaling off for the y-axis.
(4)             Returns
(4)             -------
(4)             bottom, top
(8)                 A tuple of the new y-axis limits.
(4)             Notes
(4)             -----
(4)             Calling this function with no arguments (e.g. ``ylim()``) is the pyplot
(4)             equivalent of calling `~.Axes.get_ylim` on the current Axes.
(4)             Calling this function with arguments is the pyplot equivalent of calling
(4)             `~.Axes.set_ylim` on the current Axes. All arguments are passed though.
(4)             """
(4)             ax = gca()
(4)             if not args and not kwargs:
(8)                 return ax.get_ylim()
(4)             ret = ax.set_ylim(*args, **kwargs)
(4)             return ret
(0)         def xticks(
(4)             ticks: ArrayLike | None = None,
(4)             labels: Sequence[str] | None = None,
(4)             *,
(4)             minor: bool = False,
(4)             **kwargs
(0)         ) -> tuple[list[Tick] | np.ndarray, list[Text]]:
(4)             """
(4)             Get or set the current tick locations and labels of the x-axis.
(4)             Pass no arguments to return the current values without modifying them.
(4)             Parameters
(4)             ----------
(4)             ticks : array-like, optional
(8)                 The list of xtick locations.  Passing an empty list removes all
xticks.
(4)             labels : array-like, optional
(8)                 The labels to place at the given *ticks* locations.  This argument can
(8)                 only be passed if *ticks* is passed as well.
(4)             minor : bool, default: False
(8)                 If ``False``, get/set the major ticks/labels; if ``True``, the minor
(8)                 ticks/labels.
(4)             **kwargs
(8)                 `.Text` properties can be used to control the appearance of the
labels.
(8)                 .. warning::
(12)                    This only sets the properties of the current ticks, which is
(12)                    only sufficient if you either pass *ticks*, resulting in a
```

```
1582: (12)                      fixed list of ticks, or if the plot is static.
1583: (12)                      Ticks are not guaranteed to be persistent. Various operations
1584: (12)                      can create, delete and modify the Tick instances. There is an
1585: (12)                      imminent risk that these settings can get lost if you work on
1586: (12)                      the figure further (including also panning/zooming on a
1587: (12)                      displayed figure).
1588: (12)                      Use `~.pyplot.tick_params` instead if possible.
1589: (4)               Returns
1590: (4)               -------
1591: (4)               locs
1592: (8)                   The list of xtick locations.
1593: (4)               labels
1594: (8)                   The list of xlabel `.Text` objects.
1595: (4)               Notes
1596: (4)               -----
1597: (4)               Calling this function with no arguments (e.g. ``xticks()``) is the pyplot
1598: (4)               equivalent of calling `~.Axes.get_xticks` and `~.Axes.get_xticklabels` on
1599: (4)               the current Axes.
1600: (4)               Calling this function with arguments is the pyplot equivalent of calling
1601: (4)               `~.Axes.set_xticks` and `~.Axes.set_xticklabels` on the current Axes.
1602: (4)               Examples
1603: (4)               --------
1604: (4)               >>> locs, labels = xticks()  # Get the current locations and labels.
1605: (4)               >>> xticks(np.arange(0, 1, step=0.2))  # Set label locations.
1606: (4)               >>> xticks(np.arange(3), ['Tom', 'Dick', 'Sue'])  # Set text labels.
1607: (4)               >>> xticks([0, 1, 2], ['January', 'February', 'March'],
1608: (4)               ...        rotation=20)  # Set text labels and properties.
1609: (4)               >>> xticks([])  # Disable xticks.
1610: (4)               """
1611: (4)               ax = gca()
1612: (4)               locs: list[Tick] | np.ndarray
1613: (4)               if ticks is None:
1614: (8)                   locs = ax.get_xticks(minor=minor)
1615: (8)                   if labels is not None:
1616: (12)                      raise TypeError("xticks(): Parameter 'labels' can't be set "
1617: (28)                                      "without setting 'ticks'")
1618: (4)               else:
1619: (8)                   locs = ax.set_xticks(ticks, minor=minor)
1620: (4)               labels_out: list[Text] = []
1621: (4)               if labels is None:
1622: (8)                   labels_out = ax.get_xticklabels(minor=minor)
1623: (8)                   for l in labels_out:
1624: (12)                      l._internal_update(kwargs)
1625: (4)               else:
1626: (8)                   labels_out = ax.set_xticklabels(labels, minor=minor, **kwargs)
1627: (4)               return locs, labels_out
1628: (0)           def yticks(
1629: (4)               ticks: ArrayLike | None = None,
1630: (4)               labels: Sequence[str] | None = None,
1631: (4)               *,
1632: (4)               minor: bool = False,
1633: (4)               **kwargs
1634: (0)           ) -> tuple[list[Tick] | np.ndarray, list[Text]]:
1635: (4)               """
1636: (4)               Get or set the current tick locations and labels of the y-axis.
1637: (4)               Pass no arguments to return the current values without modifying them.
1638: (4)               Parameters
1639: (4)               ----------
1640: (4)               ticks : array-like, optional
1641: (8)                   The list of ytick locations.  Passing an empty list removes all
yticks.
1642: (4)               labels : array-like, optional
1643: (8)                   The labels to place at the given *ticks* locations.  This argument can
1644: (8)                   only be passed if *ticks* is passed as well.
1645: (4)               minor : bool, default: False
1646: (8)                   If ``False``, get/set the major ticks/labels; if ``True``, the minor
1647: (8)                   ticks/labels.
1648: (4)               **kwargs
1649: (8)                   `.Text` properties can be used to control the appearance of the
```

```
labels.
(8)                 .. warning::
(12)                    This only sets the properties of the current ticks, which is
(12)                    only sufficient if you either pass *ticks*, resulting in a
(12)                    fixed list of ticks, or if the plot is static.
(12)                    Ticks are not guaranteed to be persistent. Various operations
(12)                    can create, delete and modify the Tick instances. There is an
(12)                    imminent risk that these settings can get lost if you work on
(12)                    the figure further (including also panning/zooming on a
(12)                    displayed figure).
(12)                    Use `~.pyplot.tick_params` instead if possible.
(4)             Returns
(4)             -------
(4)             locs
(8)                 The list of ytick locations.
(4)             labels
(8)                 The list of ylabel `.Text` objects.
(4)             Notes
(4)             -----
(4)             Calling this function with no arguments (e.g. ``yticks()``) is the pyplot
(4)             equivalent of calling `~.Axes.get_yticks` and `~.Axes.get_yticklabels` on
(4)             the current Axes.
(4)             Calling this function with arguments is the pyplot equivalent of calling
(4)             `~.Axes.set_yticks` and `~.Axes.set_yticklabels` on the current Axes.
(4)             Examples
(4)             --------
(4)             >>> locs, labels = yticks()  # Get the current locations and labels.
(4)             >>> yticks(np.arange(0, 1, step=0.2))  # Set label locations.
(4)             >>> yticks(np.arange(3), ['Tom', 'Dick', 'Sue'])  # Set text labels.
(4)             >>> yticks([0, 1, 2], ['January', 'February', 'March'],
(4)             ...        rotation=45)  # Set text labels and properties.
(4)             >>> yticks([])  # Disable yticks.
(4)             """
(4)             ax = gca()
(4)             locs: list[Tick] | np.ndarray
(4)             if ticks is None:
(8)                 locs = ax.get_yticks(minor=minor)
(8)                 if labels is not None:
(12)                    raise TypeError("yticks(): Parameter 'labels' can't be set "
(28)                                    "without setting 'ticks'")
(4)             else:
(8)                 locs = ax.set_yticks(ticks, minor=minor)
(4)             labels_out: list[Text] = []
(4)             if labels is None:
(8)                 labels_out = ax.get_yticklabels(minor=minor)
(8)                 for l in labels_out:
(12)                    l._internal_update(kwargs)
(4)             else:
(8)                 labels_out = ax.set_yticklabels(labels, minor=minor, **kwargs)
(4)             return locs, labels_out
(0)         def rgrids(
(4)             radii: ArrayLike | None = None,
(4)             labels: Sequence[str | Text] | None = None,
(4)             angle: float | None = None,
(4)             fmt: str | None = None,
(4)             **kwargs
(0)         ) -> tuple[list[Line2D], list[Text]]:
(4)             """
(4)             Get or set the radial gridlines on the current polar plot.
(4)             Call signatures::
(5)              lines, labels = rgrids()
(5)              lines, labels = rgrids(radii, labels=None, angle=22.5, fmt=None,
**kwargs)
(4)             When called with no arguments, `.rgrids` simply returns the tuple
(4)             (*lines*, *labels*). When called with arguments, the labels will
(4)             appear at the specified radial distances and angle.
(4)             Parameters
(4)             ----------
(4)             radii : tuple with floats
```

```
 1717: (8)                  The radii for the radial gridlines
 1718: (4)              labels : tuple with strings or None
 1719: (8)                  The labels to use at each radial gridline. The
 1720: (8)                  `matplotlib.ticker.ScalarFormatter` will be used if None.
 1721: (4)              angle : float
 1722: (8)                  The angular position of the radius labels in degrees.
 1723: (4)              fmt : str or None
 1724: (8)                  Format string used in `matplotlib.ticker.FormatStrFormatter`.
 1725: (8)                  For example '%f'.
 1726: (4)              Returns
 1727: (4)              -------
 1728: (4)              lines : list of `.lines.Line2D`
 1729: (8)                  The radial gridlines.
 1730: (4)              labels : list of `.text.Text`
 1731: (8)                  The tick labels.
 1732: (4)              Other Parameters
 1733: (4)              ----------------
 1734: (4)              **kwargs
 1735: (8)                  *kwargs* are optional `.Text` properties for the labels.
 1736: (4)              See Also
 1737: (4)              --------
 1738: (4)              .pyplot.thetagrids
 1739: (4)              .projections.polar.PolarAxes.set_rgrids
 1740: (4)              .Axis.get_gridlines
 1741: (4)              .Axis.get_ticklabels
 1742: (4)              Examples
 1743: (4)              --------
 1744: (4)              ::
 1745: (6)                lines, labels = rgrids( (0.25, 0.5, 1.0) )
 1746: (6)                lines, labels = rgrids( (0.25, 0.5, 1.0), ('Tom', 'Dick', 'Harry' ))
 1747: (4)              """
 1748: (4)              ax = gca()
 1749: (4)              if not isinstance(ax, PolarAxes):
 1750: (8)                  raise RuntimeError('rgrids only defined for polar Axes')
 1751: (4)              if all(p is None for p in [radii, labels, angle, fmt]) and not kwargs:
 1752: (8)                  lines_out: list[Line2D] = ax.yaxis.get_gridlines()
 1753: (8)                  labels_out: list[Text] = ax.yaxis.get_ticklabels()
 1754: (4)              elif radii is None:
 1755: (8)                  raise TypeError("'radii' cannot be None when other parameters are
passed")
 1756: (4)              else:
 1757: (8)                  lines_out, labels_out = ax.set_rgrids(
 1758: (12)                     radii, labels=labels, angle=angle, fmt=fmt, **kwargs)
 1759: (4)              return lines_out, labels_out
 1760: (0)          def thetagrids(
 1761: (4)              angles: ArrayLike | None = None,
 1762: (4)              labels: Sequence[str | Text] | None = None,
 1763: (4)              fmt: str | None = None,
 1764: (4)              **kwargs
 1765: (0)          ) -> tuple[list[Line2D], list[Text]]:
 1766: (4)              """
 1767: (4)              Get or set the theta gridlines on the current polar plot.
 1768: (4)              Call signatures::
 1769: (5)               lines, labels = thetagrids()
 1770: (5)               lines, labels = thetagrids(angles, labels=None, fmt=None, **kwargs)
 1771: (4)              When called with no arguments, `.thetagrids` simply returns the tuple
 1772: (4)              (*lines*, *labels*). When called with arguments, the labels will
 1773: (4)              appear at the specified angles.
 1774: (4)              Parameters
 1775: (4)              ----------
 1776: (4)              angles : tuple with floats, degrees
 1777: (8)                  The angles of the theta gridlines.
 1778: (4)              labels : tuple with strings or None
 1779: (8)                  The labels to use at each radial gridline. The
 1780: (8)                  `.projections.polar.ThetaFormatter` will be used if None.
 1781: (4)              fmt : str or None
 1782: (8)                  Format string used in `matplotlib.ticker.FormatStrFormatter`.
 1783: (8)                  For example '%f'. Note that the angle in radians will be used.
 1784: (4)              Returns
```

```
(4)             -------
(4)             lines : list of `.lines.Line2D`
(8)                 The theta gridlines.
(4)             labels : list of `.text.Text`
(8)                 The tick labels.
(4)             Other Parameters
(4)             ----------------
(4)             **kwargs
(8)                 *kwargs* are optional `.Text` properties for the labels.
(4)             See Also
(4)             --------
(4)             .pyplot.rgrids
(4)             .projections.polar.PolarAxes.set_thetagrids
(4)             .Axis.get_gridlines
(4)             .Axis.get_ticklabels
(4)             Examples
(4)             --------
(4)             ::
(6)               lines, labels = thetagrids(range(45, 360, 90))
(6)               lines, labels = thetagrids(range(45, 360, 90), ('NE', 'NW', 'SW', 'SE'))
(4)             """
(4)             ax = gca()
(4)             if not isinstance(ax, PolarAxes):
(8)                 raise RuntimeError('thetagrids only defined for polar Axes')
(4)             if all(param is None for param in [angles, labels, fmt]) and not kwargs:
(8)                 lines_out: list[Line2D] = ax.xaxis.get_ticklines()
(8)                 labels_out: list[Text] = ax.xaxis.get_ticklabels()
(4)             elif angles is None:
(8)                 raise TypeError("'angles' cannot be None when other parameters are
passed")
(4)             else:
(8)                 lines_out, labels_out = ax.set_thetagrids(angles,
(50)                                                          labels=labels, fmt=fmt,
(50)                                                          **kwargs)
(4)             return lines_out, labels_out
(0)         @_api.deprecated("3.7", pending=True)
(0)         def get_plot_commands() -> list[str]:
(4)             """
(4)             Get a sorted list of all of the plotting commands.
(4)             """
(4)             NON_PLOT_COMMANDS = {
(8)                 'connect', 'disconnect', 'get_current_fig_manager', 'ginput',
(8)                 'new_figure_manager', 'waitforbuttonpress'}
(4)             return [name for name in _get_pyplot_commands()
(12)                    if name not in NON_PLOT_COMMANDS]
(0)         def _get_pyplot_commands() -> list[str]:
(4)             exclude = {'colormaps', 'colors', 'get_plot_commands', *colormaps}
(4)             this_module = inspect.getmodule(get_plot_commands)
(4)             return sorted(
(8)                 name for name, obj in globals().items()
(8)                 if not name.startswith('_') and name not in exclude
(11)                   and inspect.isfunction(obj)
(11)                   and inspect.getmodule(obj) is this_module)
(0)         @_copy_docstring_and_deprecators(Figure.colorbar)
(0)         def colorbar(
(4)             mappable: ScalarMappable | None = None,
(4)             cax: matplotlib.axes.Axes | None = None,
(4)             ax: matplotlib.axes.Axes | Iterable[matplotlib.axes.Axes] | None = None,
(4)             **kwargs
(0)         ) -> Colorbar:
(4)             if mappable is None:
(8)                 mappable = gci()
(8)                 if mappable is None:
(12)                    raise RuntimeError('No mappable was found to use for colorbar '
(31)                                       'creation. First define a mappable such as '
(31)                                       'an image (with imshow) or a contour set ('
(31)                                       'with contourf).')
(4)             ret = gcf().colorbar(mappable, cax=cax, ax=ax, **kwargs)
(4)             return ret
```

```
(0)  def clim(vmin: float | None = None, vmax: float | None = None) -> None:
(4)      """
(4)      Set the color limits of the current image.
(4)      If either *vmin* or *vmax* is None, the image min/max respectively
(4)      will be used for color scaling.
(4)      If you want to set the clim of multiple images, use
(4)      `~.ScalarMappable.set_clim` on every image, for example::
(6)        for im in gca().get_images():
(10)           im.set_clim(0, 0.5)
(4)      """
(4)      im = gci()
(4)      if im is None:
(8)          raise RuntimeError('You must first define an image, e.g., with
imshow')
(4)      im.set_clim(vmin, vmax)
(0)  def get_cmap(name: Colormap | str | None = None, lut: int | None = None) ->
Colormap:
(4)      """
(4)      Get a colormap instance, defaulting to rc values if *name* is None.
(4)      Parameters
(4)      ----------
(4)      name : `~matplotlib.colors.Colormap` or str or None, default: None
(8)          If a `.Colormap` instance, it will be returned. Otherwise, the name of
(8)          a colormap known to Matplotlib, which will be resampled by *lut*. The
(8)          default, None, means :rc:`image.cmap`.
(4)      lut : int or None, default: None
(8)          If *name* is not already a Colormap instance and *lut* is not None,
the
(8)          colormap will be resampled to have *lut* entries in the lookup table.
(4)      Returns
(4)      -------
(4)      Colormap
(4)      """
(4)      if name is None:
(8)          name = rcParams['image.cmap']
(4)      if isinstance(name, Colormap):
(8)          return name
(4)      _api.check_in_list(sorted(_colormaps), name=name)
(4)      if lut is None:
(8)          return _colormaps[name]
(4)      else:
(8)          return _colormaps[name].resampled(lut)
(0)  def set_cmap(cmap: Colormap | str) -> None:
(4)      """
(4)      Set the default colormap, and applies it to the current image if any.
(4)      Parameters
(4)      ----------
(4)      cmap : `~matplotlib.colors.Colormap` or str
(8)          A colormap instance or the name of a registered colormap.
(4)      See Also
(4)      --------
(4)      colormaps
(4)      get_cmap
(4)      """
(4)      cmap = get_cmap(cmap)
(4)      rc('image', cmap=cmap.name)
(4)      im = gci()
(4)      if im is not None:
(8)          im.set_cmap(cmap)
(0)  @_copy_docstring_and_deprecators(matplotlib.image.imread)
(0)  def imread(
(8)          fname: str | pathlib.Path | BinaryIO, format: str | None = None
(0)  ) -> np.ndarray:
(4)      return matplotlib.image.imread(fname, format)
(0)  @_copy_docstring_and_deprecators(matplotlib.image.imsave)
(0)  def imsave(
(4)      fname: str | os.PathLike | BinaryIO, arr: ArrayLike, **kwargs
(0)  ) -> None:
(4)      matplotlib.image.imsave(fname, arr, **kwargs)
```

```
(0)           def matshow(A: ArrayLike, fignum: None | int = None, **kwargs) -> AxesImage:
(4)               """
(4)               Display a 2D array as a matrix in a new figure window.
(4)               The origin is set at the upper left hand corner.
(4)               The indexing is ``(row, column)`` so that the first index runs vertically
(4)               and the second index runs horizontally in the figure:
(4)               .. code-block:: none
(8)                   A[0, 0]   ⋯ A[0, M-1]
(11)                     ⋮             ⋮
(8)                   A[N-1, 0] ⋯ A[N-1, M-1]
(4)               The aspect ratio of the figure window is that of the array,
(4)               unless this would make an excessively short or narrow figure.
(4)               Tick labels for the xaxis are placed on top.
(4)               Parameters
(4)               ----------
(4)               A : 2D array-like
(8)                   The matrix to be displayed.
(4)               fignum : None or int
(8)                   If *None*, create a new, appropriately sized figure window.
(8)                   If 0, use the current Axes (creating one if there is none, without
ever
(8)                   adjusting the figure size).
(8)                   Otherwise, create a new Axes on the figure with the given number
(8)                   (creating it at the appropriate size if it does not exist, but not
(8)                   adjusting the figure size otherwise).  Note that this will be drawn on
(8)                   top of any preexisting Axes on the figure.
(4)               Returns
(4)               -------
(4)               `~matplotlib.image.AxesImage`
(4)               Other Parameters
(4)               ----------------
(4)               **kwargs : `~matplotlib.axes.Axes.imshow` arguments
(4)               """
(4)               A = np.asanyarray(A)
(4)               if fignum == 0:
(8)                   ax = gca()
(4)               else:
(8)                   fig = figure(fignum, figsize=figaspect(A))
(8)                   ax = fig.add_axes((0.15, 0.09, 0.775, 0.775))
(4)               im = ax.matshow(A, **kwargs)
(4)               sci(im)
(4)               return im
(0)           def polar(*args, **kwargs) -> list[Line2D]:
(4)               """
(4)               Make a polar plot.
(4)               call signature::
(6)                 polar(theta, r, **kwargs)
(4)               Multiple *theta*, *r* arguments are supported, with format strings, as in
(4)               `plot`.
(4)               """
(4)               if gcf().get_axes():
(8)                   ax = gca()
(8)                   if not isinstance(ax, PolarAxes):
(12)                      _api.warn_external('Trying to create polar plot on an Axes '
(31)                                         'that does not have a polar projection.')
(4)               else:
(8)                   ax = axes(projection="polar")
(4)               return ax.plot(*args, **kwargs)
(0)           if (rcParams["backend_fallback"]
(8)                   and rcParams._get_backend_or_none() in (  # type: ignore[attr-defined]
(12)                      set(backend_registry.list_builtin(BackendFilter.INTERACTIVE)) -
(12)                      {'webagg', 'nbagg'})
(8)                   and cbook._get_running_interactive_framework()):
(4)               rcParams._set("backend", rcsetup._auto_backend_sentinel)
(0)           @_copy_docstring_and_deprecators(Figure.figimage)
(0)           def figimage(
(4)               X: ArrayLike,
(4)               xo: int = 0,
(4)               yo: int = 0,
```

```
1987: (4)               alpha: float | None = None,
1988: (4)               norm: str | Normalize | None = None,
1989: (4)               cmap: str | Colormap | None = None,
1990: (4)               vmin: float | None = None,
1991: (4)               vmax: float | None = None,
1992: (4)               origin: Literal["upper", "lower"] | None = None,
1993: (4)               resize: bool = False,
1994: (4)               **kwargs,
1995: (0)           ) -> FigureImage:
1996: (4)               return gcf().figimage(
1997: (8)                   X,
1998: (8)                   xo=xo,
1999: (8)                   yo=yo,
2000: (8)                   alpha=alpha,
2001: (8)                   norm=norm,
2002: (8)                   cmap=cmap,
2003: (8)                   vmin=vmin,
2004: (8)                   vmax=vmax,
2005: (8)                   origin=origin,
2006: (8)                   resize=resize,
2007: (8)                   **kwargs,
2008: (4)               )
2009: (0)           @_copy_docstring_and_deprecators(Figure.text)
2010: (0)           def figtext(
2011: (4)               x: float, y: float, s: str, fontdict: dict[str, Any] | None = None,
**kwargs
2012: (0)           ) -> Text:
2013: (4)               return gcf().text(x, y, s, fontdict=fontdict, **kwargs)
2014: (0)           @_copy_docstring_and_deprecators(Figure.gca)
2015: (0)           def gca() -> Axes:
2016: (4)               return gcf().gca()
2017: (0)           @_copy_docstring_and_deprecators(Figure._gci)
2018: (0)           def gci() -> ScalarMappable | None:
2019: (4)               return gcf()._gci()
2020: (0)           @_copy_docstring_and_deprecators(Figure.ginput)
2021: (0)           def ginput(
2022: (4)               n: int = 1,
2023: (4)               timeout: float = 30,
2024: (4)               show_clicks: bool = True,
2025: (4)               mouse_add: MouseButton = MouseButton.LEFT,
2026: (4)               mouse_pop: MouseButton = MouseButton.RIGHT,
2027: (4)               mouse_stop: MouseButton = MouseButton.MIDDLE,
2028: (0)           ) -> list[tuple[int, int]]:
2029: (4)               return gcf().ginput(
2030: (8)                   n=n,
2031: (8)                   timeout=timeout,
2032: (8)                   show_clicks=show_clicks,
2033: (8)                   mouse_add=mouse_add,
2034: (8)                   mouse_pop=mouse_pop,
2035: (8)                   mouse_stop=mouse_stop,
2036: (4)               )
2037: (0)           @_copy_docstring_and_deprecators(Figure.subplots_adjust)
2038: (0)           def subplots_adjust(
2039: (4)               left: float | None = None,
2040: (4)               bottom: float | None = None,
2041: (4)               right: float | None = None,
2042: (4)               top: float | None = None,
2043: (4)               wspace: float | None = None,
2044: (4)               hspace: float | None = None,
2045: (0)           ) -> None:
2046: (4)               gcf().subplots_adjust(
2047: (8)                   left=left, bottom=bottom, right=right, top=top, wspace=wspace,
hspace=hspace
2048: (4)               )
2049: (0)           @_copy_docstring_and_deprecators(Figure.suptitle)
2050: (0)           def suptitle(t: str, **kwargs) -> Text:
2051: (4)               return gcf().suptitle(t, **kwargs)
2052: (0)           @_copy_docstring_and_deprecators(Figure.tight_layout)
2053: (0)           def tight_layout(
```

```
    *,
    pad: float = 1.08,
    h_pad: float | None = None,
    w_pad: float | None = None,
    rect: tuple[float, float, float, float] | None = None,
) -> None:
    gcf().tight_layout(pad=pad, h_pad=h_pad, w_pad=w_pad, rect=rect)
@_copy_docstring_and_deprecators(Figure.waitforbuttonpress)
def waitforbuttonpress(timeout: float = -1) -> None | bool:
    return gcf().waitforbuttonpress(timeout=timeout)
@_copy_docstring_and_deprecators(Axes.acorr)
def acorr(
    x: ArrayLike, *, data=None, **kwargs
) -> tuple[np.ndarray, np.ndarray, LineCollection | Line2D, Line2D | None]:
    return gca().acorr(x, **({"data": data} if data is not None else {}),
**kwargs)
@_copy_docstring_and_deprecators(Axes.angle_spectrum)
def angle_spectrum(
    x: ArrayLike,
    Fs: float | None = None,
    Fc: int | None = None,
    window: Callable[[ArrayLike], ArrayLike] | ArrayLike | None = None,
    pad_to: int | None = None,
    sides: Literal["default", "onesided", "twosided"] | None = None,
    *,
    data=None,
    **kwargs,
) -> tuple[np.ndarray, np.ndarray, Line2D]:
    return gca().angle_spectrum(
        x,
        Fs=Fs,
        Fc=Fc,
        window=window,
        pad_to=pad_to,
        sides=sides,
        **({"data": data} if data is not None else {}),
        **kwargs,
    )
@_copy_docstring_and_deprecators(Axes.annotate)
def annotate(
    text: str,
    xy: tuple[float, float],
    xytext: tuple[float, float] | None = None,
    xycoords: str
    | Artist
    | Transform
    | Callable[[RendererBase], Bbox | Transform]
    | tuple[float, float] = "data",
    textcoords: str
    | Artist
    | Transform
    | Callable[[RendererBase], Bbox | Transform]
    | tuple[float, float]
    | None = None,
    arrowprops: dict[str, Any] | None = None,
    annotation_clip: bool | None = None,
    **kwargs,
) -> Annotation:
    return gca().annotate(
        text,
        xy,
        xytext=xytext,
        xycoords=xycoords,
        textcoords=textcoords,
        arrowprops=arrowprops,
        annotation_clip=annotation_clip,
        **kwargs,
    )
@_copy_docstring_and_deprecators(Axes.arrow)
```

```
2122: (0)           def arrow(x: float, y: float, dx: float, dy: float, **kwargs) -> FancyArrow:
2123: (4)               return gca().arrow(x, y, dx, dy, **kwargs)
2124: (0)           @_copy_docstring_and_deprecators(Axes.autoscale)
2125: (0)           def autoscale(
2126: (4)               enable: bool = True,
2127: (4)               axis: Literal["both", "x", "y"] = "both",
2128: (4)               tight: bool | None = None,
2129: (0)           ) -> None:
2130: (4)               gca().autoscale(enable=enable, axis=axis, tight=tight)
2131: (0)           @_copy_docstring_and_deprecators(Axes.axhline)
2132: (0)           def axhline(y: float = 0, xmin: float = 0, xmax: float = 1, **kwargs) ->
Line2D:
2133: (4)               return gca().axhline(y=y, xmin=xmin, xmax=xmax, **kwargs)
2134: (0)           @_copy_docstring_and_deprecators(Axes.axhspan)
2135: (0)           def axhspan(
2136: (4)               ymin: float, ymax: float, xmin: float = 0, xmax: float = 1, **kwargs
2137: (0)           ) -> Rectangle:
2138: (4)               return gca().axhspan(ymin, ymax, xmin=xmin, xmax=xmax, **kwargs)
2139: (0)           @_copy_docstring_and_deprecators(Axes.axis)
2140: (0)           def axis(
2141: (4)               arg: tuple[float, float, float, float] | bool | str | None = None,
2142: (4)               /,
2143: (4)               *,
2144: (4)               emit: bool = True,
2145: (4)               **kwargs,
2146: (0)           ) -> tuple[float, float, float, float]:
2147: (4)               return gca().axis(arg, emit=emit, **kwargs)
2148: (0)           @_copy_docstring_and_deprecators(Axes.axline)
2149: (0)           def axline(
2150: (4)               xy1: tuple[float, float],
2151: (4)               xy2: tuple[float, float] | None = None,
2152: (4)               *,
2153: (4)               slope: float | None = None,
2154: (4)               **kwargs,
2155: (0)           ) -> AxLine:
2156: (4)               return gca().axline(xy1, xy2=xy2, slope=slope, **kwargs)
2157: (0)           @_copy_docstring_and_deprecators(Axes.axvline)
2158: (0)           def axvline(x: float = 0, ymin: float = 0, ymax: float = 1, **kwargs) ->
Line2D:
2159: (4)               return gca().axvline(x=x, ymin=ymin, ymax=ymax, **kwargs)
2160: (0)           @_copy_docstring_and_deprecators(Axes.axvspan)
2161: (0)           def axvspan(
2162: (4)               xmin: float, xmax: float, ymin: float = 0, ymax: float = 1, **kwargs
2163: (0)           ) -> Rectangle:
2164: (4)               return gca().axvspan(xmin, xmax, ymin=ymin, ymax=ymax, **kwargs)
2165: (0)           @_copy_docstring_and_deprecators(Axes.bar)
2166: (0)           def bar(
2167: (4)               x: float | ArrayLike,
2168: (4)               height: float | ArrayLike,
2169: (4)               width: float | ArrayLike = 0.8,
2170: (4)               bottom: float | ArrayLike | None = None,
2171: (4)               *,
2172: (4)               align: Literal["center", "edge"] = "center",
2173: (4)               data=None,
2174: (4)               **kwargs,
2175: (0)           ) -> BarContainer:
2176: (4)               return gca().bar(
2177: (8)                   x,
2178: (8)                   height,
2179: (8)                   width=width,
2180: (8)                   bottom=bottom,
2181: (8)                   align=align,
2182: (8)                   **({"data": data} if data is not None else {}),
2183: (8)                   **kwargs,
2184: (4)               )
2185: (0)           @_copy_docstring_and_deprecators(Axes.barbs)
2186: (0)           def barbs(*args, data=None, **kwargs) -> Barbs:
2187: (4)               return gca().barbs(*args, **({"data": data} if data is not None else {}),
**kwargs)
```

```
(0)  @_copy_docstring_and_deprecators(Axes.barh)
(0)  def barh(
(4)      y: float | ArrayLike,
(4)      width: float | ArrayLike,
(4)      height: float | ArrayLike = 0.8,
(4)      left: float | ArrayLike | None = None,
(4)      *,
(4)      align: Literal["center", "edge"] = "center",
(4)      data=None,
(4)      **kwargs,
(0)  ) -> BarContainer:
(4)      return gca().barh(
(8)          y,
(8)          width,
(8)          height=height,
(8)          left=left,
(8)          align=align,
(8)          **({"data": data} if data is not None else {}),
(8)          **kwargs,
(4)      )
(0)  @_copy_docstring_and_deprecators(Axes.bar_label)
(0)  def bar_label(
(4)      container: BarContainer,
(4)      labels: ArrayLike | None = None,
(4)      *,
(4)      fmt: str | Callable[[float], str] = "%g",
(4)      label_type: Literal["center", "edge"] = "edge",
(4)      padding: float = 0,
(4)      **kwargs,
(0)  ) -> list[Annotation]:
(4)      return gca().bar_label(
(8)          container,
(8)          labels=labels,
(8)          fmt=fmt,
(8)          label_type=label_type,
(8)          padding=padding,
(8)          **kwargs,
(4)      )
(0)  @_copy_docstring_and_deprecators(Axes.boxplot)
(0)  def boxplot(
(4)      x: ArrayLike | Sequence[ArrayLike],
(4)      notch: bool | None = None,
(4)      sym: str | None = None,
(4)      vert: bool | None = None,
(4)      whis: float | tuple[float, float] | None = None,
(4)      positions: ArrayLike | None = None,
(4)      widths: float | ArrayLike | None = None,
(4)      patch_artist: bool | None = None,
(4)      bootstrap: int | None = None,
(4)      usermedians: ArrayLike | None = None,
(4)      conf_intervals: ArrayLike | None = None,
(4)      meanline: bool | None = None,
(4)      showmeans: bool | None = None,
(4)      showcaps: bool | None = None,
(4)      showbox: bool | None = None,
(4)      showfliers: bool | None = None,
(4)      boxprops: dict[str, Any] | None = None,
(4)      tick_labels: Sequence[str] | None = None,
(4)      flierprops: dict[str, Any] | None = None,
(4)      medianprops: dict[str, Any] | None = None,
(4)      meanprops: dict[str, Any] | None = None,
(4)      capprops: dict[str, Any] | None = None,
(4)      whiskerprops: dict[str, Any] | None = None,
(4)      manage_ticks: bool = True,
(4)      autorange: bool = False,
(4)      zorder: float | None = None,
(4)      capwidths: float | ArrayLike | None = None,
(4)      label: Sequence[str] | None = None,
(4)      *,
```

```
2257: (4)                data=None,
2258: (0)            ) -> dict[str, Any]:
2259: (4)                return gca().boxplot(
2260: (8)                    x,
2261: (8)                    notch=notch,
2262: (8)                    sym=sym,
2263: (8)                    vert=vert,
2264: (8)                    whis=whis,
2265: (8)                    positions=positions,
2266: (8)                    widths=widths,
2267: (8)                    patch_artist=patch_artist,
2268: (8)                    bootstrap=bootstrap,
2269: (8)                    usermedians=usermedians,
2270: (8)                    conf_intervals=conf_intervals,
2271: (8)                    meanline=meanline,
2272: (8)                    showmeans=showmeans,
2273: (8)                    showcaps=showcaps,
2274: (8)                    showbox=showbox,
2275: (8)                    showfliers=showfliers,
2276: (8)                    boxprops=boxprops,
2277: (8)                    tick_labels=tick_labels,
2278: (8)                    flierprops=flierprops,
2279: (8)                    medianprops=medianprops,
2280: (8)                    meanprops=meanprops,
2281: (8)                    capprops=capprops,
2282: (8)                    whiskerprops=whiskerprops,
2283: (8)                    manage_ticks=manage_ticks,
2284: (8)                    autorange=autorange,
2285: (8)                    zorder=zorder,
2286: (8)                    capwidths=capwidths,
2287: (8)                    label=label,
2288: (8)                    **({"data": data} if data is not None else {}),
2289: (4)                )
2290: (0)            @_copy_docstring_and_deprecators(Axes.broken_barh)
2291: (0)            def broken_barh(
2292: (4)                xranges: Sequence[tuple[float, float]],
2293: (4)                yrange: tuple[float, float],
2294: (4)                *,
2295: (4)                data=None,
2296: (4)                **kwargs,
2297: (0)            ) -> PolyCollection:
2298: (4)                return gca().broken_barh(
2299: (8)                    xranges, yrange, **({"data": data} if data is not None else {}),
**kwargs
2300: (4)                )
2301: (0)            @_copy_docstring_and_deprecators(Axes.clabel)
2302: (0)            def clabel(CS: ContourSet, levels: ArrayLike | None = None, **kwargs) ->
list[Text]:
2303: (4)                return gca().clabel(CS, levels=levels, **kwargs)
2304: (0)            @_copy_docstring_and_deprecators(Axes.cohere)
2305: (0)            def cohere(
2306: (4)                x: ArrayLike,
2307: (4)                y: ArrayLike,
2308: (4)                NFFT: int = 256,
2309: (4)                Fs: float = 2,
2310: (4)                Fc: int = 0,
2311: (4)                detrend: Literal["none", "mean", "linear"]
2312: (4)                | Callable[[ArrayLike], ArrayLike] = mlab.detrend_none,
2313: (4)                window: Callable[[ArrayLike], ArrayLike] | ArrayLike =
mlab.window_hanning,
2314: (4)                noverlap: int = 0,
2315: (4)                pad_to: int | None = None,
2316: (4)                sides: Literal["default", "onesided", "twosided"] = "default",
2317: (4)                scale_by_freq: bool | None = None,
2318: (4)                *,
2319: (4)                data=None,
2320: (4)                **kwargs,
2321: (0)            ) -> tuple[np.ndarray, np.ndarray]:
2322: (4)                return gca().cohere(
```

```
(8)                 x,
(8)                 y,
(8)                 NFFT=NFFT,
(8)                 Fs=Fs,
(8)                 Fc=Fc,
(8)                 detrend=detrend,
(8)                 window=window,
(8)                 noverlap=noverlap,
(8)                 pad_to=pad_to,
(8)                 sides=sides,
(8)                 scale_by_freq=scale_by_freq,
(8)                 **({"data": data} if data is not None else {}),
(8)                 **kwargs,
(4)             )
(0)         @_copy_docstring_and_deprecators(Axes.contour)
(0)         def contour(*args, data=None, **kwargs) -> QuadContourSet:
(4)             __ret = gca().contour(
(8)                 *args, **({"data": data} if data is not None else {}), **kwargs
(4)             )
(4)             if __ret._A is not None:  # type: ignore[attr-defined]
(8)                 sci(__ret)
(4)             return __ret
(0)         @_copy_docstring_and_deprecators(Axes.contourf)
(0)         def contourf(*args, data=None, **kwargs) -> QuadContourSet:
(4)             __ret = gca().contourf(
(8)                 *args, **({"data": data} if data is not None else {}), **kwargs
(4)             )
(4)             if __ret._A is not None:  # type: ignore[attr-defined]
(8)                 sci(__ret)
(4)             return __ret
(0)         @_copy_docstring_and_deprecators(Axes.csd)
(0)         def csd(
(4)             x: ArrayLike,
(4)             y: ArrayLike,
(4)             NFFT: int | None = None,
(4)             Fs: float | None = None,
(4)             Fc: int | None = None,
(4)             detrend: Literal["none", "mean", "linear"]
(4)             | Callable[[ArrayLike], ArrayLike]
(4)             | None = None,
(4)             window: Callable[[ArrayLike], ArrayLike] | ArrayLike | None = None,
(4)             noverlap: int | None = None,
(4)             pad_to: int | None = None,
(4)             sides: Literal["default", "onesided", "twosided"] | None = None,
(4)             scale_by_freq: bool | None = None,
(4)             return_line: bool | None = None,
(4)             *,
(4)             data=None,
(4)             **kwargs,
(0)         ) -> tuple[np.ndarray, np.ndarray] | tuple[np.ndarray, np.ndarray, Line2D]:
(4)             return gca().csd(
(8)                 x,
(8)                 y,
(8)                 NFFT=NFFT,
(8)                 Fs=Fs,
(8)                 Fc=Fc,
(8)                 detrend=detrend,
(8)                 window=window,
(8)                 noverlap=noverlap,
(8)                 pad_to=pad_to,
(8)                 sides=sides,
(8)                 scale_by_freq=scale_by_freq,
(8)                 return_line=return_line,
(8)                 **({"data": data} if data is not None else {}),
(8)                 **kwargs,
(4)             )
(0)         @_copy_docstring_and_deprecators(Axes.ecdf)
(0)         def ecdf(
(4)             x: ArrayLike,
```

```
(4)             weights: ArrayLike | None = None,
(4)             *,
(4)             complementary: bool = False,
(4)             orientation: Literal["vertical", "horizonatal"] = "vertical",
(4)             compress: bool = False,
(4)             data=None,
(4)             **kwargs,
(0)         ) -> Line2D:
(4)             return gca().ecdf(
(8)                 x,
(8)                 weights=weights,
(8)                 complementary=complementary,
(8)                 orientation=orientation,
(8)                 compress=compress,
(8)                 **({"data": data} if data is not None else {}),
(8)                 **kwargs,
(4)             )
(0)         @_copy_docstring_and_deprecators(Axes.errorbar)
(0)         def errorbar(
(4)             x: float | ArrayLike,
(4)             y: float | ArrayLike,
(4)             yerr: float | ArrayLike | None = None,
(4)             xerr: float | ArrayLike | None = None,
(4)             fmt: str = "",
(4)             ecolor: ColorType | None = None,
(4)             elinewidth: float | None = None,
(4)             capsize: float | None = None,
(4)             barsabove: bool = False,
(4)             lolims: bool | ArrayLike = False,
(4)             uplims: bool | ArrayLike = False,
(4)             xlolims: bool | ArrayLike = False,
(4)             xuplims: bool | ArrayLike = False,
(4)             errorevery: int | tuple[int, int] = 1,
(4)             capthick: float | None = None,
(4)             *,
(4)             data=None,
(4)             **kwargs,
(0)         ) -> ErrorbarContainer:
(4)             return gca().errorbar(
(8)                 x,
(8)                 y,
(8)                 yerr=yerr,
(8)                 xerr=xerr,
(8)                 fmt=fmt,
(8)                 ecolor=ecolor,
(8)                 elinewidth=elinewidth,
(8)                 capsize=capsize,
(8)                 barsabove=barsabove,
(8)                 lolims=lolims,
(8)                 uplims=uplims,
(8)                 xlolims=xlolims,
(8)                 xuplims=xuplims,
(8)                 errorevery=errorevery,
(8)                 capthick=capthick,
(8)                 **({"data": data} if data is not None else {}),
(8)                 **kwargs,
(4)             )
(0)         @_copy_docstring_and_deprecators(Axes.eventplot)
(0)         def eventplot(
(4)             positions: ArrayLike | Sequence[ArrayLike],
(4)             orientation: Literal["horizontal", "vertical"] = "horizontal",
(4)             lineoffsets: float | Sequence[float] = 1,
(4)             linelengths: float | Sequence[float] = 1,
(4)             linewidths: float | Sequence[float] | None = None,
(4)             colors: ColorType | Sequence[ColorType] | None = None,
(4)             alpha: float | Sequence[float] | None = None,
(4)             linestyles: LineStyleType | Sequence[LineStyleType] = "solid",
(4)             *,
(4)             data=None,
```

```
(4)                 **kwargs,
(0)             ) -> EventCollection:
(4)                 return gca().eventplot(
(8)                     positions,
(8)                     orientation=orientation,
(8)                     lineoffsets=lineoffsets,
(8)                     linelengths=linelengths,
(8)                     linewidths=linewidths,
(8)                     colors=colors,
(8)                     alpha=alpha,
(8)                     linestyles=linestyles,
(8)                     **({"data": data} if data is not None else {}),
(8)                     **kwargs,
(4)                 )
(0)             @_copy_docstring_and_deprecators(Axes.fill)
(0)             def fill(*args, data=None, **kwargs) -> list[Polygon]:
(4)                 return gca().fill(*args, **({"data": data} if data is not None else {}),
**kwargs)
(0)             @_copy_docstring_and_deprecators(Axes.fill_between)
(0)             def fill_between(
(4)                 x: ArrayLike,
(4)                 y1: ArrayLike | float,
(4)                 y2: ArrayLike | float = 0,
(4)                 where: Sequence[bool] | None = None,
(4)                 interpolate: bool = False,
(4)                 step: Literal["pre", "post", "mid"] | None = None,
(4)                 *,
(4)                 data=None,
(4)                 **kwargs,
(0)             ) -> PolyCollection:
(4)                 return gca().fill_between(
(8)                     x,
(8)                     y1,
(8)                     y2=y2,
(8)                     where=where,
(8)                     interpolate=interpolate,
(8)                     step=step,
(8)                     **({"data": data} if data is not None else {}),
(8)                     **kwargs,
(4)                 )
(0)             @_copy_docstring_and_deprecators(Axes.fill_betweenx)
(0)             def fill_betweenx(
(4)                 y: ArrayLike,
(4)                 x1: ArrayLike | float,
(4)                 x2: ArrayLike | float = 0,
(4)                 where: Sequence[bool] | None = None,
(4)                 step: Literal["pre", "post", "mid"] | None = None,
(4)                 interpolate: bool = False,
(4)                 *,
(4)                 data=None,
(4)                 **kwargs,
(0)             ) -> PolyCollection:
(4)                 return gca().fill_betweenx(
(8)                     y,
(8)                     x1,
(8)                     x2=x2,
(8)                     where=where,
(8)                     step=step,
(8)                     interpolate=interpolate,
(8)                     **({"data": data} if data is not None else {}),
(8)                     **kwargs,
(4)                 )
(0)             @_copy_docstring_and_deprecators(Axes.grid)
(0)             def grid(
(4)                 visible: bool | None = None,
(4)                 which: Literal["major", "minor", "both"] = "major",
(4)                 axis: Literal["both", "x", "y"] = "both",
(4)                 **kwargs,
(0)             ) -> None:
```

```
2529: (4)                gca().grid(visible=visible, which=which, axis=axis, **kwargs)
2530: (0)            @_copy_docstring_and_deprecators(Axes.hexbin)
2531: (0)            def hexbin(
2532: (4)                x: ArrayLike,
2533: (4)                y: ArrayLike,
2534: (4)                C: ArrayLike | None = None,
2535: (4)                gridsize: int | tuple[int, int] = 100,
2536: (4)                bins: Literal["log"] | int | Sequence[float] | None = None,
2537: (4)                xscale: Literal["linear", "log"] = "linear",
2538: (4)                yscale: Literal["linear", "log"] = "linear",
2539: (4)                extent: tuple[float, float, float, float] | None = None,
2540: (4)                cmap: str | Colormap | None = None,
2541: (4)                norm: str | Normalize | None = None,
2542: (4)                vmin: float | None = None,
2543: (4)                vmax: float | None = None,
2544: (4)                alpha: float | None = None,
2545: (4)                linewidths: float | None = None,
2546: (4)                edgecolors: Literal["face", "none"] | ColorType = "face",
2547: (4)                reduce_C_function: Callable[[np.ndarray | list[float]], float] = np.mean,
2548: (4)                mincnt: int | None = None,
2549: (4)                marginals: bool = False,
2550: (4)                *,
2551: (4)                data=None,
2552: (4)                **kwargs,
2553: (0)            ) -> PolyCollection:
2554: (4)                __ret = gca().hexbin(
2555: (8)                    x,
2556: (8)                    y,
2557: (8)                    C=C,
2558: (8)                    gridsize=gridsize,
2559: (8)                    bins=bins,
2560: (8)                    xscale=xscale,
2561: (8)                    yscale=yscale,
2562: (8)                    extent=extent,
2563: (8)                    cmap=cmap,
2564: (8)                    norm=norm,
2565: (8)                    vmin=vmin,
2566: (8)                    vmax=vmax,
2567: (8)                    alpha=alpha,
2568: (8)                    linewidths=linewidths,
2569: (8)                    edgecolors=edgecolors,
2570: (8)                    reduce_C_function=reduce_C_function,
2571: (8)                    mincnt=mincnt,
2572: (8)                    marginals=marginals,
2573: (8)                    **({"data": data} if data is not None else {}),
2574: (8)                    **kwargs,
2575: (4)                )
2576: (4)                sci(__ret)
2577: (4)                return __ret
2578: (0)            @_copy_docstring_and_deprecators(Axes.hist)
2579: (0)            def hist(
2580: (4)                x: ArrayLike | Sequence[ArrayLike],
2581: (4)                bins: int | Sequence[float] | str | None = None,
2582: (4)                range: tuple[float, float] | None = None,
2583: (4)                density: bool = False,
2584: (4)                weights: ArrayLike | None = None,
2585: (4)                cumulative: bool | float = False,
2586: (4)                bottom: ArrayLike | float | None = None,
2587: (4)                histtype: Literal["bar", "barstacked", "step", "stepfilled"] = "bar",
2588: (4)                align: Literal["left", "mid", "right"] = "mid",
2589: (4)                orientation: Literal["vertical", "horizontal"] = "vertical",
2590: (4)                rwidth: float | None = None,
2591: (4)                log: bool = False,
2592: (4)                color: ColorType | Sequence[ColorType] | None = None,
2593: (4)                label: str | Sequence[str] | None = None,
2594: (4)                stacked: bool = False,
2595: (4)                *,
2596: (4)                data=None,
2597: (4)                **kwargs,
```

```
(0)         ) -> tuple[
(4)             np.ndarray | list[np.ndarray],
(4)             np.ndarray,
(4)             BarContainer | Polygon | list[BarContainer | Polygon],
(0)         ]:
(4)             return gca().hist(
(8)                 x,
(8)                 bins=bins,
(8)                 range=range,
(8)                 density=density,
(8)                 weights=weights,
(8)                 cumulative=cumulative,
(8)                 bottom=bottom,
(8)                 histtype=histtype,
(8)                 align=align,
(8)                 orientation=orientation,
(8)                 rwidth=rwidth,
(8)                 log=log,
(8)                 color=color,
(8)                 label=label,
(8)                 stacked=stacked,
(8)                 **({"data": data} if data is not None else {}),
(8)                 **kwargs,
(4)             )
(0)         @_copy_docstring_and_deprecators(Axes.stairs)
(0)         def stairs(
(4)             values: ArrayLike,
(4)             edges: ArrayLike | None = None,
(4)             *,
(4)             orientation: Literal["vertical", "horizontal"] = "vertical",
(4)             baseline: float | ArrayLike | None = 0,
(4)             fill: bool = False,
(4)             data=None,
(4)             **kwargs,
(0)         ) -> StepPatch:
(4)             return gca().stairs(
(8)                 values,
(8)                 edges=edges,
(8)                 orientation=orientation,
(8)                 baseline=baseline,
(8)                 fill=fill,
(8)                 **({"data": data} if data is not None else {}),
(8)                 **kwargs,
(4)             )
(0)         @_copy_docstring_and_deprecators(Axes.hist2d)
(0)         def hist2d(
(4)             x: ArrayLike,
(4)             y: ArrayLike,
(4)             bins: None | int | tuple[int, int] | ArrayLike | tuple[ArrayLike,
ArrayLike] = 10,
(4)             range: ArrayLike | None = None,
(4)             density: bool = False,
(4)             weights: ArrayLike | None = None,
(4)             cmin: float | None = None,
(4)             cmax: float | None = None,
(4)             *,
(4)             data=None,
(4)             **kwargs,
(0)         ) -> tuple[np.ndarray, np.ndarray, np.ndarray, QuadMesh]:
(4)             __ret = gca().hist2d(
(8)                 x,
(8)                 y,
(8)                 bins=bins,
(8)                 range=range,
(8)                 density=density,
(8)                 weights=weights,
(8)                 cmin=cmin,
(8)                 cmax=cmax,
(8)                 **({"data": data} if data is not None else {}),
```

```
(8)                 **kwargs,
(4)             )
(4)             sci(__ret[-1])
(4)             return __ret
(0)         @_copy_docstring_and_deprecators(Axes.hlines)
(0)         def hlines(
(4)             y: float | ArrayLike,
(4)             xmin: float | ArrayLike,
(4)             xmax: float | ArrayLike,
(4)             colors: ColorType | Sequence[ColorType] | None = None,
(4)             linestyles: LineStyleType = "solid",
(4)             label: str = "",
(4)             *,
(4)             data=None,
(4)             **kwargs,
(0)         ) -> LineCollection:
(4)             return gca().hlines(
(8)                 y,
(8)                 xmin,
(8)                 xmax,
(8)                 colors=colors,
(8)                 linestyles=linestyles,
(8)                 label=label,
(8)                 **({"data": data} if data is not None else {}),
(8)                 **kwargs,
(4)             )
(0)         @_copy_docstring_and_deprecators(Axes.imshow)
(0)         def imshow(
(4)             X: ArrayLike | PIL.Image.Image,
(4)             cmap: str | Colormap | None = None,
(4)             norm: str | Normalize | None = None,
(4)             *,
(4)             aspect: Literal["equal", "auto"] | float | None = None,
(4)             interpolation: str | None = None,
(4)             alpha: float | ArrayLike | None = None,
(4)             vmin: float | None = None,
(4)             vmax: float | None = None,
(4)             origin: Literal["upper", "lower"] | None = None,
(4)             extent: tuple[float, float, float, float] | None = None,
(4)             interpolation_stage: Literal["data", "rgba"] | None = None,
(4)             filternorm: bool = True,
(4)             filterrad: float = 4.0,
(4)             resample: bool | None = None,
(4)             url: str | None = None,
(4)             data=None,
(4)             **kwargs,
(0)         ) -> AxesImage:
(4)             __ret = gca().imshow(
(8)                 X,
(8)                 cmap=cmap,
(8)                 norm=norm,
(8)                 aspect=aspect,
(8)                 interpolation=interpolation,
(8)                 alpha=alpha,
(8)                 vmin=vmin,
(8)                 vmax=vmax,
(8)                 origin=origin,
(8)                 extent=extent,
(8)                 interpolation_stage=interpolation_stage,
(8)                 filternorm=filternorm,
(8)                 filterrad=filterrad,
(8)                 resample=resample,
(8)                 url=url,
(8)                 **({"data": data} if data is not None else {}),
(8)                 **kwargs,
(4)             )
(4)             sci(__ret)
(4)             return __ret
(0)         @_copy_docstring_and_deprecators(Axes.legend)
```

```
(0)  def legend(*args, **kwargs) -> Legend:
(4)      return gca().legend(*args, **kwargs)
(0)  @_copy_docstring_and_deprecators(Axes.locator_params)
(0)  def locator_params(
(4)      axis: Literal["both", "x", "y"] = "both", tight: bool | None = None, **kwargs
(0)  ) -> None:
(4)      gca().locator_params(axis=axis, tight=tight, **kwargs)
(0)  @_copy_docstring_and_deprecators(Axes.loglog)
(0)  def loglog(*args, **kwargs) -> list[Line2D]:
(4)      return gca().loglog(*args, **kwargs)
(0)  @_copy_docstring_and_deprecators(Axes.magnitude_spectrum)
(0)  def magnitude_spectrum(
(4)      x: ArrayLike,
(4)      Fs: float | None = None,
(4)      Fc: int | None = None,
(4)      window: Callable[[ArrayLike], ArrayLike] | ArrayLike | None = None,
(4)      pad_to: int | None = None,
(4)      sides: Literal["default", "onesided", "twosided"] | None = None,
(4)      scale: Literal["default", "linear", "dB"] | None = None,
(4)      *,
(4)      data=None,
(4)      **kwargs,
(0)  ) -> tuple[np.ndarray, np.ndarray, Line2D]:
(4)      return gca().magnitude_spectrum(
(8)          x,
(8)          Fs=Fs,
(8)          Fc=Fc,
(8)          window=window,
(8)          pad_to=pad_to,
(8)          sides=sides,
(8)          scale=scale,
(8)          **({"data": data} if data is not None else {}),
(8)          **kwargs,
(4)      )
(0)  @_copy_docstring_and_deprecators(Axes.margins)
(0)  def margins(
(4)      *margins: float,
(4)      x: float | None = None,
(4)      y: float | None = None,
(4)      tight: bool | None = True,
(0)  ) -> tuple[float, float] | None:
(4)      return gca().margins(*margins, x=x, y=y, tight=tight)
(0)  @_copy_docstring_and_deprecators(Axes.minorticks_off)
(0)  def minorticks_off() -> None:
(4)      gca().minorticks_off()
(0)  @_copy_docstring_and_deprecators(Axes.minorticks_on)
(0)  def minorticks_on() -> None:
(4)      gca().minorticks_on()
(0)  @_copy_docstring_and_deprecators(Axes.pcolor)
(0)  def pcolor(
(4)      *args: ArrayLike,
(4)      shading: Literal["flat", "nearest", "auto"] | None = None,
(4)      alpha: float | None = None,
(4)      norm: str | Normalize | None = None,
(4)      cmap: str | Colormap | None = None,
(4)      vmin: float | None = None,
(4)      vmax: float | None = None,
(4)      data=None,
(4)      **kwargs,
(0)  ) -> Collection:
(4)      __ret = gca().pcolor(
(8)          *args,
(8)          shading=shading,
(8)          alpha=alpha,
(8)          norm=norm,
(8)          cmap=cmap,
(8)          vmin=vmin,
(8)          vmax=vmax,
```

```
(8)                 **({"data": data} if data is not None else {}),
(8)                 **kwargs,
(4)             )
(4)             sci(__ret)
(4)             return __ret
(0)         @_copy_docstring_and_deprecators(Axes.pcolormesh)
(0)         def pcolormesh(
(4)             *args: ArrayLike,
(4)             alpha: float | None = None,
(4)             norm: str | Normalize | None = None,
(4)             cmap: str | Colormap | None = None,
(4)             vmin: float | None = None,
(4)             vmax: float | None = None,
(4)             shading: Literal["flat", "nearest", "gouraud", "auto"] | None = None,
(4)             antialiased: bool = False,
(4)             data=None,
(4)             **kwargs,
(0)         ) -> QuadMesh:
(4)             __ret = gca().pcolormesh(
(8)                 *args,
(8)                 alpha=alpha,
(8)                 norm=norm,
(8)                 cmap=cmap,
(8)                 vmin=vmin,
(8)                 vmax=vmax,
(8)                 shading=shading,
(8)                 antialiased=antialiased,
(8)                 **({"data": data} if data is not None else {}),
(8)                 **kwargs,
(4)             )
(4)             sci(__ret)
(4)             return __ret
(0)         @_copy_docstring_and_deprecators(Axes.phase_spectrum)
(0)         def phase_spectrum(
(4)             x: ArrayLike,
(4)             Fs: float | None = None,
(4)             Fc: int | None = None,
(4)             window: Callable[[ArrayLike], ArrayLike] | ArrayLike | None = None,
(4)             pad_to: int | None = None,
(4)             sides: Literal["default", "onesided", "twosided"] | None = None,
(4)             *,
(4)             data=None,
(4)             **kwargs,
(0)         ) -> tuple[np.ndarray, np.ndarray, Line2D]:
(4)             return gca().phase_spectrum(
(8)                 x,
(8)                 Fs=Fs,
(8)                 Fc=Fc,
(8)                 window=window,
(8)                 pad_to=pad_to,
(8)                 sides=sides,
(8)                 **({"data": data} if data is not None else {}),
(8)                 **kwargs,
(4)             )
(0)         @_copy_docstring_and_deprecators(Axes.pie)
(0)         def pie(
(4)             x: ArrayLike,
(4)             explode: ArrayLike | None = None,
(4)             labels: Sequence[str] | None = None,
(4)             colors: ColorType | Sequence[ColorType] | None = None,
(4)             autopct: str | Callable[[float], str] | None = None,
(4)             pctdistance: float = 0.6,
(4)             shadow: bool = False,
(4)             labeldistance: float | None = 1.1,
(4)             startangle: float = 0,
(4)             radius: float = 1,
(4)             counterclock: bool = True,
(4)             wedgeprops: dict[str, Any] | None = None,
(4)             textprops: dict[str, Any] | None = None,
```

```
(4)             center: tuple[float, float] = (0, 0),
(4)             frame: bool = False,
(4)             rotatelabels: bool = False,
(4)             *,
(4)             normalize: bool = True,
(4)             hatch: str | Sequence[str] | None = None,
(4)             data=None,
(0)         ) -> tuple[list[Wedge], list[Text]] | tuple[list[Wedge], list[Text],
list[Text]]:
(4)             return gca().pie(
(8)                 x,
(8)                 explode=explode,
(8)                 labels=labels,
(8)                 colors=colors,
(8)                 autopct=autopct,
(8)                 pctdistance=pctdistance,
(8)                 shadow=shadow,
(8)                 labeldistance=labeldistance,
(8)                 startangle=startangle,
(8)                 radius=radius,
(8)                 counterclock=counterclock,
(8)                 wedgeprops=wedgeprops,
(8)                 textprops=textprops,
(8)                 center=center,
(8)                 frame=frame,
(8)                 rotatelabels=rotatelabels,
(8)                 normalize=normalize,
(8)                 hatch=hatch,
(8)                 **({"data": data} if data is not None else {}),
(4)             )
(0)         @_copy_docstring_and_deprecators(Axes.plot)
(0)         def plot(
(4)             *args: float | ArrayLike | str,
(4)             scalex: bool = True,
(4)             scaley: bool = True,
(4)             data=None,
(4)             **kwargs,
(0)         ) -> list[Line2D]:
(4)             return gca().plot(
(8)                 *args,
(8)                 scalex=scalex,
(8)                 scaley=scaley,
(8)                 **({"data": data} if data is not None else {}),
(8)                 **kwargs,
(4)             )
(0)         @_copy_docstring_and_deprecators(Axes.plot_date)
(0)         def plot_date(
(4)             x: ArrayLike,
(4)             y: ArrayLike,
(4)             fmt: str = "o",
(4)             tz: str | datetime.tzinfo | None = None,
(4)             xdate: bool = True,
(4)             ydate: bool = False,
(4)             *,
(4)             data=None,
(4)             **kwargs,
(0)         ) -> list[Line2D]:
(4)             return gca().plot_date(
(8)                 x,
(8)                 y,
(8)                 fmt=fmt,
(8)                 tz=tz,
(8)                 xdate=xdate,
(8)                 ydate=ydate,
(8)                 **({"data": data} if data is not None else {}),
(8)                 **kwargs,
(4)             )
(0)         @_copy_docstring_and_deprecators(Axes.psd)
(0)         def psd(
```

```
(4)             x: ArrayLike,
(4)             NFFT: int | None = None,
(4)             Fs: float | None = None,
(4)             Fc: int | None = None,
(4)             detrend: Literal["none", "mean", "linear"]
(4)             | Callable[[ArrayLike], ArrayLike]
(4)             | None = None,
(4)             window: Callable[[ArrayLike], ArrayLike] | ArrayLike | None = None,
(4)             noverlap: int | None = None,
(4)             pad_to: int | None = None,
(4)             sides: Literal["default", "onesided", "twosided"] | None = None,
(4)             scale_by_freq: bool | None = None,
(4)             return_line: bool | None = None,
(4)             *,
(4)             data=None,
(4)             **kwargs,
(0)         ) -> tuple[np.ndarray, np.ndarray] | tuple[np.ndarray, np.ndarray, Line2D]:
(4)             return gca().psd(
(8)                 x,
(8)                 NFFT=NFFT,
(8)                 Fs=Fs,
(8)                 Fc=Fc,
(8)                 detrend=detrend,
(8)                 window=window,
(8)                 noverlap=noverlap,
(8)                 pad_to=pad_to,
(8)                 sides=sides,
(8)                 scale_by_freq=scale_by_freq,
(8)                 return_line=return_line,
(8)                 **({"data": data} if data is not None else {}),
(8)                 **kwargs,
(4)             )
(0)         @_copy_docstring_and_deprecators(Axes.quiver)
(0)         def quiver(*args, data=None, **kwargs) -> Quiver:
(4)             __ret = gca().quiver(
(8)                 *args, **({"data": data} if data is not None else {}), **kwargs
(4)             )
(4)             sci(__ret)
(4)             return __ret
(0)         @_copy_docstring_and_deprecators(Axes.quiverkey)
(0)         def quiverkey(
(4)             Q: Quiver, X: float, Y: float, U: float, label: str, **kwargs
(0)         ) -> QuiverKey:
(4)             return gca().quiverkey(Q, X, Y, U, label, **kwargs)
(0)         @_copy_docstring_and_deprecators(Axes.scatter)
(0)         def scatter(
(4)             x: float | ArrayLike,
(4)             y: float | ArrayLike,
(4)             s: float | ArrayLike | None = None,
(4)             c: ArrayLike | Sequence[ColorType] | ColorType | None = None,
(4)             marker: MarkerType | None = None,
(4)             cmap: str | Colormap | None = None,
(4)             norm: str | Normalize | None = None,
(4)             vmin: float | None = None,
(4)             vmax: float | None = None,
(4)             alpha: float | None = None,
(4)             linewidths: float | Sequence[float] | None = None,
(4)             *,
(4)             edgecolors: Literal["face", "none"] | ColorType | Sequence[ColorType] |
None = None,
(4)             plotnonfinite: bool = False,
(4)             data=None,
(4)             **kwargs,
(0)         ) -> PathCollection:
(4)             __ret = gca().scatter(
(8)                 x,
(8)                 y,
(8)                 s=s,
(8)                 c=c,
```

```
3008: (8)                   marker=marker,
3009: (8)                   cmap=cmap,
3010: (8)                   norm=norm,
3011: (8)                   vmin=vmin,
3012: (8)                   vmax=vmax,
3013: (8)                   alpha=alpha,
3014: (8)                   linewidths=linewidths,
3015: (8)                   edgecolors=edgecolors,
3016: (8)                   plotnonfinite=plotnonfinite,
3017: (8)                   **({"data": data} if data is not None else {}),
3018: (8)                   **kwargs,
3019: (4)               )
3020: (4)               sci(__ret)
3021: (4)               return __ret
3022: (0)           @_copy_docstring_and_deprecators(Axes.semilogx)
3023: (0)           def semilogx(*args, **kwargs) -> list[Line2D]:
3024: (4)               return gca().semilogx(*args, **kwargs)
3025: (0)           @_copy_docstring_and_deprecators(Axes.semilogy)
3026: (0)           def semilogy(*args, **kwargs) -> list[Line2D]:
3027: (4)               return gca().semilogy(*args, **kwargs)
3028: (0)           @_copy_docstring_and_deprecators(Axes.specgram)
3029: (0)           def specgram(
3030: (4)               x: ArrayLike,
3031: (4)               NFFT: int | None = None,
3032: (4)               Fs: float | None = None,
3033: (4)               Fc: int | None = None,
3034: (4)               detrend: Literal["none", "mean", "linear"]
3035: (4)               | Callable[[ArrayLike], ArrayLike]
3036: (4)               | None = None,
3037: (4)               window: Callable[[ArrayLike], ArrayLike] | ArrayLike | None = None,
3038: (4)               noverlap: int | None = None,
3039: (4)               cmap: str | Colormap | None = None,
3040: (4)               xextent: tuple[float, float] | None = None,
3041: (4)               pad_to: int | None = None,
3042: (4)               sides: Literal["default", "onesided", "twosided"] | None = None,
3043: (4)               scale_by_freq: bool | None = None,
3044: (4)               mode: Literal["default", "psd", "magnitude", "angle", "phase"] | None =
None,
3045: (4)               scale: Literal["default", "linear", "dB"] | None = None,
3046: (4)               vmin: float | None = None,
3047: (4)               vmax: float | None = None,
3048: (4)               *,
3049: (4)               data=None,
3050: (4)               **kwargs,
3051: (0)           ) -> tuple[np.ndarray, np.ndarray, np.ndarray, AxesImage]:
3052: (4)               __ret = gca().specgram(
3053: (8)                   x,
3054: (8)                   NFFT=NFFT,
3055: (8)                   Fs=Fs,
3056: (8)                   Fc=Fc,
3057: (8)                   detrend=detrend,
3058: (8)                   window=window,
3059: (8)                   noverlap=noverlap,
3060: (8)                   cmap=cmap,
3061: (8)                   xextent=xextent,
3062: (8)                   pad_to=pad_to,
3063: (8)                   sides=sides,
3064: (8)                   scale_by_freq=scale_by_freq,
3065: (8)                   mode=mode,
3066: (8)                   scale=scale,
3067: (8)                   vmin=vmin,
3068: (8)                   vmax=vmax,
3069: (8)                   **({"data": data} if data is not None else {}),
3070: (8)                   **kwargs,
3071: (4)               )
3072: (4)               sci(__ret[-1])
3073: (4)               return __ret
3074: (0)           @_copy_docstring_and_deprecators(Axes.spy)
3075: (0)           def spy(
```

```
3076: (4)               Z: ArrayLike,
3077: (4)               precision: float | Literal["present"] = 0,
3078: (4)               marker: str | None = None,
3079: (4)               markersize: float | None = None,
3080: (4)               aspect: Literal["equal", "auto"] | float | None = "equal",
3081: (4)               origin: Literal["upper", "lower"] = "upper",
3082: (4)               **kwargs,
3083: (0)           ) -> AxesImage:
3084: (4)               __ret = gca().spy(
3085: (8)                   Z,
3086: (8)                   precision=precision,
3087: (8)                   marker=marker,
3088: (8)                   markersize=markersize,
3089: (8)                   aspect=aspect,
3090: (8)                   origin=origin,
3091: (8)                   **kwargs,
3092: (4)               )
3093: (4)               if isinstance(__ret, cm.ScalarMappable):
3094: (8)                   sci(__ret)  # noqa
3095: (4)               return __ret
3096: (0)           @_copy_docstring_and_deprecators(Axes.stackplot)
3097: (0)           def stackplot(
3098: (4)               x, *args, labels=(), colors=None, hatch=None, baseline="zero", data=None,
**kwargs
3099: (0)           ):
3100: (4)               return gca().stackplot(
3101: (8)                   x,
3102: (8)                   *args,
3103: (8)                   labels=labels,
3104: (8)                   colors=colors,
3105: (8)                   hatch=hatch,
3106: (8)                   baseline=baseline,
3107: (8)                   **({"data": data} if data is not None else {}),
3108: (8)                   **kwargs,
3109: (4)               )
3110: (0)           @_copy_docstring_and_deprecators(Axes.stem)
3111: (0)           def stem(
3112: (4)               *args: ArrayLike | str,
3113: (4)               linefmt: str | None = None,
3114: (4)               markerfmt: str | None = None,
3115: (4)               basefmt: str | None = None,
3116: (4)               bottom: float = 0,
3117: (4)               label: str | None = None,
3118: (4)               orientation: Literal["vertical", "horizontal"] = "vertical",
3119: (4)               data=None,
3120: (0)           ) -> StemContainer:
3121: (4)               return gca().stem(
3122: (8)                   *args,
3123: (8)                   linefmt=linefmt,
3124: (8)                   markerfmt=markerfmt,
3125: (8)                   basefmt=basefmt,
3126: (8)                   bottom=bottom,
3127: (8)                   label=label,
3128: (8)                   orientation=orientation,
3129: (8)                   **({"data": data} if data is not None else {}),
3130: (4)               )
3131: (0)           @_copy_docstring_and_deprecators(Axes.step)
3132: (0)           def step(
3133: (4)               x: ArrayLike,
3134: (4)               y: ArrayLike,
3135: (4)               *args,
3136: (4)               where: Literal["pre", "post", "mid"] = "pre",
3137: (4)               data=None,
3138: (4)               **kwargs,
3139: (0)           ) -> list[Line2D]:
3140: (4)               return gca().step(
3141: (8)                   x,
3142: (8)                   y,
3143: (8)                   *args,
```

```
3144: (8)                   where=where,
3145: (8)                   **({"data": data} if data is not None else {}),
3146: (8)                   **kwargs,
3147: (4)               )
3148: (0)           @_copy_docstring_and_deprecators(Axes.streamplot)
3149: (0)           def streamplot(
3150: (4)               x,
3151: (4)               y,
3152: (4)               u,
3153: (4)               v,
3154: (4)               density=1,
3155: (4)               linewidth=None,
3156: (4)               color=None,
3157: (4)               cmap=None,
3158: (4)               norm=None,
3159: (4)               arrowsize=1,
3160: (4)               arrowstyle="-|>",
3161: (4)               minlength=0.1,
3162: (4)               transform=None,
3163: (4)               zorder=None,
3164: (4)               start_points=None,
3165: (4)               maxlength=4.0,
3166: (4)               integration_direction="both",
3167: (4)               broken_streamlines=True,
3168: (4)               *,
3169: (4)               data=None,
3170: (0)           ):
3171: (4)               __ret = gca().streamplot(
3172: (8)                   x,
3173: (8)                   y,
3174: (8)                   u,
3175: (8)                   v,
3176: (8)                   density=density,
3177: (8)                   linewidth=linewidth,
3178: (8)                   color=color,
3179: (8)                   cmap=cmap,
3180: (8)                   norm=norm,
3181: (8)                   arrowsize=arrowsize,
3182: (8)                   arrowstyle=arrowstyle,
3183: (8)                   minlength=minlength,
3184: (8)                   transform=transform,
3185: (8)                   zorder=zorder,
3186: (8)                   start_points=start_points,
3187: (8)                   maxlength=maxlength,
3188: (8)                   integration_direction=integration_direction,
3189: (8)                   broken_streamlines=broken_streamlines,
3190: (8)                   **({"data": data} if data is not None else {}),
3191: (4)               )
3192: (4)               sci(__ret.lines)
3193: (4)               return __ret
3194: (0)           @_copy_docstring_and_deprecators(Axes.table)
3195: (0)           def table(
3196: (4)               cellText=None,
3197: (4)               cellColours=None,
3198: (4)               cellLoc="right",
3199: (4)               colWidths=None,
3200: (4)               rowLabels=None,
3201: (4)               rowColours=None,
3202: (4)               rowLoc="left",
3203: (4)               colLabels=None,
3204: (4)               colColours=None,
3205: (4)               colLoc="center",
3206: (4)               loc="bottom",
3207: (4)               bbox=None,
3208: (4)               edges="closed",
3209: (4)               **kwargs,
3210: (0)           ):
3211: (4)               return gca().table(
3212: (8)                   cellText=cellText,
```

```
3213: (8)                 cellColours=cellColours,
3214: (8)                 cellLoc=cellLoc,
3215: (8)                 colWidths=colWidths,
3216: (8)                 rowLabels=rowLabels,
3217: (8)                 rowColours=rowColours,
3218: (8)                 rowLoc=rowLoc,
3219: (8)                 colLabels=colLabels,
3220: (8)                 colColours=colColours,
3221: (8)                 colLoc=colLoc,
3222: (8)                 loc=loc,
3223: (8)                 bbox=bbox,
3224: (8)                 edges=edges,
3225: (8)                 **kwargs,
3226: (4)             )
3227: (0)         @_copy_docstring_and_deprecators(Axes.text)
3228: (0)         def text(
3229: (4)             x: float, y: float, s: str, fontdict: dict[str, Any] | None = None,
**kwargs
3230: (0)         ) -> Text:
3231: (4)             return gca().text(x, y, s, fontdict=fontdict, **kwargs)
3232: (0)         @_copy_docstring_and_deprecators(Axes.tick_params)
3233: (0)         def tick_params(axis: Literal["both", "x", "y"] = "both", **kwargs) -> None:
3234: (4)             gca().tick_params(axis=axis, **kwargs)
3235: (0)         @_copy_docstring_and_deprecators(Axes.ticklabel_format)
3236: (0)         def ticklabel_format(
3237: (4)             *,
3238: (4)             axis: Literal["both", "x", "y"] = "both",
3239: (4)             style: Literal["", "sci", "scientific", "plain"] | None = None,
3240: (4)             scilimits: tuple[int, int] | None = None,
3241: (4)             useOffset: bool | float | None = None,
3242: (4)             useLocale: bool | None = None,
3243: (4)             useMathText: bool | None = None,
3244: (0)         ) -> None:
3245: (4)             gca().ticklabel_format(
3246: (8)                 axis=axis,
3247: (8)                 style=style,
3248: (8)                 scilimits=scilimits,
3249: (8)                 useOffset=useOffset,
3250: (8)                 useLocale=useLocale,
3251: (8)                 useMathText=useMathText,
3252: (4)             )
3253: (0)         @_copy_docstring_and_deprecators(Axes.tricontour)
3254: (0)         def tricontour(*args, **kwargs):
3255: (4)             __ret = gca().tricontour(*args, **kwargs)
3256: (4)             if __ret._A is not None:  # type: ignore[attr-defined]
3257: (8)                 sci(__ret)
3258: (4)             return __ret
3259: (0)         @_copy_docstring_and_deprecators(Axes.tricontourf)
3260: (0)         def tricontourf(*args, **kwargs):
3261: (4)             __ret = gca().tricontourf(*args, **kwargs)
3262: (4)             if __ret._A is not None:  # type: ignore[attr-defined]
3263: (8)                 sci(__ret)
3264: (4)             return __ret
3265: (0)         @_copy_docstring_and_deprecators(Axes.tripcolor)
3266: (0)         def tripcolor(
3267: (4)             *args,
3268: (4)             alpha=1.0,
3269: (4)             norm=None,
3270: (4)             cmap=None,
3271: (4)             vmin=None,
3272: (4)             vmax=None,
3273: (4)             shading="flat",
3274: (4)             facecolors=None,
3275: (4)             **kwargs,
3276: (0)         ):
3277: (4)             __ret = gca().tripcolor(
3278: (8)                 *args,
3279: (8)                 alpha=alpha,
3280: (8)                 norm=norm,
```

```
(8)                 cmap=cmap,
(8)                 vmin=vmin,
(8)                 vmax=vmax,
(8)                 shading=shading,
(8)                 facecolors=facecolors,
(8)                 **kwargs,
(4)             )
(4)             sci(__ret)
(4)             return __ret
(0)         @_copy_docstring_and_deprecators(Axes.triplot)
(0)         def triplot(*args, **kwargs):
(4)             return gca().triplot(*args, **kwargs)
(0)         @_copy_docstring_and_deprecators(Axes.violinplot)
(0)         def violinplot(
(4)             dataset: ArrayLike | Sequence[ArrayLike],
(4)             positions: ArrayLike | None = None,
(4)             vert: bool = True,
(4)             widths: float | ArrayLike = 0.5,
(4)             showmeans: bool = False,
(4)             showextrema: bool = True,
(4)             showmedians: bool = False,
(4)             quantiles: Sequence[float | Sequence[float]] | None = None,
(4)             points: int = 100,
(4)             bw_method: Literal["scott", "silverman"]
(4)             | float
(4)             | Callable[[GaussianKDE], float]
(4)             | None = None,
(4)             side: Literal["both", "low", "high"] = "both",
(4)             *,
(4)             data=None,
(0)         ) -> dict[str, Collection]:
(4)             return gca().violinplot(
(8)                 dataset,
(8)                 positions=positions,
(8)                 vert=vert,
(8)                 widths=widths,
(8)                 showmeans=showmeans,
(8)                 showextrema=showextrema,
(8)                 showmedians=showmedians,
(8)                 quantiles=quantiles,
(8)                 points=points,
(8)                 bw_method=bw_method,
(8)                 side=side,
(8)                 **({"data": data} if data is not None else {}),
(4)             )
(0)         @_copy_docstring_and_deprecators(Axes.vlines)
(0)         def vlines(
(4)             x: float | ArrayLike,
(4)             ymin: float | ArrayLike,
(4)             ymax: float | ArrayLike,
(4)             colors: ColorType | Sequence[ColorType] | None = None,
(4)             linestyles: LineStyleType = "solid",
(4)             label: str = "",
(4)             *,
(4)             data=None,
(4)             **kwargs,
(0)         ) -> LineCollection:
(4)             return gca().vlines(
(8)                 x,
(8)                 ymin,
(8)                 ymax,
(8)                 colors=colors,
(8)                 linestyles=linestyles,
(8)                 label=label,
(8)                 **({"data": data} if data is not None else {}),
(8)                 **kwargs,
(4)             )
(0)         @_copy_docstring_and_deprecators(Axes.xcorr)
(0)         def xcorr(
```

```
3350: (4)               x: ArrayLike,
3351: (4)               y: ArrayLike,
3352: (4)               normed: bool = True,
3353: (4)               detrend: Callable[[ArrayLike], ArrayLike] = mlab.detrend_none,
3354: (4)               usevlines: bool = True,
3355: (4)               maxlags: int = 10,
3356: (4)               *,
3357: (4)               data=None,
3358: (4)               **kwargs,
3359: (0)           ) -> tuple[np.ndarray, np.ndarray, LineCollection | Line2D, Line2D | None]:
3360: (4)               return gca().xcorr(
3361: (8)                   x,
3362: (8)                   y,
3363: (8)                   normed=normed,
3364: (8)                   detrend=detrend,
3365: (8)                   usevlines=usevlines,
3366: (8)                   maxlags=maxlags,
3367: (8)                   **({"data": data} if data is not None else {}),
3368: (8)                   **kwargs,
3369: (4)               )
3370: (0)           @_copy_docstring_and_deprecators(Axes._sci)
3371: (0)           def sci(im: ScalarMappable) -> None:
3372: (4)               gca()._sci(im)
3373: (0)           @_copy_docstring_and_deprecators(Axes.set_title)
3374: (0)           def title(
3375: (4)               label: str,
3376: (4)               fontdict: dict[str, Any] | None = None,
3377: (4)               loc: Literal["left", "center", "right"] | None = None,
3378: (4)               pad: float | None = None,
3379: (4)               *,
3380: (4)               y: float | None = None,
3381: (4)               **kwargs,
3382: (0)           ) -> Text:
3383: (4)               return gca().set_title(label, fontdict=fontdict, loc=loc, pad=pad, y=y,
**kwargs)
3384: (0)           @_copy_docstring_and_deprecators(Axes.set_xlabel)
3385: (0)           def xlabel(
3386: (4)               xlabel: str,
3387: (4)               fontdict: dict[str, Any] | None = None,
3388: (4)               labelpad: float | None = None,
3389: (4)               *,
3390: (4)               loc: Literal["left", "center", "right"] | None = None,
3391: (4)               **kwargs,
3392: (0)           ) -> Text:
3393: (4)               return gca().set_xlabel(
3394: (8)                   xlabel, fontdict=fontdict, labelpad=labelpad, loc=loc, **kwargs
3395: (4)               )
3396: (0)           @_copy_docstring_and_deprecators(Axes.set_ylabel)
3397: (0)           def ylabel(
3398: (4)               ylabel: str,
3399: (4)               fontdict: dict[str, Any] | None = None,
3400: (4)               labelpad: float | None = None,
3401: (4)               *,
3402: (4)               loc: Literal["bottom", "center", "top"] | None = None,
3403: (4)               **kwargs,
3404: (0)           ) -> Text:
3405: (4)               return gca().set_ylabel(
3406: (8)                   ylabel, fontdict=fontdict, labelpad=labelpad, loc=loc, **kwargs
3407: (4)               )
3408: (0)           @_copy_docstring_and_deprecators(Axes.set_xscale)
3409: (0)           def xscale(value: str | ScaleBase, **kwargs) -> None:
3410: (4)               gca().set_xscale(value, **kwargs)
3411: (0)           @_copy_docstring_and_deprecators(Axes.set_yscale)
3412: (0)           def yscale(value: str | ScaleBase, **kwargs) -> None:
3413: (4)               gca().set_yscale(value, **kwargs)
3414: (0)           def autumn() -> None:
3415: (4)               """
3416: (4)               Set the colormap to 'autumn'.
3417: (4)               This changes the default colormap as well as the colormap of the current
```

```
(4)             image if there is one. See ``help(colormaps)`` for more information.
(4)             """
(4)             set_cmap("autumn")
(0)         def bone() -> None:
(4)             """
(4)             Set the colormap to 'bone'.
(4)             This changes the default colormap as well as the colormap of the current
(4)             image if there is one. See ``help(colormaps)`` for more information.
(4)             """
(4)             set_cmap("bone")
(0)         def cool() -> None:
(4)             """
(4)             Set the colormap to 'cool'.
(4)             This changes the default colormap as well as the colormap of the current
(4)             image if there is one. See ``help(colormaps)`` for more information.
(4)             """
(4)             set_cmap("cool")
(0)         def copper() -> None:
(4)             """
(4)             Set the colormap to 'copper'.
(4)             This changes the default colormap as well as the colormap of the current
(4)             image if there is one. See ``help(colormaps)`` for more information.
(4)             """
(4)             set_cmap("copper")
(0)         def flag() -> None:
(4)             """
(4)             Set the colormap to 'flag'.
(4)             This changes the default colormap as well as the colormap of the current
(4)             image if there is one. See ``help(colormaps)`` for more information.
(4)             """
(4)             set_cmap("flag")
(0)         def gray() -> None:
(4)             """
(4)             Set the colormap to 'gray'.
(4)             This changes the default colormap as well as the colormap of the current
(4)             image if there is one. See ``help(colormaps)`` for more information.
(4)             """
(4)             set_cmap("gray")
(0)         def hot() -> None:
(4)             """
(4)             Set the colormap to 'hot'.
(4)             This changes the default colormap as well as the colormap of the current
(4)             image if there is one. See ``help(colormaps)`` for more information.
(4)             """
(4)             set_cmap("hot")
(0)         def hsv() -> None:
(4)             """
(4)             Set the colormap to 'hsv'.
(4)             This changes the default colormap as well as the colormap of the current
(4)             image if there is one. See ``help(colormaps)`` for more information.
(4)             """
(4)             set_cmap("hsv")
(0)         def jet() -> None:
(4)             """
(4)             Set the colormap to 'jet'.
(4)             This changes the default colormap as well as the colormap of the current
(4)             image if there is one. See ``help(colormaps)`` for more information.
(4)             """
(4)             set_cmap("jet")
(0)         def pink() -> None:
(4)             """
(4)             Set the colormap to 'pink'.
(4)             This changes the default colormap as well as the colormap of the current
(4)             image if there is one. See ``help(colormaps)`` for more information.
(4)             """
(4)             set_cmap("pink")
(0)         def prism() -> None:
(4)             """
(4)             Set the colormap to 'prism'.
```

```
3487: (4)               This changes the default colormap as well as the colormap of the current
3488: (4)               image if there is one. See ``help(colormaps)`` for more information.
3489: (4)               """
3490: (4)               set_cmap("prism")
3491: (0)           def spring() -> None:
3492: (4)               """
3493: (4)               Set the colormap to 'spring'.
3494: (4)               This changes the default colormap as well as the colormap of the current
3495: (4)               image if there is one. See ``help(colormaps)`` for more information.
3496: (4)               """
3497: (4)               set_cmap("spring")
3498: (0)           def summer() -> None:
3499: (4)               """
3500: (4)               Set the colormap to 'summer'.
3501: (4)               This changes the default colormap as well as the colormap of the current
3502: (4)               image if there is one. See ``help(colormaps)`` for more information.
3503: (4)               """
3504: (4)               set_cmap("summer")
3505: (0)           def winter() -> None:
3506: (4)               """
3507: (4)               Set the colormap to 'winter'.
3508: (4)               This changes the default colormap as well as the colormap of the current
3509: (4)               image if there is one. See ``help(colormaps)`` for more information.
3510: (4)               """
3511: (4)               set_cmap("winter")
3512: (0)           def magma() -> None:
3513: (4)               """
3514: (4)               Set the colormap to 'magma'.
3515: (4)               This changes the default colormap as well as the colormap of the current
3516: (4)               image if there is one. See ``help(colormaps)`` for more information.
3517: (4)               """
3518: (4)               set_cmap("magma")
3519: (0)           def inferno() -> None:
3520: (4)               """
3521: (4)               Set the colormap to 'inferno'.
3522: (4)               This changes the default colormap as well as the colormap of the current
3523: (4)               image if there is one. See ``help(colormaps)`` for more information.
3524: (4)               """
3525: (4)               set_cmap("inferno")
3526: (0)           def plasma() -> None:
3527: (4)               """
3528: (4)               Set the colormap to 'plasma'.
3529: (4)               This changes the default colormap as well as the colormap of the current
3530: (4)               image if there is one. See ``help(colormaps)`` for more information.
3531: (4)               """
3532: (4)               set_cmap("plasma")
3533: (0)           def viridis() -> None:
3534: (4)               """
3535: (4)               Set the colormap to 'viridis'.
3536: (4)               This changes the default colormap as well as the colormap of the current
3537: (4)               image if there is one. See ``help(colormaps)`` for more information.
3538: (4)               """
3539: (4)               set_cmap("viridis")
3540: (0)           def nipy_spectral() -> None:
3541: (4)               """
3542: (4)               Set the colormap to 'nipy_spectral'.
3543: (4)               This changes the default colormap as well as the colormap of the current
3544: (4)               image if there is one. See ``help(colormaps)`` for more information.
3545: (4)               """
3546: (4)               set_cmap("nipy_spectral")

----------------------------------------

File 56 - quiver.py:

1: (0)             """
2: (0)             Support for plotting vector fields.
3: (0)             Presently this contains Quiver and Barb. Quiver plots an arrow in the
4: (0)             direction of the vector, with the size of the arrow related to the
```

```
 5: (0)             magnitude of the vector.
 6: (0)             Barbs are like quiver in that they point along a vector, but
 7: (0)             the magnitude of the vector is given schematically by the presence of barbs
 8: (0)             or flags on the barb.
 9: (0)             This will also become a home for things such as standard
10: (0)             deviation ellipses, which can and will be derived very easily from
11: (0)             the Quiver code.
12: (0)             """
13: (0)             import math
14: (0)             import numpy as np
15: (0)             from numpy import ma
16: (0)             from matplotlib import _api, cbook, _docstring
17: (0)             import matplotlib.artist as martist
18: (0)             import matplotlib.collections as mcollections
19: (0)             from matplotlib.patches import CirclePolygon
20: (0)             import matplotlib.text as mtext
21: (0)             import matplotlib.transforms as transforms
22: (0)             _quiver_doc = """
23: (0)             Plot a 2D field of arrows.
24: (0)             Call signature::
25: (2)               quiver([X, Y], U, V, [C], **kwargs)
26: (0)             *X*, *Y* define the arrow locations, *U*, *V* define the arrow directions, and
27: (0)             *C* optionally sets the color.
28: (0)             **Arrow length**
29: (0)             The default settings auto-scales the length of the arrows to a reasonable
size.
30: (0)             To change this behavior see the *scale* and *scale_units* parameters.
31: (0)             **Arrow shape**
32: (0)             The arrow shape is determined by *width*, *headwidth*, *headlength* and
33: (0)             *headaxislength*. See the notes below.
34: (0)             **Arrow styling**
35: (0)             Each arrow is internally represented by a filled polygon with a default edge
36: (0)             linewidth of 0. As a result, an arrow is rather a filled area, not a line with
37: (0)             a head, and `.PolyCollection` properties like *linewidth*, *edgecolor*,
38: (0)             *facecolor*, etc. act accordingly.
39: (0)             Parameters
40: (0)             ----------
41: (0)             X, Y : 1D or 2D array-like, optional
42: (4)                 The x and y coordinates of the arrow locations.
43: (4)                 If not given, they will be generated as a uniform integer meshgrid based
44: (4)                 on the dimensions of *U* and *V*.
45: (4)                 If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D
46: (4)                 using ``X, Y = np.meshgrid(X, Y)``. In this case ``len(X)`` and ``len(Y)``
47: (4)                 must match the column and row dimensions of *U* and *V*.
48: (0)             U, V : 1D or 2D array-like
49: (4)                 The x and y direction components of the arrow vectors. The interpretation
50: (4)                 of these components (in data or in screen space) depends on *angles*.
51: (4)                 *U* and *V* must have the same number of elements, matching the number of
52: (4)                 arrow locations in *X*, *Y*. *U* and *V* may be masked. Locations masked
53: (4)                 in any of *U*, *V*, and *C* will not be drawn.
54: (0)             C : 1D or 2D array-like, optional
55: (4)                 Numeric data that defines the arrow colors by colormapping via *norm* and
56: (4)                 *cmap*.
57: (4)                 This does not support explicit colors. If you want to set colors directly,
58: (4)                 use *color* instead.  The size of *C* must match the number of arrow
59: (4)                 locations.
60: (0)             angles : {'uv', 'xy'} or array-like, default: 'uv'
61: (4)                 Method for determining the angle of the arrows.
62: (4)                 - 'uv': Arrow direction in screen coordinates. Use this if the arrows
63: (6)                   symbolize a quantity that is not based on *X*, *Y* data coordinates.
64: (6)                   If *U* == *V* the orientation of the arrow on the plot is 45 degrees
65: (6)                   counter-clockwise from the horizontal axis (positive to the right).
66: (4)                 - 'xy': Arrow direction in data coordinates, i.e. the arrows point from
67: (6)                   (x, y) to (x+u, y+v). Use this e.g. for plotting a gradient field.
68: (4)                 - Arbitrary angles may be specified explicitly as an array of values
69: (6)                   in degrees, counter-clockwise from the horizontal axis.
70: (6)                   In this case *U*, *V* is only used to determine the length of the
71: (6)                   arrows.
72: (4)                 Note: inverting a data axis will correspondingly invert the
```

```
 73: (4)                arrows only with ``angles='xy'``.
 74: (0)            pivot : {'tail', 'mid', 'middle', 'tip'}, default: 'tail'
 75: (4)                The part of the arrow that is anchored to the *X*, *Y* grid. The arrow
 76: (4)                rotates about this point.
 77: (4)                'mid' is a synonym for 'middle'.
 78: (0)            scale : float, optional
 79: (4)                Scales the length of the arrow inversely.
 80: (4)                Number of data units per arrow length unit, e.g., m/s per plot width; a
 81: (4)                smaller scale parameter makes the arrow longer. Default is *None*.
 82: (4)                If *None*, a simple autoscaling algorithm is used, based on the average
 83: (4)                vector length and the number of vectors. The arrow length unit is given by
 84: (4)                the *scale_units* parameter.
 85: (0)            scale_units : {'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, optional
 86: (4)                If the *scale* kwarg is *None*, the arrow length unit. Default is *None*.
 87: (4)                e.g. *scale_units* is 'inches', *scale* is 2.0, and ``(u, v) = (1, 0)``,
 88: (4)                then the vector will be 0.5 inches long.
 89: (4)                If *scale_units* is 'width' or 'height', then the vector will be half the
 90: (4)                width/height of the axes.
 91: (4)                If *scale_units* is 'x' then the vector will be 0.5 x-axis
 92: (4)                units. To plot vectors in the x-y plane, with u and v having
 93: (4)                the same units as x and y, use
 94: (4)                ``angles='xy', scale_units='xy', scale=1``.
 95: (0)            units : {'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, default:
'width'
 96: (4)
 97: (4)                Affects the arrow size (except for the length). In particular, the shaft
 98: (4)                *width* is measured in multiples of this unit.
 99: (4)                Supported values are:
100: (4)                - 'width', 'height': The width or height of the Axes.
101: (4)                - 'dots', 'inches': Pixels or inches based on the figure dpi.
102: (4)                - 'x', 'y', 'xy': *X*, *Y* or :math:`\\sqrt{X^2 + Y^2}` in data units.
103: (4)                The following table summarizes how these values affect the visible arrow
104: (4)                size under zooming and figure size changes:
105: (4)                =================  =================   ==================
106: (4)                units              zoom                figure size change
107: (4)                =================  =================   ==================
108: (4)                'x', 'y', 'xy'     arrow size scales   —
109: (4)                'width', 'height'  —                   arrow size scales
110: (4)                'dots', 'inches'   —                   —
111: (4)                =================  =================   ==================
112: (0)            width : float, optional
113: (4)                Shaft width in arrow units. All head parameters are relative to *width*.
114: (4)                The default depends on choice of *units* above, and number of vectors;
115: (4)                a typical starting value is about 0.005 times the width of the plot.
116: (0)            headwidth : float, default: 3
117: (4)                Head width as multiple of shaft *width*. See the notes below.
118: (0)            headlength : float, default: 5
119: (4)                Head length as multiple of shaft *width*. See the notes below.
120: (0)            headaxislength : float, default: 4.5
121: (4)                Head length at shaft intersection as multiple of shaft *width*.
122: (4)                See the notes below.
123: (0)            minshaft : float, default: 1
124: (4)                Length below which arrow scales, in units of head length. Do not
125: (4)                set this to less than 1, or small arrows will look terrible!
126: (0)            minlength : float, default: 1
127: (4)                Minimum length as a multiple of shaft width; if an arrow length
128: (4)                is less than this, plot a dot (hexagon) of this diameter instead.
129: (0)            color : :mpltype:`color` or list :mpltype:`color`, optional
130: (4)                Explicit color(s) for the arrows. If *C* has been set, *color* has no
131: (4)                effect.
132: (4)                This is a synonym for the `.PolyCollection` *facecolor* parameter.
133: (0)            Other Parameters
134: (0)            ----------------
135: (0)            data : indexable object, optional
136: (4)                DATA_PARAMETER_PLACEHOLDER
137: (0)            **kwargs : `~matplotlib.collections.PolyCollection` properties, optional
138: (4)                All other keyword arguments are passed on to `.PolyCollection`:
139: (4)                %(PolyCollection:kwdoc)s
140: (0)            Returns
141: (0)            -------
```

```
(0)            `~matplotlib.quiver.Quiver`
(0)            See Also
(0)            --------
(0)            .Axes.quiverkey : Add a key to a quiver plot.
(0)            Notes
(0)            -----
(0)            **Arrow shape**
(0)            The arrow is drawn as a polygon using the nodes as shown below. The values
(0)            *headwidth*, *headlength*, and *headaxislength* are in units of *width*.
(0)            .. image:: /_static/quiver_sizes.svg
(3)               :width: 500px
(0)            The defaults give a slightly swept-back arrow. Here are some guidelines how to
(0)            get other head shapes:
(0)            - To make the head a triangle, make *headaxislength* the same as *headlength*.
(0)            - To make the arrow more pointed, reduce *headwidth* or increase *headlength*
(2)              and *headaxislength*.
(0)            - To make the head smaller relative to the shaft, scale down all the head
(2)              parameters proportionally.
(0)            - To remove the head completely, set all *head* parameters to 0.
(0)            - To get a diamond-shaped head, make *headaxislength* larger than
*headlength*.
(0)            - Warning: For *headaxislength* < (*headlength* / *headwidth*), the "headaxis"
(2)              nodes (i.e. the ones connecting the head with the shaft) will protrude out
(2)              of the head in forward direction so that the arrow head looks broken.
(0)            """ % _docstring.interpd.params
(0)            _docstring.interpd.update(quiver_doc=_quiver_doc)
(0)            class QuiverKey(martist.Artist):
(4)                """Labelled arrow for use as a quiver plot scale key."""
(4)                halign = {'N': 'center', 'S': 'center', 'E': 'left', 'W': 'right'}
(4)                valign = {'N': 'bottom', 'S': 'top', 'E': 'center', 'W': 'center'}
(4)                pivot = {'N': 'middle', 'S': 'middle', 'E': 'tip', 'W': 'tail'}
(4)                def __init__(self, Q, X, Y, U, label,
(17)                            *, angle=0, coordinates='axes', color=None, labelsep=0.1,
(17)                            labelpos='N', labelcolor=None, fontproperties=None,
**kwargs):
(8)                    """
(8)                    Add a key to a quiver plot.
(8)                    The positioning of the key depends on *X*, *Y*, *coordinates*, and
(8)                    *labelpos*.  If *labelpos* is 'N' or 'S', *X*, *Y* give the position
of
(8)                    the middle of the key arrow.  If *labelpos* is 'E', *X*, *Y* positions
(8)                    the head, and if *labelpos* is 'W', *X*, *Y* positions the tail; in
(8)                    either of these two cases, *X*, *Y* is somewhere in the middle of the
(8)                    arrow+label key object.
(8)                    Parameters
(8)                    ----------
(8)                    Q : `~matplotlib.quiver.Quiver`
(12)                       A `.Quiver` object as returned by a call to `~.Axes.quiver()`.
(8)                    X, Y : float
(12)                       The location of the key.
(8)                    U : float
(12)                       The length of the key.
(8)                    label : str
(12)                       The key label (e.g., length and units of the key).
(8)                    angle : float, default: 0
(12)                       The angle of the key arrow, in degrees anti-clockwise from the
(12)                       horizontal axis.
(8)                    coordinates : {'axes', 'figure', 'data', 'inches'}, default: 'axes'
(12)                       Coordinate system and units for *X*, *Y*: 'axes' and 'figure' are
(12)                       normalized coordinate systems with (0, 0) in the lower left and
(12)                       (1, 1) in the upper right; 'data' are the axes data coordinates
(12)                       (used for the locations of the vectors in the quiver plot itself);
(12)                       'inches' is position in the figure in inches, with (0, 0) at the
(12)                       lower left corner.
(8)                    color : :mpltype:`color`
(12)                       Overrides face and edge colors from *Q*.
(8)                    labelpos : {'N', 'S', 'E', 'W'}
(12)                       Position the label above, below, to the right, to the left of the
(12)                       arrow, respectively.
```

```
(8)                     labelsep : float, default: 0.1
(12)                        Distance in inches between the arrow and the label.
(8)                     labelcolor : :mpltype:`color`, default: :rc:`text.color`
(12)                        Label color.
(8)                     fontproperties : dict, optional
(12)                        A dictionary with keyword arguments accepted by the
(12)                        `~matplotlib.font_manager.FontProperties` initializer:
(12)                        *family*, *style*, *variant*, *size*, *weight*.
(8)                     **kwargs
(12)                        Any additional keyword arguments are used to override vector
(12)                        properties taken from *Q*.
(8)                     """
(8)                     super().__init__()
(8)                     self.Q = Q
(8)                     self.X = X
(8)                     self.Y = Y
(8)                     self.U = U
(8)                     self.angle = angle
(8)                     self.coord = coordinates
(8)                     self.color = color
(8)                     self.label = label
(8)                     self._labelsep_inches = labelsep
(8)                     self.labelpos = labelpos
(8)                     self.labelcolor = labelcolor
(8)                     self.fontproperties = fontproperties or dict()
(8)                     self.kw = kwargs
(8)                     self.text = mtext.Text(
(12)                        text=label,
(12)                        horizontalalignment=self.halign[self.labelpos],
(12)                        verticalalignment=self.valign[self.labelpos],
(12)                        fontproperties=self.fontproperties)
(8)                     if self.labelcolor is not None:
(12)                        self.text.set_color(self.labelcolor)
(8)                     self._dpi_at_last_init = None
(8)                     self.zorder = Q.zorder + 0.1
(4)                 @property
(4)                 def labelsep(self):
(8)                     return self._labelsep_inches * self.Q.axes.figure.dpi
(4)                 def _init(self):
(8)                     if True:  # self._dpi_at_last_init != self.axes.figure.dpi
(12)                        if self.Q._dpi_at_last_init != self.Q.axes.figure.dpi:
(16)                            self.Q._init()
(12)                        self._set_transform()
(12)                        with cbook._setattr_cm(self.Q, pivot=self.pivot[self.labelpos],
(35)                                               Umask=ma.nomask):
(16)                            u = self.U * np.cos(np.radians(self.angle))
(16)                            v = self.U * np.sin(np.radians(self.angle))
(16)                            self.verts = self.Q._make_verts([[0., 0.]],
(48)                                                            np.array([u]), np.array([v]),
'uv')
(12)                        kwargs = self.Q.polykw
(12)                        kwargs.update(self.kw)
(12)                        self.vector = mcollections.PolyCollection(
(16)                            self.verts,
(16)                            offsets=[(self.X, self.Y)],
(16)                            offset_transform=self.get_transform(),
(16)                            **kwargs)
(12)                        if self.color is not None:
(16)                            self.vector.set_color(self.color)
(12)                        self.vector.set_transform(self.Q.get_transform())
(12)                        self.vector.set_figure(self.get_figure())
(12)                        self._dpi_at_last_init = self.Q.axes.figure.dpi
(4)                 def _text_shift(self):
(8)                     return {
(12)                        "N": (0, +self.labelsep),
(12)                        "S": (0, -self.labelsep),
(12)                        "E": (+self.labelsep, 0),
(12)                        "W": (-self.labelsep, 0),
(8)                     }[self.labelpos]
```

```
(4)                 @martist.allow_rasterization
(4)                 def draw(self, renderer):
(8)                     self._init()
(8)                     self.vector.draw(renderer)
(8)                     pos = self.get_transform().transform((self.X, self.Y))
(8)                     self.text.set_position(pos + self._text_shift())
(8)                     self.text.draw(renderer)
(8)                     self.stale = False
(4)                 def _set_transform(self):
(8)                     self.set_transform(_api.check_getitem({
(12)                        "data": self.Q.axes.transData,
(12)                        "axes": self.Q.axes.transAxes,
(12)                        "figure": self.Q.axes.figure.transFigure,
(12)                        "inches": self.Q.axes.figure.dpi_scale_trans,
(8)                     }, coordinates=self.coord))
(4)                 def set_figure(self, fig):
(8)                     super().set_figure(fig)
(8)                     self.text.set_figure(fig)
(4)                 def contains(self, mouseevent):
(8)                     if self._different_canvas(mouseevent):
(12)                        return False, {}
(8)                     if (self.text.contains(mouseevent)[0] or
(16)                            self.vector.contains(mouseevent)[0]):
(12)                        return True, {}
(8)                     return False, {}
(0)             def _parse_args(*args, caller_name='function'):
(4)                 """
(4)                 Helper function to parse positional parameters for colored vector plots.
(4)                 This is currently used for Quiver and Barbs.
(4)                 Parameters
(4)                 ----------
(4)                 *args : list
(8)                     list of 2-5 arguments. Depending on their number they are parsed to::
(12)                        U, V
(12)                        U, V, C
(12)                        X, Y, U, V
(12)                        X, Y, U, V, C
(4)                 caller_name : str
(8)                     Name of the calling method (used in error messages).
(4)                 """
(4)                 X = Y = C = None
(4)                 nargs = len(args)
(4)                 if nargs == 2:
(8)                     U, V = np.atleast_1d(*args)
(4)                 elif nargs == 3:
(8)                     U, V, C = np.atleast_1d(*args)
(4)                 elif nargs == 4:
(8)                     X, Y, U, V = np.atleast_1d(*args)
(4)                 elif nargs == 5:
(8)                     X, Y, U, V, C = np.atleast_1d(*args)
(4)                 else:
(8)                     raise _api.nargs_error(caller_name, takes="from 2 to 5", given=nargs)
(4)                 nr, nc = (1, U.shape[0]) if U.ndim == 1 else U.shape
(4)                 if X is not None:
(8)                     X = X.ravel()
(8)                     Y = Y.ravel()
(8)                     if len(X) == nc and len(Y) == nr:
(12)                        X, Y = [a.ravel() for a in np.meshgrid(X, Y)]
(8)                     elif len(X) != len(Y):
(12)                        raise ValueError('X and Y must be the same size, but '
(29)                                         f'X.size is {X.size} and Y.size is {Y.size}.')
(4)                 else:
(8)                     indexgrid = np.meshgrid(np.arange(nc), np.arange(nr))
(8)                     X, Y = [np.ravel(a) for a in indexgrid]
(4)                 return X, Y, U, V, C
(0)             def _check_consistent_shapes(*arrays):
(4)                 all_shapes = {a.shape for a in arrays}
(4)                 if len(all_shapes) != 1:
(8)                     raise ValueError('The shapes of the passed in arrays do not match')
```

```
(0)            class Quiver(mcollections.PolyCollection):
(4)                """
(4)                Specialized PolyCollection for arrows.
(4)                The only API method is set_UVC(), which can be used
(4)                to change the size, orientation, and color of the
(4)                arrows; their locations are fixed when the class is
(4)                instantiated.  Possibly this method will be useful
(4)                in animations.
(4)                Much of the work in this class is done in the draw()
(4)                method so that as much information as possible is available
(4)                about the plot.  In subsequent draw() calls, recalculation
(4)                is limited to things that might have changed, so there
(4)                should be no performance penalty from putting the calculations
(4)                in the draw() method.
(4)                """
(4)                _PIVOT_VALS = ('tail', 'middle', 'tip')
(4)                @_docstring.Substitution(_quiver_doc)
(4)                def __init__(self, ax, *args,
(17)                            scale=None, headwidth=3, headlength=5, headaxislength=4.5,
(17)                            minshaft=1, minlength=1, units='width', scale_units=None,
(17)                            angles='uv', width=None, color='k', pivot='tail', **kwargs):
(8)                    """
(8)                    The constructor takes one required argument, an Axes
(8)                    instance, followed by the args and kwargs described
(8)                    by the following pyplot interface documentation:
(8)                    %s
(8)                    """
(8)                    self._axes = ax  # The attr actually set by the Artist.axes property.
(8)                    X, Y, U, V, C = _parse_args(*args, caller_name='quiver')
(8)                    self.X = X
(8)                    self.Y = Y
(8)                    self.XY = np.column_stack((X, Y))
(8)                    self.N = len(X)
(8)                    self.scale = scale
(8)                    self.headwidth = headwidth
(8)                    self.headlength = float(headlength)
(8)                    self.headaxislength = headaxislength
(8)                    self.minshaft = minshaft
(8)                    self.minlength = minlength
(8)                    self.units = units
(8)                    self.scale_units = scale_units
(8)                    self.angles = angles
(8)                    self.width = width
(8)                    if pivot.lower() == 'mid':
(12)                       pivot = 'middle'
(8)                    self.pivot = pivot.lower()
(8)                    _api.check_in_list(self._PIVOT_VALS, pivot=self.pivot)
(8)                    self.transform = kwargs.pop('transform', ax.transData)
(8)                    kwargs.setdefault('facecolors', color)
(8)                    kwargs.setdefault('linewidths', (0,))
(8)                    super().__init__([], offsets=self.XY, offset_transform=self.transform,
(25)                                    closed=False, **kwargs)
(8)                    self.polykw = kwargs
(8)                    self.set_UVC(U, V, C)
(8)                    self._dpi_at_last_init = None
(4)                def _init(self):
(8)                    """
(8)                    Initialization delayed until first draw;
(8)                    allow time for axes setup.
(8)                    """
(8)                    if True:  # self._dpi_at_last_init != self.axes.figure.dpi
(12)                       trans = self._set_transform()
(12)                       self.span = trans.inverted().transform_bbox(self.axes.bbox).width
(12)                       if self.width is None:
(16)                           sn = np.clip(math.sqrt(self.N), 8, 25)
(16)                           self.width = 0.06 * self.span / sn
(12)                       if (self._dpi_at_last_init != self.axes.figure.dpi
(20)                               and self.scale is None):
(16)                           self._make_verts(self.XY, self.U, self.V, self.angles)
```

```
(12)                        self._dpi_at_last_init = self.axes.figure.dpi
(4)                 def get_datalim(self, transData):
(8)                     trans = self.get_transform()
(8)                     offset_trf = self.get_offset_transform()
(8)                     full_transform = (trans - transData) + (offset_trf - transData)
(8)                     XY = full_transform.transform(self.XY)
(8)                     bbox = transforms.Bbox.null()
(8)                     bbox.update_from_data_xy(XY, ignore=True)
(8)                     return bbox
(4)                 @martist.allow_rasterization
(4)                 def draw(self, renderer):
(8)                     self._init()
(8)                     verts = self._make_verts(self.XY, self.U, self.V, self.angles)
(8)                     self.set_verts(verts, closed=False)
(8)                     super().draw(renderer)
(8)                     self.stale = False
(4)                 def set_UVC(self, U, V, C=None):
(8)                     U = ma.masked_invalid(U, copy=True).ravel()
(8)                     V = ma.masked_invalid(V, copy=True).ravel()
(8)                     if C is not None:
(12)                        C = ma.masked_invalid(C, copy=True).ravel()
(8)                     for name, var in zip(('U', 'V', 'C'), (U, V, C)):
(12)                        if not (var is None or var.size == self.N or var.size == 1):
(16)                            raise ValueError(f'Argument {name} has a size {var.size}'
(33)                                             f' which does not match {self.N},'
(33)                                             ' the number of arrow positions')
(8)                     mask = ma.mask_or(U.mask, V.mask, copy=False, shrink=True)
(8)                     if C is not None:
(12)                        mask = ma.mask_or(mask, C.mask, copy=False, shrink=True)
(12)                        if mask is ma.nomask:
(16)                            C = C.filled()
(12)                        else:
(16)                            C = ma.array(C, mask=mask, copy=False)
(8)                     self.U = U.filled(1)
(8)                     self.V = V.filled(1)
(8)                     self.Umask = mask
(8)                     if C is not None:
(12)                        self.set_array(C)
(8)                     self.stale = True
(4)                 def _dots_per_unit(self, units):
(8)                     """Return a scale factor for converting from units to pixels."""
(8)                     bb = self.axes.bbox
(8)                     vl = self.axes.viewLim
(8)                     return _api.check_getitem({
(12)                        'x': bb.width / vl.width,
(12)                        'y': bb.height / vl.height,
(12)                        'xy': np.hypot(*bb.size) / np.hypot(*vl.size),
(12)                        'width': bb.width,
(12)                        'height': bb.height,
(12)                        'dots': 1.,
(12)                        'inches': self.axes.figure.dpi,
(8)                     }, units=units)
(4)                 def _set_transform(self):
(8)                     """
(8)                     Set the PolyCollection transform to go
(8)                     from arrow width units to pixels.
(8)                     """
(8)                     dx = self._dots_per_unit(self.units)
(8)                     self._trans_scale = dx  # pixels per arrow width unit
(8)                     trans = transforms.Affine2D().scale(dx)
(8)                     self.set_transform(trans)
(8)                     return trans
(4)                 def _angles_lengths(self, XY, U, V, eps=1):
(8)                     xy = self.axes.transData.transform(XY)
(8)                     uv = np.column_stack((U, V))
(8)                     xyp = self.axes.transData.transform(XY + eps * uv)
(8)                     dxy = xyp - xy
(8)                     angles = np.arctan2(dxy[:, 1], dxy[:, 0])
(8)                     lengths = np.hypot(*dxy.T) / eps
```

```
482: (8)                    return angles, lengths
483: (4)                def _make_verts(self, XY, U, V, angles):
484: (8)                    uv = (U + V * 1j)
485: (8)                    str_angles = angles if isinstance(angles, str) else ''
486: (8)                    if str_angles == 'xy' and self.scale_units == 'xy':
487: (12)                       angles, lengths = self._angles_lengths(XY, U, V, eps=1)
488: (8)                    elif str_angles == 'xy' or self.scale_units == 'xy':
489: (12)                       eps = np.abs(self.axes.dataLim.extents).max() * 0.001
490: (12)                       angles, lengths = self._angles_lengths(XY, U, V, eps=eps)
491: (8)                    if str_angles and self.scale_units == 'xy':
492: (12)                       a = lengths
493: (8)                    else:
494: (12)                       a = np.abs(uv)
495: (8)                    if self.scale is None:
496: (12)                       sn = max(10, math.sqrt(self.N))
497: (12)                       if self.Umask is not ma.nomask:
498: (16)                           amean = a[~self.Umask].mean()
499: (12)                       else:
500: (16)                           amean = a.mean()
501: (12)                       scale = 1.8 * amean * sn / self.span
502: (8)                    if self.scale_units is None:
503: (12)                       if self.scale is None:
504: (16)                           self.scale = scale
505: (12)                       widthu_per_lenu = 1.0
506: (8)                    else:
507: (12)                       if self.scale_units == 'xy':
508: (16)                           dx = 1
509: (12)                       else:
510: (16)                           dx = self._dots_per_unit(self.scale_units)
511: (12)                       widthu_per_lenu = dx / self._trans_scale
512: (12)                       if self.scale is None:
513: (16)                           self.scale = scale * widthu_per_lenu
514: (8)                    length = a * (widthu_per_lenu / (self.scale * self.width))
515: (8)                    X, Y = self._h_arrows(length)
516: (8)                    if str_angles == 'xy':
517: (12)                       theta = angles
518: (8)                    elif str_angles == 'uv':
519: (12)                       theta = np.angle(uv)
520: (8)                    else:
521: (12)                       theta = ma.masked_invalid(np.deg2rad(angles)).filled(0)
522: (8)                    theta = theta.reshape((-1, 1))  # for broadcasting
523: (8)                    xy = (X + Y * 1j) * np.exp(1j * theta) * self.width
524: (8)                    XY = np.stack((xy.real, xy.imag), axis=2)
525: (8)                    if self.Umask is not ma.nomask:
526: (12)                       XY = ma.array(XY)
527: (12)                       XY[self.Umask] = ma.masked
528: (8)                    return XY
529: (4)                def _h_arrows(self, length):
530: (8)                    """Length is in arrow width units."""
531: (8)                    minsh = self.minshaft * self.headlength
532: (8)                    N = len(length)
533: (8)                    length = length.reshape(N, 1)
534: (8)                    np.clip(length, 0, 2 ** 16, out=length)
535: (8)                    x = np.array([0, -self.headaxislength,
536: (22)                                 -self.headlength, 0],
537: (21)                                np.float64)
538: (8)                    x = x + np.array([0, 1, 1, 1]) * length
539: (8)                    y = 0.5 * np.array([1, 1, self.headwidth, 0], np.float64)
540: (8)                    y = np.repeat(y[np.newaxis, :], N, axis=0)
541: (8)                    x0 = np.array([0, minsh - self.headaxislength,
542: (23)                                  minsh - self.headlength, minsh], np.float64)
543: (8)                    y0 = 0.5 * np.array([1, 1, self.headwidth, 0], np.float64)
544: (8)                    ii = [0, 1, 2, 3, 2, 1, 0, 0]
545: (8)                    X = x[:, ii]
546: (8)                    Y = y[:, ii]
547: (8)                    Y[:, 3:-1] *= -1
548: (8)                    X0 = x0[ii]
549: (8)                    Y0 = y0[ii]
550: (8)                    Y0[3:-1] *= -1
```

```
(8)                 shrink = length / minsh if minsh != 0. else 0.
(8)                 X0 = shrink * X0[np.newaxis, :]
(8)                 Y0 = shrink * Y0[np.newaxis, :]
(8)                 short = np.repeat(length < minsh, 8, axis=1)
(8)                 np.copyto(X, X0, where=short)
(8)                 np.copyto(Y, Y0, where=short)
(8)                 if self.pivot == 'middle':
(12)                    X -= 0.5 * X[:, 3, np.newaxis]
(8)                 elif self.pivot == 'tip':
(12)                    X = X - X[:, 3, np.newaxis]
(8)                 elif self.pivot != 'tail':
(12)                    _api.check_in_list(["middle", "tip", "tail"], pivot=self.pivot)
(8)                 tooshort = length < self.minlength
(8)                 if tooshort.any():
(12)                    th = np.arange(0, 8, 1, np.float64) * (np.pi / 3.0)
(12)                    x1 = np.cos(th) * self.minlength * 0.5
(12)                    y1 = np.sin(th) * self.minlength * 0.5
(12)                    X1 = np.repeat(x1[np.newaxis, :], N, axis=0)
(12)                    Y1 = np.repeat(y1[np.newaxis, :], N, axis=0)
(12)                    tooshort = np.repeat(tooshort, 8, 1)
(12)                    np.copyto(X, X1, where=tooshort)
(12)                    np.copyto(Y, Y1, where=tooshort)
(8)                 return X, Y
(0)         _barbs_doc = r"""
(0)         Plot a 2D field of wind barbs.
(0)         Call signature::
(2)           barbs([X, Y], U, V, [C], **kwargs)
(0)         Where *X*, *Y* define the barb locations, *U*, *V* define the barb
(0)         directions, and *C* optionally sets the color.
(0)         All arguments may be 1D or 2D. *U*, *V*, *C* may be masked arrays, but masked
(0)         *X*, *Y* are not supported at present.
(0)         Barbs are traditionally used in meteorology as a way to plot the speed
(0)         and direction of wind observations, but can technically be used to
(0)         plot any two dimensional vector quantity.  As opposed to arrows, which
(0)         give vector magnitude by the length of the arrow, the barbs give more
(0)         quantitative information about the vector magnitude by putting slanted
(0)         lines or a triangle for various increments in magnitude, as show
(0)         schematically below::
(2)           :                   /\    \
(2)           :                  /  \    \
(2)           :                 /    \    \    \
(2)           :                /      \    \    \
(2)           :               ------------------------------
(0)         The largest increment is given by a triangle (or "flag"). After those
(0)         come full lines (barbs). The smallest increment is a half line.  There
(0)         is only, of course, ever at most 1 half line.  If the magnitude is
(0)         small and only needs a single half-line and no full lines or
(0)         triangles, the half-line is offset from the end of the barb so that it
(0)         can be easily distinguished from barbs with a single full line.  The
(0)         magnitude for the barb shown above would nominally be 65, using the
(0)         standard increments of 50, 10, and 5.
(0)         See also https://en.wikipedia.org/wiki/Wind_barb.
(0)         Parameters
(0)         ----------
(0)         X, Y : 1D or 2D array-like, optional
(4)             The x and y coordinates of the barb locations. See *pivot* for how the
(4)             barbs are drawn to the x, y positions.
(4)             If not given, they will be generated as a uniform integer meshgrid based
(4)             on the dimensions of *U* and *V*.
(4)             If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D
(4)             using ``X, Y = np.meshgrid(X, Y)``. In this case ``len(X)`` and ``len(Y)``
(4)             must match the column and row dimensions of *U* and *V*.
(0)         U, V : 1D or 2D array-like
(4)             The x and y components of the barb shaft.
(0)         C : 1D or 2D array-like, optional
(4)             Numeric data that defines the barb colors by colormapping via *norm* and
(4)             *cmap*.
(4)             This does not support explicit colors. If you want to set colors directly,
(4)             use *barbcolor* instead.
```

```
(0)             length : float, default: 7
(4)                 Length of the barb in points; the other parts of the barb
(4)                 are scaled against this.
(0)             pivot : {'tip', 'middle'} or float, default: 'tip'
(4)                 The part of the arrow that is anchored to the *X*, *Y* grid. The barb
(4)                 rotates about this point. This can also be a number, which shifts the
(4)                 start of the barb that many points away from grid point.
(0)             barbcolor : :mpltype:`color` or color sequence
(4)                 The color of all parts of the barb except for the flags.  This parameter
(4)                 is analogous to the *edgecolor* parameter for polygons, which can be used
(4)                 instead. However this parameter will override facecolor.
(0)             flagcolor : :mpltype:`color` or color sequence
(4)                 The color of any flags on the barb.  This parameter is analogous to the
(4)                 *facecolor* parameter for polygons, which can be used instead. However,
(4)                 this parameter will override facecolor.  If this is not set (and *C* has
(4)                 not either) then *flagcolor* will be set to match *barbcolor* so that the
(4)                 barb has a uniform color. If *C* has been set, *flagcolor* has no effect.
(0)             sizes : dict, optional
(4)                 A dictionary of coefficients specifying the ratio of a given
(4)                 feature to the length of the barb. Only those values one wishes to
(4)                 override need to be included.  These features include:
(4)                 - 'spacing' - space between features (flags, full/half barbs)
(4)                 - 'height' - height (distance from shaft to top) of a flag or full barb
(4)                 - 'width' - width of a flag, twice the width of a full barb
(4)                 - 'emptybarb' - radius of the circle used for low magnitudes
(0)             fill_empty : bool, default: False
(4)                 Whether the empty barbs (circles) that are drawn should be filled with
(4)                 the flag color.  If they are not filled, the center is transparent.
(0)             rounding : bool, default: True
(4)                 Whether the vector magnitude should be rounded when allocating barb
(4)                 components.  If True, the magnitude is rounded to the nearest multiple
(4)                 of the half-barb increment.  If False, the magnitude is simply truncated
(4)                 to the next lowest multiple.
(0)             barb_increments : dict, optional
(4)                 A dictionary of increments specifying values to associate with
(4)                 different parts of the barb. Only those values one wishes to
(4)                 override need to be included.
(4)                 - 'half' - half barbs (Default is 5)
(4)                 - 'full' - full barbs (Default is 10)
(4)                 - 'flag' - flags (default is 50)
(0)             flip_barb : bool or array-like of bool, default: False
(4)                 Whether the lines and flags should point opposite to normal.
(4)                 Normal behavior is for the barbs and lines to point right (comes from wind
(4)                 barbs having these features point towards low pressure in the Northern
(4)                 Hemisphere).
(4)                 A single value is applied to all barbs. Individual barbs can be flipped by
(4)                 passing a bool array of the same size as *U* and *V*.
(0)             Returns
(0)             -------
(0)             barbs : `~matplotlib.quiver.Barbs`
(0)             Other Parameters
(0)             ----------------
(0)             data : indexable object, optional
(4)                 DATA_PARAMETER_PLACEHOLDER
(0)             **kwargs
(4)                 The barbs can further be customized using `.PolyCollection` keyword
(4)                 arguments:
(4)                 %(PolyCollection:kwdoc)s
(0)             """ % _docstring.interpd.params
(0)             _docstring.interpd.update(barbs_doc=_barbs_doc)
(0)             class Barbs(mcollections.PolyCollection):
(4)                 """
(4)                 Specialized PolyCollection for barbs.
(4)                 The only API method is :meth:`set_UVC`, which can be used to
(4)                 change the size, orientation, and color of the arrows.  Locations
(4)                 are changed using the :meth:`set_offsets` collection method.
(4)                 Possibly this method will be useful in animations.
(4)                 There is one internal function :meth:`_find_tails` which finds
(4)                 exactly what should be put on the barb given the vector magnitude.
```

```
(4)                 From there :meth:`_make_barbs` is used to find the vertices of the
(4)                 polygon to represent the barb based on this information.
(4)                 """
(4)                 @_docstring.interpd
(4)                 def __init__(self, ax, *args,
(17)                             pivot='tip', length=7, barbcolor=None, flagcolor=None,
(17)                             sizes=None, fill_empty=False, barb_increments=None,
(17)                             rounding=True, flip_barb=False, **kwargs):
(8)                     """
(8)                     The constructor takes one required argument, an Axes
(8)                     instance, followed by the args and kwargs described
(8)                     by the following pyplot interface documentation:
(8)                     %(barbs_doc)s
(8)                     """
(8)                     self.sizes = sizes or dict()
(8)                     self.fill_empty = fill_empty
(8)                     self.barb_increments = barb_increments or dict()
(8)                     self.rounding = rounding
(8)                     self.flip = np.atleast_1d(flip_barb)
(8)                     transform = kwargs.pop('transform', ax.transData)
(8)                     self._pivot = pivot
(8)                     self._length = length
(8)                     if None in (barbcolor, flagcolor):
(12)                        kwargs['edgecolors'] = 'face'
(12)                        if flagcolor:
(16)                            kwargs['facecolors'] = flagcolor
(12)                        elif barbcolor:
(16)                            kwargs['facecolors'] = barbcolor
(12)                        else:
(16)                            kwargs.setdefault('facecolors', 'k')
(8)                     else:
(12)                        kwargs['edgecolors'] = barbcolor
(12)                        kwargs['facecolors'] = flagcolor
(8)                     if 'linewidth' not in kwargs and 'lw' not in kwargs:
(12)                        kwargs['linewidth'] = 1
(8)                     x, y, u, v, c = _parse_args(*args, caller_name='barbs')
(8)                     self.x = x
(8)                     self.y = y
(8)                     xy = np.column_stack((x, y))
(8)                     barb_size = self._length ** 2 / 4  # Empirically determined
(8)                     super().__init__(
(12)                        [], (barb_size,), offsets=xy, offset_transform=transform,
**kwargs)
(8)                     self.set_transform(transforms.IdentityTransform())
(8)                     self.set_UVC(u, v, c)
(4)                 def _find_tails(self, mag, rounding=True, half=5, full=10, flag=50):
(8)                     """
(8)                     Find how many of each of the tail pieces is necessary.
(8)                     Parameters
(8)                     ----------
(8)                     mag : `~numpy.ndarray`
(12)                        Vector magnitudes; must be non-negative (and an actual ndarray).
(8)                     rounding : bool, default: True
(12)                        Whether to round or to truncate to the nearest half-barb.
(8)                     half, full, flag : float, defaults: 5, 10, 50
(12)                        Increments for a half-barb, a barb, and a flag.
(8)                     Returns
(8)                     -------
(8)                     n_flags, n_barbs : int array
(12)                        For each entry in *mag*, the number of flags and barbs.
(8)                     half_flag : bool array
(12)                        For each entry in *mag*, whether a half-barb is needed.
(8)                     empty_flag : bool array
(12)                        For each entry in *mag*, whether nothing is drawn.
(8)                     """
(8)                     if rounding:
(12)                        mag = half * np.around(mag / half)
(8)                     n_flags, mag = divmod(mag, flag)
(8)                     n_barb, mag = divmod(mag, full)
```

```
(8)                 half_flag = mag >= half
(8)                 empty_flag = ~(half_flag | (n_flags > 0) | (n_barb > 0))
(8)                 return n_flags.astype(int), n_barb.astype(int), half_flag, empty_flag
(4)             def _make_barbs(self, u, v, nflags, nbarbs, half_barb, empty_flag, length,
(20)                            pivot, sizes, fill_empty, flip):
(8)                 """
(8)                 Create the wind barbs.
(8)                 Parameters
(8)                 ----------
(8)                 u, v
(12)                    Components of the vector in the x and y directions, respectively.
(8)                 nflags, nbarbs, half_barb, empty_flag
(12)                    Respectively, the number of flags, number of barbs, flag for
(12)                    half a barb, and flag for empty barb, ostensibly obtained from
(12)                    :meth:`_find_tails`.
(8)                 length
(12)                    The length of the barb staff in points.
(8)                 pivot : {"tip", "middle"} or number
(12)                    The point on the barb around which the entire barb should be
(12)                    rotated.  If a number, the start of the barb is shifted by that
(12)                    many points from the origin.
(8)                 sizes : dict
(12)                    Coefficients specifying the ratio of a given feature to the length
(12)                    of the barb. These features include:
(12)                    - *spacing*: space between features (flags, full/half barbs).
(12)                    - *height*: distance from shaft of top of a flag or full barb.
(12)                    - *width*: width of a flag, twice the width of a full barb.
(12)                    - *emptybarb*: radius of the circle used for low magnitudes.
(8)                 fill_empty : bool
(12)                    Whether the circle representing an empty barb should be filled or
(12)                    not (this changes the drawing of the polygon).
(8)                 flip : list of bool
(12)                    Whether the features should be flipped to the other side of the
(12)                    barb (useful for winds in the southern hemisphere).
(8)                 Returns
(8)                 -------
(8)                 list of arrays of vertices
(12)                    Polygon vertices for each of the wind barbs.  These polygons have
(12)                    been rotated to properly align with the vector direction.
(8)                 """
(8)                 spacing = length * sizes.get('spacing', 0.125)
(8)                 full_height = length * sizes.get('height', 0.4)
(8)                 full_width = length * sizes.get('width', 0.25)
(8)                 empty_rad = length * sizes.get('emptybarb', 0.15)
(8)                 pivot_points = dict(tip=0.0, middle=-length / 2.)
(8)                 endx = 0.0
(8)                 try:
(12)                    endy = float(pivot)
(8)                 except ValueError:
(12)                    endy = pivot_points[pivot.lower()]
(8)                 angles = -(ma.arctan2(v, u) + np.pi / 2)
(8)                 circ = CirclePolygon((0, 0), radius=empty_rad).get_verts()
(8)                 if fill_empty:
(12)                    empty_barb = circ
(8)                 else:
(12)                    empty_barb = np.concatenate((circ, circ[::-1]))
(8)                 barb_list = []
(8)                 for index, angle in np.ndenumerate(angles):
(12)                    if empty_flag[index]:
(16)                        barb_list.append(empty_barb)
(16)                        continue
(12)                    poly_verts = [(endx, endy)]
(12)                    offset = length
(12)                    barb_height = -full_height if flip[index] else full_height
(12)                    for i in range(nflags[index]):
(16)                        if offset != length:
(20)                            offset += spacing / 2.
(16)                        poly_verts.extend(
(20)                            [[endx, endy + offset],
```

```
826: (21)                                [endx + barb_height, endy - full_width / 2 + offset],
827: (21)                                [endx, endy - full_width + offset]])
828: (16)                            offset -= full_width + spacing
829: (12)                        for i in range(nbarbs[index]):
830: (16)                            poly_verts.extend(
831: (20)                                [(endx, endy + offset),
832: (21)                                 (endx + barb_height, endy + offset + full_width / 2),
833: (21)                                 (endx, endy + offset)])
834: (16)                            offset -= spacing
835: (12)                        if half_barb[index]:
836: (16)                            if offset == length:
837: (20)                                poly_verts.append((endx, endy + offset))
838: (20)                                offset -= 1.5 * spacing
839: (16)                            poly_verts.extend(
840: (20)                                [(endx, endy + offset),
841: (21)                                 (endx + barb_height / 2, endy + offset + full_width / 4),
842: (21)                                 (endx, endy + offset)])
843: (12)                        poly_verts = transforms.Affine2D().rotate(-angle).transform(
844: (16)                            poly_verts)
845: (12)                        barb_list.append(poly_verts)
846: (8)                    return barb_list
847: (4)                def set_UVC(self, U, V, C=None):
848: (8)                    self.u = ma.masked_invalid(U, copy=True).ravel()
849: (8)                    self.v = ma.masked_invalid(V, copy=True).ravel()
850: (8)                    if len(self.flip) == 1:
851: (12)                       flip = np.broadcast_to(self.flip, self.u.shape)
852: (8)                    else:
853: (12)                       flip = self.flip
854: (8)                    if C is not None:
855: (12)                       c = ma.masked_invalid(C, copy=True).ravel()
856: (12)                       x, y, u, v, c, flip = cbook.delete_masked_points(
857: (16)                           self.x.ravel(), self.y.ravel(), self.u, self.v, c,
858: (16)                           flip.ravel())
859: (12)                       _check_consistent_shapes(x, y, u, v, c, flip)
860: (8)                    else:
861: (12)                       x, y, u, v, flip = cbook.delete_masked_points(
862: (16)                           self.x.ravel(), self.y.ravel(), self.u, self.v, flip.ravel())
863: (12)                       _check_consistent_shapes(x, y, u, v, flip)
864: (8)                    magnitude = np.hypot(u, v)
865: (8)                    flags, barbs, halves, empty = self._find_tails(
866: (12)                       magnitude, self.rounding, **self.barb_increments)
867: (8)                    plot_barbs = self._make_barbs(u, v, flags, barbs, halves, empty,
868: (38)                                                 self._length, self._pivot, self.sizes,
869: (38)                                                 self.fill_empty, flip)
870: (8)                    self.set_verts(plot_barbs)
871: (8)                    if C is not None:
872: (12)                       self.set_array(c)
873: (8)                    xy = np.column_stack((x, y))
874: (8)                    self._offsets = xy
875: (8)                    self.stale = True
876: (4)                def set_offsets(self, xy):
877: (8)                    """
878: (8)                    Set the offsets for the barb polygons.  This saves the offsets passed
879: (8)                    in and masks them as appropriate for the existing U/V data.
880: (8)                    Parameters
881: (8)                    ----------
882: (8)                    xy : sequence of pairs of floats
883: (8)                    """
884: (8)                    self.x = xy[:, 0]
885: (8)                    self.y = xy[:, 1]
886: (8)                    x, y, u, v = cbook.delete_masked_points(
887: (12)                       self.x.ravel(), self.y.ravel(), self.u, self.v)
888: (8)                    _check_consistent_shapes(x, y, u, v)
889: (8)                    xy = np.column_stack((x, y))
890: (8)                    super().set_offsets(xy)
891: (8)                    self.stale = True

----------------------------------------
```

File 57 - rcsetup.py:

```
1: (0)              """
2: (0)              The rcsetup module contains the validation code for customization using
3: (0)              Matplotlib's rc settings.
4: (0)              Each rc setting is assigned a function used to validate any attempted changes
5: (0)              to that setting.  The validation functions are defined in the rcsetup module,
6: (0)              and are used to construct the rcParams global object which stores the settings
7: (0)              and is referenced throughout Matplotlib.
8: (0)              The default values of the rc settings are set in the default matplotlibrc
file.
9: (0)              Any additions or deletions to the parameter set listed here should also be
10: (0)             propagated to the :file:`lib/matplotlib/mpl-data/matplotlibrc` in Matplotlib's
11: (0)             root source directory.
12: (0)             """
13: (0)             import ast
14: (0)             from functools import lru_cache, reduce
15: (0)             from numbers import Real
16: (0)             import operator
17: (0)             import os
18: (0)             import re
19: (0)             import numpy as np
20: (0)             from matplotlib import _api, cbook
21: (0)             from matplotlib.backends import BackendFilter, backend_registry
22: (0)             from matplotlib.cbook import ls_mapper
23: (0)             from matplotlib.colors import Colormap, is_color_like
24: (0)             from matplotlib._fontconfig_pattern import parse_fontconfig_pattern
25: (0)             from matplotlib._enums import JoinStyle, CapStyle
26: (0)             from cycler import Cycler, cycler as ccycler
27: (0)             @_api.caching_module_getattr
28: (0)             class __getattr__:
29: (4)                 @_api.deprecated(
30: (8)                     "3.9",
31: (8)                     alternative="``matplotlib.backends.backend_registry.list_builtin"
32: (12)                        "(matplotlib.backends.BackendFilter.INTERACTIVE)``")
33: (4)                 @property
34: (4)                 def interactive_bk(self):
35: (8)                     return backend_registry.list_builtin(BackendFilter.INTERACTIVE)
36: (4)                 @_api.deprecated(
37: (8)                     "3.9",
38: (8)                     alternative="``matplotlib.backends.backend_registry.list_builtin"
39: (12)                        "(matplotlib.backends.BackendFilter.NON_INTERACTIVE)``")
40: (4)                 @property
41: (4)                 def non_interactive_bk(self):
42: (8)                     return backend_registry.list_builtin(BackendFilter.NON_INTERACTIVE)
43: (4)                 @_api.deprecated(
44: (8)                     "3.9",
45: (8)                     alternative="``matplotlib.backends.backend_registry.list_builtin()``")
46: (4)                 @property
47: (4)                 def all_backends(self):
48: (8)                     return backend_registry.list_builtin()
49: (0)             class ValidateInStrings:
50: (4)                 def __init__(self, key, valid, ignorecase=False, *,
51: (17)                             _deprecated_since=None):
52: (8)                     """*valid* is a list of legal strings."""
53: (8)                     self.key = key
54: (8)                     self.ignorecase = ignorecase
55: (8)                     self._deprecated_since = _deprecated_since
56: (8)                     def func(s):
57: (12)                        if ignorecase:
58: (16)                            return s.lower()
59: (12)                        else:
60: (16)                            return s
61: (8)                     self.valid = {func(k): k for k in valid}
62: (4)                 def __call__(self, s):
63: (8)                     if self._deprecated_since:
64: (12)                        name, = (k for k, v in globals().items() if v is self)
65: (12)                        _api.warn_deprecated(
66: (16)                            self._deprecated_since, name=name, obj_type="function")
```

```
67: (8)                     if self.ignorecase and isinstance(s, str):
68: (12)                        s = s.lower()
69: (8)                     if s in self.valid:
70: (12)                        return self.valid[s]
71: (8)                     msg = (f"{s!r} is not a valid value for {self.key}; supported values "
72: (15)                           f"are {[*self.valid.values()]}")
73: (8)                     if (isinstance(s, str)
74: (16)                            and (s.startswith('"') and s.endswith('"')
75: (21)                                 or s.startswith("'") and s.endswith("'"))
76: (16)                            and s[1:-1] in self.valid):
77: (12)                        msg += "; remove quotes surrounding your string"
78: (8)                     raise ValueError(msg)
79: (0)             @lru_cache
80: (0)             def _listify_validator(scalar_validator, allow_stringlist=False, *,
81: (23)                                   n=None, doc=None):
82: (4)                 def f(s):
83: (8)                     if isinstance(s, str):
84: (12)                        try:
85: (16)                            val = [scalar_validator(v.strip()) for v in s.split(',')
86: (23)                                   if v.strip()]
87: (12)                        except Exception:
88: (16)                            if allow_stringlist:
89: (20)                                val = [scalar_validator(v.strip()) for v in s if
v.strip()]
90: (16)                            else:
91: (20)                                raise
92: (8)                     elif np.iterable(s) and not isinstance(s, (set, frozenset)):
93: (12)                        val = [scalar_validator(v) for v in s
94: (19)                               if not isinstance(v, str) or v]
95: (8)                     else:
96: (12)                        raise ValueError(
97: (16)                            f"Expected str or other non-set iterable, but got {s}")
98: (8)                     if n is not None and len(val) != n:
99: (12)                        raise ValueError(
100: (16)                           f"Expected {n} values, but there are {len(val)} values in
{s}")
101: (8)                    return val
102: (4)                try:
103: (8)                    f.__name__ = f"{scalar_validator.__name__}list"
104: (4)                except AttributeError:  # class instance.
105: (8)                    f.__name__ = f"{type(scalar_validator).__name__}List"
106: (4)                f.__qualname__ = f.__qualname__.rsplit(".", 1)[0] + "." + f.__name__
107: (4)                f.__doc__ = doc if doc is not None else scalar_validator.__doc__
108: (4)                return f
109: (0)            def validate_any(s):
110: (4)                return s
111: (0)            validate_anylist = _listify_validator(validate_any)
112: (0)            def _validate_date(s):
113: (4)                try:
114: (8)                    np.datetime64(s)
115: (8)                    return s
116: (4)                except ValueError:
117: (8)                    raise ValueError(
118: (12)                       f'{s!r} should be a string that can be parsed by
numpy.datetime64')
119: (0)            def validate_bool(b):
120: (4)                """Convert b to ``bool`` or raise."""
121: (4)                if isinstance(b, str):
122: (8)                    b = b.lower()
123: (4)                if b in ('t', 'y', 'yes', 'on', 'true', '1', 1, True):
124: (8)                    return True
125: (4)                elif b in ('f', 'n', 'no', 'off', 'false', '0', 0, False):
126: (8)                    return False
127: (4)                else:
128: (8)                    raise ValueError(f'Cannot convert {b!r} to bool')
129: (0)            def validate_axisbelow(s):
130: (4)                try:
131: (8)                    return validate_bool(s)
132: (4)                except ValueError:
```

```
133: (8)                    if isinstance(s, str):
134: (12)                       if s == 'line':
135: (16)                           return 'line'
136: (4)                raise ValueError(f'{s!r} cannot be interpreted as'
137: (21)                                ' True, False, or "line"')
138: (0)            def validate_dpi(s):
139: (4)                """Confirm s is string 'figure' or convert s to float or raise."""
140: (4)                if s == 'figure':
141: (8)                    return s
142: (4)                try:
143: (8)                    return float(s)
144: (4)                except ValueError as e:
145: (8)                    raise ValueError(f'{s!r} is not string "figure" and '
146: (25)                                     f'could not convert {s!r} to float') from e
147: (0)            def _make_type_validator(cls, *, allow_none=False):
148: (4)                """
149: (4)                Return a validator that converts inputs to *cls* or raises (and possibly
150: (4)                allows ``None`` as well).
151: (4)                """
152: (4)                def validator(s):
153: (8)                    if (allow_none and
154: (16)                           (s is None or cbook._str_lower_equal(s, "none"))):
155: (12)                       return None
156: (8)                    if cls is str and not isinstance(s, str):
157: (12)                       raise ValueError(f'Could not convert {s!r} to str')
158: (8)                    try:
159: (12)                       return cls(s)
160: (8)                    except (TypeError, ValueError) as e:
161: (12)                       raise ValueError(
162: (16)                           f'Could not convert {s!r} to {cls.__name__}') from e
163: (4)                validator.__name__ = f"validate_{cls.__name__}"
164: (4)                if allow_none:
165: (8)                    validator.__name__ += "_or_None"
166: (4)                validator.__qualname__ = (
167: (8)                    validator.__qualname__.rsplit(".", 1)[0] + "." + validator.__name__)
168: (4)                return validator
169: (0)            validate_string = _make_type_validator(str)
170: (0)            validate_string_or_None = _make_type_validator(str, allow_none=True)
171: (0)            validate_stringlist = _listify_validator(
172: (4)                validate_string, doc='return a list of strings')
173: (0)            validate_int = _make_type_validator(int)
174: (0)            validate_int_or_None = _make_type_validator(int, allow_none=True)
175: (0)            validate_float = _make_type_validator(float)
176: (0)            validate_float_or_None = _make_type_validator(float, allow_none=True)
177: (0)            validate_floatlist = _listify_validator(
178: (4)                validate_float, doc='return a list of floats')
179: (0)            def _validate_marker(s):
180: (4)                try:
181: (8)                    return validate_int(s)
182: (4)                except ValueError as e:
183: (8)                    try:
184: (12)                       return validate_string(s)
185: (8)                    except ValueError as e:
186: (12)                       raise ValueError('Supported markers are [string, int]') from e
187: (0)            _validate_markerlist = _listify_validator(
188: (4)                _validate_marker, doc='return a list of markers')
189: (0)            def _validate_pathlike(s):
190: (4)                if isinstance(s, (str, os.PathLike)):
191: (8)                    return os.fsdecode(s)
192: (4)                else:
193: (8)                    return validate_string(s)
194: (0)            def validate_fonttype(s):
195: (4)                """
196: (4)                Confirm that this is a Postscript or PDF font type that we know how to
197: (4)                convert to.
198: (4)                """
199: (4)                fonttypes = {'type3':    3,
200: (17)                            'truetype': 42}
201: (4)                try:
```

```
202: (8)                    fonttype = validate_int(s)
203: (4)                except ValueError:
204: (8)                    try:
205: (12)                       return fonttypes[s.lower()]
206: (8)                    except KeyError as e:
207: (12)                       raise ValueError('Supported Postscript/PDF font types are %s'
208: (29)                                        % list(fonttypes)) from e
209: (4)                else:
210: (8)                    if fonttype not in fonttypes.values():
211: (12)                       raise ValueError(
212: (16)                           'Supported Postscript/PDF font types are %s' %
213: (16)                           list(fonttypes.values()))
214: (8)                    return fonttype
215: (0)            _auto_backend_sentinel = object()
216: (0)            def validate_backend(s):
217: (4)                if s is _auto_backend_sentinel or backend_registry.is_valid_backend(s):
218: (8)                    return s
219: (4)                else:
220: (8)                    msg = (f"'{s}' is not a valid value for backend; supported values are
"
221: (15)                          f"{backend_registry.list_all()}")
222: (8)                    raise ValueError(msg)
223: (0)            def _validate_toolbar(s):
224: (4)                s = ValidateInStrings(
225: (8)                    'toolbar', ['None', 'toolbar2', 'toolmanager'], ignorecase=True)(s)
226: (4)                if s == 'toolmanager':
227: (8)                    _api.warn_external(
228: (12)                       "Treat the new Tool classes introduced in v1.5 as experimental "
229: (12)                       "for now; the API and rcParam may change in future versions.")
230: (4)                return s
231: (0)            def validate_color_or_inherit(s):
232: (4)                """Return a valid color arg."""
233: (4)                if cbook._str_equal(s, 'inherit'):
234: (8)                    return s
235: (4)                return validate_color(s)
236: (0)            def validate_color_or_auto(s):
237: (4)                if cbook._str_equal(s, 'auto'):
238: (8)                    return s
239: (4)                return validate_color(s)
240: (0)            def validate_color_for_prop_cycle(s):
241: (4)                if isinstance(s, str) and re.match("^C[0-9]$", s):
242: (8)                    raise ValueError(f"Cannot put cycle reference ({s!r}) in prop_cycler")
243: (4)                return validate_color(s)
244: (0)            def _validate_color_or_linecolor(s):
245: (4)                if cbook._str_equal(s, 'linecolor'):
246: (8)                    return s
247: (4)                elif cbook._str_equal(s, 'mfc') or cbook._str_equal(s, 'markerfacecolor'):
248: (8)                    return 'markerfacecolor'
249: (4)                elif cbook._str_equal(s, 'mec') or cbook._str_equal(s, 'markeredgecolor'):
250: (8)                    return 'markeredgecolor'
251: (4)                elif s is None:
252: (8)                    return None
253: (4)                elif isinstance(s, str) and len(s) == 6 or len(s) == 8:
254: (8)                    stmp = '#' + s
255: (8)                    if is_color_like(stmp):
256: (12)                       return stmp
257: (8)                    if s.lower() == 'none':
258: (12)                       return None
259: (4)                elif is_color_like(s):
260: (8)                    return s
261: (4)                raise ValueError(f'{s!r} does not look like a color arg')
262: (0)            def validate_color(s):
263: (4)                """Return a valid color arg."""
264: (4)                if isinstance(s, str):
265: (8)                    if s.lower() == 'none':
266: (12)                       return 'none'
267: (8)                    if len(s) == 6 or len(s) == 8:
268: (12)                       stmp = '#' + s
269: (12)                       if is_color_like(stmp):
```

```
270: (16)                            return stmp
271: (4)                 if is_color_like(s):
272: (8)                     return s
273: (4)                 try:
274: (8)                     color = ast.literal_eval(s)
275: (4)                 except (SyntaxError, ValueError):
276: (8)                     pass
277: (4)                 else:
278: (8)                     if is_color_like(color):
279: (12)                        return color
280: (4)                 raise ValueError(f'{s!r} does not look like a color arg')
281: (0)             validate_colorlist = _listify_validator(
282: (4)                 validate_color, allow_stringlist=True, doc='return a list of colorspecs')
283: (0)             def _validate_cmap(s):
284: (4)                 _api.check_isinstance((str, Colormap), cmap=s)
285: (4)                 return s
286: (0)             def validate_aspect(s):
287: (4)                 if s in ('auto', 'equal'):
288: (8)                     return s
289: (4)                 try:
290: (8)                     return float(s)
291: (4)                 except ValueError as e:
292: (8)                     raise ValueError('not a valid aspect specification') from e
293: (0)             def validate_fontsize_None(s):
294: (4)                 if s is None or s == 'None':
295: (8)                     return None
296: (4)                 else:
297: (8)                     return validate_fontsize(s)
298: (0)             def validate_fontsize(s):
299: (4)                 fontsizes = ['xx-small', 'x-small', 'small', 'medium', 'large',
300: (17)                             'x-large', 'xx-large', 'smaller', 'larger']
301: (4)                 if isinstance(s, str):
302: (8)                     s = s.lower()
303: (4)                 if s in fontsizes:
304: (8)                     return s
305: (4)                 try:
306: (8)                     return float(s)
307: (4)                 except ValueError as e:
308: (8)                     raise ValueError("%s is not a valid font size. Valid font sizes "
309: (25)                                      "are %s." % (s, ", ".join(fontsizes))) from e
310: (0)             validate_fontsizelist = _listify_validator(validate_fontsize)
311: (0)             def validate_fontweight(s):
312: (4)                 weights = [
313: (8)                     'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman',
314: (8)                     'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold',
'black']
315: (4)                 if s in weights:
316: (8)                     return s
317: (4)                 try:
318: (8)                     return int(s)
319: (4)                 except (ValueError, TypeError) as e:
320: (8)                     raise ValueError(f'{s} is not a valid font weight.') from e
321: (0)             def validate_fontstretch(s):
322: (4)                 stretchvalues = [
323: (8)                     'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed',
324: (8)                     'normal', 'semi-expanded', 'expanded', 'extra-expanded',
325: (8)                     'ultra-expanded']
326: (4)                 if s in stretchvalues:
327: (8)                     return s
328: (4)                 try:
329: (8)                     return int(s)
330: (4)                 except (ValueError, TypeError) as e:
331: (8)                     raise ValueError(f'{s} is not a valid font stretch.') from e
332: (0)             def validate_font_properties(s):
333: (4)                 parse_fontconfig_pattern(s)
334: (4)                 return s
335: (0)             def _validate_mathtext_fallback(s):
336: (4)                 _fallback_fonts = ['cm', 'stix', 'stixsans']
337: (4)                 if isinstance(s, str):
```

```
(8)                 s = s.lower()
(4)             if s is None or s == 'none':
(8)                 return None
(4)             elif s.lower() in _fallback_fonts:
(8)                 return s
(4)             else:
(8)                 raise ValueError(
(12)                    f"{s} is not a valid fallback font name. Valid fallback font "
(12)                    f"names are {','.join(_fallback_fonts)}. Passing 'None' will turn
"
(12)                    "fallback off.")
(0)         def validate_whiskers(s):
(4)             try:
(8)                 return _listify_validator(validate_float, n=2)(s)
(4)             except (TypeError, ValueError):
(8)                 try:
(12)                    return float(s)
(8)                 except ValueError as e:
(12)                    raise ValueError("Not a valid whisker value [float, "
(29)                                     "(float, float)]") from e
(0)         def validate_ps_distiller(s):
(4)             if isinstance(s, str):
(8)                 s = s.lower()
(4)             if s in ('none', None, 'false', False):
(8)                 return None
(4)             else:
(8)                 return ValidateInStrings('ps.usedistiller', ['ghostscript', 'xpdf'])
(s)
(0)         def _validate_papersize(s):
(4)             s = ValidateInStrings("ps.papersize",
(26)                                  ["figure", "auto", "letter", "legal", "ledger",
(27)                                   *[f"{ab}{i}" for ab in "ab" for i in range(11)]],
(26)                                  ignorecase=True)(s)
(4)             if s == "auto":
(8)                 _api.warn_deprecated("3.8", name="ps.papersize='auto'",
(29)                                     addendum="Pass an explicit paper type, figure, or
omit "
(29)                                     "the *ps.papersize* rcParam entirely.")
(4)             return s
(0)         _validate_named_linestyle = ValidateInStrings(
(4)             'linestyle',
(4)             [*ls_mapper.keys(), *ls_mapper.values(), 'None', 'none', ' ', ''],
(4)             ignorecase=True)
(0)         def _validate_linestyle(ls):
(4)             """
(4)             A validator for all possible line styles, the named ones *and*
(4)             the on-off ink sequences.
(4)             """
(4)             if isinstance(ls, str):
(8)                 try:  # Look first for a valid named line style, like '--' or 'solid'.
(12)                    return _validate_named_linestyle(ls)
(8)                 except ValueError:
(12)                    pass
(8)                 try:
(12)                    ls = ast.literal_eval(ls)  # Parsing matplotlibrc.
(8)                 except (SyntaxError, ValueError):
(12)                    pass  # Will error with the ValueError at the end.
(4)             def _is_iterable_not_string_like(x):
(8)                 return np.iterable(x) and not isinstance(x, (str, bytes, bytearray))
(4)             if _is_iterable_not_string_like(ls):
(8)                 if len(ls) == 2 and _is_iterable_not_string_like(ls[1]):
(12)                    offset, onoff = ls
(8)                 else:
(12)                    offset = 0
(12)                    onoff = ls
(8)                 if (isinstance(offset, Real)
(16)                        and len(onoff) % 2 == 0
(16)                        and all(isinstance(elem, Real) for elem in onoff)):
(12)                    return (offset, onoff)
```

```
(4)                 raise ValueError(f"linestyle {ls!r} is not a valid on-off ink sequence.")
(0)             validate_fillstyle = ValidateInStrings(
(4)                 'markers.fillstyle', ['full', 'left', 'right', 'bottom', 'top', 'none'])
(0)             validate_fillstylelist = _listify_validator(validate_fillstyle)
(0)             def validate_markevery(s):
(4)                 """
(4)                 Validate the markevery property of a Line2D object.
(4)                 Parameters
(4)                 ----------
(4)                 s : None, int, (int, int), slice, float, (float, float), or list[int]
(4)                 Returns
(4)                 -------
(4)                 None, int, (int, int), slice, float, (float, float), or list[int]
(4)                 """
(4)                 if isinstance(s, (slice, float, int, type(None))):
(8)                     return s
(4)                 if isinstance(s, tuple):
(8)                     if (len(s) == 2
(16)                            and (all(isinstance(e, int) for e in s)
(21)                                 or all(isinstance(e, float) for e in s))):
(12)                        return s
(8)                     else:
(12)                        raise TypeError(
(16)                            "'markevery' tuple must be pair of ints or of floats")
(4)                 if isinstance(s, list):
(8)                     if all(isinstance(e, int) for e in s):
(12)                        return s
(8)                     else:
(12)                        raise TypeError(
(16)                            "'markevery' list must have all elements of type int")
(4)                 raise TypeError("'markevery' is of an invalid type")
(0)             validate_markeverylist = _listify_validator(validate_markevery)
(0)             def validate_bbox(s):
(4)                 if isinstance(s, str):
(8)                     s = s.lower()
(8)                     if s == 'tight':
(12)                        return s
(8)                     if s == 'standard':
(12)                        return None
(8)                     raise ValueError("bbox should be 'tight' or 'standard'")
(4)                 elif s is not None:
(8)                     raise ValueError("bbox should be 'tight' or 'standard'")
(4)                 return s
(0)             def validate_sketch(s):
(4)                 if isinstance(s, str):
(8)                     s = s.lower().strip()
(8)                     if s.startswith("(") and s.endswith(")"):
(12)                        s = s[1:-1]
(4)                 if s == 'none' or s is None:
(8)                     return None
(4)                 try:
(8)                     return tuple(_listify_validator(validate_float, n=3)(s))
(4)                 except ValueError as exc:
(8)                     raise ValueError("Expected a (scale, length, randomness) tuple") from
exc
(0)             def _validate_greaterthan_minushalf(s):
(4)                 s = validate_float(s)
(4)                 if s > -0.5:
(8)                     return s
(4)                 else:
(8)                     raise RuntimeError(f'Value must be >-0.5; got {s}')
(0)             def _validate_greaterequal0_lessequal1(s):
(4)                 s = validate_float(s)
(4)                 if 0 <= s <= 1:
(8)                     return s
(4)                 else:
(8)                     raise RuntimeError(f'Value must be >=0 and <=1; got {s}')
(0)             def _validate_int_greaterequal0(s):
(4)                 s = validate_int(s)
```

```
(4)         if s >= 0:
(8)             return s
(4)         else:
(8)             raise RuntimeError(f'Value must be >=0; got {s}')
(0)     def validate_hatch(s):
(4)         r"""
(4)         Validate a hatch pattern.
(4)         A hatch pattern string can have any sequence of the following
(4)         characters: ``\ / | - + * . x o O``.
(4)         """
(4)         if not isinstance(s, str):
(8)             raise ValueError("Hatch pattern must be a string")
(4)         _api.check_isinstance(str, hatch_pattern=s)
(4)         unknown = set(s) - {'\\', '/', '|', '-', '+', '*', '.', 'x', 'o', 'O'}
(4)         if unknown:
(8)             raise ValueError("Unknown hatch symbol(s): %s" % list(unknown))
(4)         return s
(0)     validate_hatchlist = _listify_validator(validate_hatch)
(0)     validate_dashlist = _listify_validator(validate_floatlist)
(0)     def _validate_minor_tick_ndivs(n):
(4)         """
(4)         Validate ndiv parameter related to the minor ticks.
(4)         It controls the number of minor ticks to be placed between
(4)         two major ticks.
(4)         """
(4)         if cbook._str_lower_equal(n, 'auto'):
(8)             return n
(4)         try:
(8)             n = _validate_int_greaterequal0(n)
(8)             return n
(4)         except (RuntimeError, ValueError):
(8)             pass
(4)         raise ValueError("'tick.minor.ndivs' must be 'auto' or non-negative int")
(0)     _prop_validators = {
(8)             'color': _listify_validator(validate_color_for_prop_cycle,
(36)                                         allow_stringlist=True),
(8)             'linewidth': validate_floatlist,
(8)             'linestyle': _listify_validator(_validate_linestyle),
(8)             'facecolor': validate_colorlist,
(8)             'edgecolor': validate_colorlist,
(8)             'joinstyle': _listify_validator(JoinStyle),
(8)             'capstyle': _listify_validator(CapStyle),
(8)             'fillstyle': validate_fillstylelist,
(8)             'markerfacecolor': validate_colorlist,
(8)             'markersize': validate_floatlist,
(8)             'markeredgewidth': validate_floatlist,
(8)             'markeredgecolor': validate_colorlist,
(8)             'markevery': validate_markeverylist,
(8)             'alpha': validate_floatlist,
(8)             'marker': _validate_markerlist,
(8)             'hatch': validate_hatchlist,
(8)             'dashes': validate_dashlist,
(4)         }
(0)     _prop_aliases = {
(8)             'c': 'color',
(8)             'lw': 'linewidth',
(8)             'ls': 'linestyle',
(8)             'fc': 'facecolor',
(8)             'ec': 'edgecolor',
(8)             'mfc': 'markerfacecolor',
(8)             'mec': 'markeredgecolor',
(8)             'mew': 'markeredgewidth',
(8)             'ms': 'markersize',
(4)         }
(0)     def cycler(*args, **kwargs):
(4)         """
(4)         Create a `~cycler.Cycler` object much like :func:`cycler.cycler`,
(4)         but includes input validation.
(4)         Call signatures::
```

```
(6)                 cycler(cycler)
(6)                 cycler(label=values[, label2=values2[, ...]])
(6)                 cycler(label, values)
(4)             Form 1 copies a given `~cycler.Cycler` object.
(4)             Form 2 creates a `~cycler.Cycler` which cycles over one or more
(4)             properties simultaneously. If multiple properties are given, their
(4)             value lists must have the same length.
(4)             Form 3 creates a `~cycler.Cycler` for a single property. This form
(4)             exists for compatibility with the original cycler. Its use is
(4)             discouraged in favor of the kwarg form, i.e. ``cycler(label=values)``.
(4)             Parameters
(4)             ----------
(4)             cycler : Cycler
(8)                 Copy constructor for Cycler.
(4)             label : str
(8)                 The property key. Must be a valid `.Artist` property.
(8)                 For example, 'color' or 'linestyle'. Aliases are allowed,
(8)                 such as 'c' for 'color' and 'lw' for 'linewidth'.
(4)             values : iterable
(8)                 Finite-length iterable of the property values. These values
(8)                 are validated and will raise a ValueError if invalid.
(4)             Returns
(4)             -------
(4)             Cycler
(8)                 A new :class:`~cycler.Cycler` for the given properties.
(4)             Examples
(4)             --------
(4)             Creating a cycler for a single property:
(4)             >>> c = cycler(color=['red', 'green', 'blue'])
(4)             Creating a cycler for simultaneously cycling over multiple properties
(4)             (e.g. red circle, green plus, blue cross):
(4)             >>> c = cycler(color=['red', 'green', 'blue'],
(4)             ...            marker=['o', '+', 'x'])
(4)             """
(4)             if args and kwargs:
(8)                 raise TypeError("cycler() can only accept positional OR keyword "
(24)                                "arguments -- not both.")
(4)             elif not args and not kwargs:
(8)                 raise TypeError("cycler() must have positional OR keyword arguments")
(4)             if len(args) == 1:
(8)                 if not isinstance(args[0], Cycler):
(12)                    raise TypeError("If only one positional argument given, it must "
(28)                                    "be a Cycler instance.")
(8)                 return validate_cycler(args[0])
(4)             elif len(args) == 2:
(8)                 pairs = [(args[0], args[1])]
(4)             elif len(args) > 2:
(8)                 raise _api.nargs_error('cycler', '0-2', len(args))
(4)             else:
(8)                 pairs = kwargs.items()
(4)             validated = []
(4)             for prop, vals in pairs:
(8)                 norm_prop = _prop_aliases.get(prop, prop)
(8)                 validator = _prop_validators.get(norm_prop, None)
(8)                 if validator is None:
(12)                    raise TypeError("Unknown artist property: %s" % prop)
(8)                 vals = validator(vals)
(8)                 validated.append((norm_prop, vals))
(4)             return reduce(operator.add, (ccycler(k, v) for k, v in validated))
(0)         class _DunderChecker(ast.NodeVisitor):
(4)             def visit_Attribute(self, node):
(8)                 if node.attr.startswith("__") and node.attr.endswith("__"):
(12)                    raise ValueError("cycler strings with dunders are forbidden")
(8)                 self.generic_visit(node)
(0)         _validate_named_legend_loc = ValidateInStrings(
(4)             'legend.loc',
(4)             [
(8)                 "best",
(8)                 "upper right", "upper left", "lower left", "lower right", "right",
```

```
610: (8)                    "center left", "center right", "lower center", "upper center",
611: (8)                    "center"],
612: (4)                ignorecase=True)
613: (0)            def _validate_legend_loc(loc):
614: (4)                """
615: (4)                Confirm that loc is a type which rc.Params["legend.loc"] supports.
616: (4)                .. versionadded:: 3.8
617: (4)                Parameters
618: (4)                ----------
619: (4)                loc : str | int | (float, float) | str((float, float))
620: (8)                    The location of the legend.
621: (4)                Returns
622: (4)                -------
623: (4)                loc : str | int | (float, float) or raise ValueError exception
624: (8)                    The location of the legend.
625: (4)                """
626: (4)                if isinstance(loc, str):
627: (8)                    try:
628: (12)                       return _validate_named_legend_loc(loc)
629: (8)                    except ValueError:
630: (12)                       pass
631: (8)                    try:
632: (12)                       loc = ast.literal_eval(loc)
633: (8)                    except (SyntaxError, ValueError):
634: (12)                       pass
635: (4)                if isinstance(loc, int):
636: (8)                    if 0 <= loc <= 10:
637: (12)                       return loc
638: (4)                if isinstance(loc, tuple):
639: (8)                    if len(loc) == 2 and all(isinstance(e, Real) for e in loc):
640: (12)                       return loc
641: (4)                raise ValueError(f"{loc} is not a valid legend location.")
642: (0)            def validate_cycler(s):
643: (4)                """Return a Cycler object from a string repr or the object itself."""
644: (4)                if isinstance(s, str):
645: (8)                    try:
646: (12)                       _DunderChecker().visit(ast.parse(s))
647: (12)                       s = eval(s, {'cycler': cycler, '__builtins__': {}})
648: (8)                    except BaseException as e:
649: (12)                       raise ValueError(f"{s!r} is not a valid cycler construction: {e}"
650: (29)                                        ) from e
651: (4)                if isinstance(s, Cycler):
652: (8)                    cycler_inst = s
653: (4)                else:
654: (8)                    raise ValueError(f"Object is not a string or Cycler instance: {s!r}")
655: (4)                unknowns = cycler_inst.keys - (set(_prop_validators) | set(_prop_aliases))
656: (4)                if unknowns:
657: (8)                    raise ValueError("Unknown artist properties: %s" % unknowns)
658: (4)                checker = set()
659: (4)                for prop in cycler_inst.keys:
660: (8)                    norm_prop = _prop_aliases.get(prop, prop)
661: (8)                    if norm_prop != prop and norm_prop in cycler_inst.keys:
662: (12)                       raise ValueError(f"Cannot specify both {norm_prop!r} and alias "
663: (29)                                        f"{prop!r} in the same prop_cycle")
664: (8)                    if norm_prop in checker:
665: (12)                       raise ValueError(f"Another property was already aliased to "
666: (29)                                        f"{norm_prop!r}. Collision normalizing
{prop!r}.")
667: (8)                    checker.update([norm_prop])
668: (4)                assert len(checker) == len(cycler_inst.keys)
669: (4)                for prop in cycler_inst.keys:
670: (8)                    norm_prop = _prop_aliases.get(prop, prop)
671: (8)                    cycler_inst.change_key(prop, norm_prop)
672: (4)                for key, vals in cycler_inst.by_key().items():
673: (8)                    _prop_validators[key](vals)
674: (4)                return cycler_inst
675: (0)            def validate_hist_bins(s):
676: (4)                valid_strs = ["auto", "sturges", "fd", "doane", "scott", "rice", "sqrt"]
677: (4)                if isinstance(s, str) and s in valid_strs:
```

```
(8)                 return s
(4)             try:
(8)                 return int(s)
(4)             except (TypeError, ValueError):
(8)                 pass
(4)             try:
(8)                 return validate_floatlist(s)
(4)             except ValueError:
(8)                 pass
(4)             raise ValueError(f"'hist.bins' must be one of {valid_strs}, an int or"
(21)                             " a sequence of floats")
(0)         class _ignorecase(list):
(4)             """A marker class indicating that a list-of-str is case-insensitive."""
(0)         def _convert_validator_spec(key, conv):
(4)             if isinstance(conv, list):
(8)                 ignorecase = isinstance(conv, _ignorecase)
(8)                 return ValidateInStrings(key, conv, ignorecase=ignorecase)
(4)             else:
(8)                 return conv
(0)         _validators = {
(4)             "backend":           validate_backend,
(4)             "backend_fallback":  validate_bool,
(4)             "figure.hooks":      validate_stringlist,
(4)             "toolbar":           _validate_toolbar,
(4)             "interactive":       validate_bool,
(4)             "timezone":          validate_string,
(4)             "webagg.port":            validate_int,
(4)             "webagg.address":         validate_string,
(4)             "webagg.open_in_browser": validate_bool,
(4)             "webagg.port_retries":    validate_int,
(4)             "lines.linewidth":       validate_float,  # line width in points
(4)             "lines.linestyle":       _validate_linestyle,  # solid line
(4)             "lines.color":           validate_color,  # first color in color cycle
(4)             "lines.marker":          _validate_marker,  # marker name
(4)             "lines.markerfacecolor": validate_color_or_auto,  # default color
(4)             "lines.markeredgecolor": validate_color_or_auto,  # default color
(4)             "lines.markeredgewidth": validate_float,
(4)             "lines.markersize":      validate_float,  # markersize, in points
(4)             "lines.antialiased":     validate_bool,  # antialiased (no jaggies)
(4)             "lines.dash_joinstyle":  JoinStyle,
(4)             "lines.solid_joinstyle": JoinStyle,
(4)             "lines.dash_capstyle":   CapStyle,
(4)             "lines.solid_capstyle":  CapStyle,
(4)             "lines.dashed_pattern":  validate_floatlist,
(4)             "lines.dashdot_pattern": validate_floatlist,
(4)             "lines.dotted_pattern":  validate_floatlist,
(4)             "lines.scale_dashes":    validate_bool,
(4)             "markers.fillstyle": validate_fillstyle,
(4)             "pcolor.shading": ["auto", "flat", "nearest", "gouraud"],
(4)             "pcolormesh.snap": validate_bool,
(4)             "patch.linewidth":       validate_float,  # line width in points
(4)             "patch.edgecolor":       validate_color,
(4)             "patch.force_edgecolor": validate_bool,
(4)             "patch.facecolor":       validate_color,  # first color in cycle
(4)             "patch.antialiased":     validate_bool,  # antialiased (no jaggies)
(4)             "hatch.color":     validate_color,
(4)             "hatch.linewidth": validate_float,
(4)             "hist.bins": validate_hist_bins,
(4)             "boxplot.notch":       validate_bool,
(4)             "boxplot.vertical":    validate_bool,
(4)             "boxplot.whiskers":    validate_whiskers,
(4)             "boxplot.bootstrap":   validate_int_or_None,
(4)             "boxplot.patchartist": validate_bool,
(4)             "boxplot.showmeans":   validate_bool,
(4)             "boxplot.showcaps":    validate_bool,
(4)             "boxplot.showbox":     validate_bool,
(4)             "boxplot.showfliers":  validate_bool,
(4)             "boxplot.meanline":    validate_bool,
(4)             "boxplot.flierprops.color":           validate_color,
```

```
747: (4)                "boxplot.flierprops.marker":          _validate_marker,
748: (4)                "boxplot.flierprops.markerfacecolor": validate_color_or_auto,
749: (4)                "boxplot.flierprops.markeredgecolor": validate_color,
750: (4)                "boxplot.flierprops.markeredgewidth": validate_float,
751: (4)                "boxplot.flierprops.markersize":      validate_float,
752: (4)                "boxplot.flierprops.linestyle":       _validate_linestyle,
753: (4)                "boxplot.flierprops.linewidth":       validate_float,
754: (4)                "boxplot.boxprops.color":     validate_color,
755: (4)                "boxplot.boxprops.linewidth": validate_float,
756: (4)                "boxplot.boxprops.linestyle": _validate_linestyle,
757: (4)                "boxplot.whiskerprops.color":     validate_color,
758: (4)                "boxplot.whiskerprops.linewidth": validate_float,
759: (4)                "boxplot.whiskerprops.linestyle": _validate_linestyle,
760: (4)                "boxplot.capprops.color":     validate_color,
761: (4)                "boxplot.capprops.linewidth": validate_float,
762: (4)                "boxplot.capprops.linestyle": _validate_linestyle,
763: (4)                "boxplot.medianprops.color":     validate_color,
764: (4)                "boxplot.medianprops.linewidth": validate_float,
765: (4)                "boxplot.medianprops.linestyle": _validate_linestyle,
766: (4)                "boxplot.meanprops.color":           validate_color,
767: (4)                "boxplot.meanprops.marker":          _validate_marker,
768: (4)                "boxplot.meanprops.markerfacecolor": validate_color,
769: (4)                "boxplot.meanprops.markeredgecolor": validate_color,
770: (4)                "boxplot.meanprops.markersize":      validate_float,
771: (4)                "boxplot.meanprops.linestyle":       _validate_linestyle,
772: (4)                "boxplot.meanprops.linewidth":       validate_float,
773: (4)                "font.family":     validate_stringlist,  # used by text object
774: (4)                "font.style":      validate_string,
775: (4)                "font.variant":    validate_string,
776: (4)                "font.stretch":    validate_fontstretch,
777: (4)                "font.weight":     validate_fontweight,
778: (4)                "font.size":       validate_float,  # Base font size in points
779: (4)                "font.serif":      validate_stringlist,
780: (4)                "font.sans-serif": validate_stringlist,
781: (4)                "font.cursive":    validate_stringlist,
782: (4)                "font.fantasy":    validate_stringlist,
783: (4)                "font.monospace":  validate_stringlist,
784: (4)                "text.color":          validate_color,
785: (4)                "text.usetex":         validate_bool,
786: (4)                "text.latex.preamble": validate_string,
787: (4)                "text.hinting":        ["default", "no_autohint", "force_autohint",
788: (28)                                      "no_hinting", "auto", "native", "either", "none"],
789: (4)                "text.hinting_factor": validate_int,
790: (4)                "text.kerning_factor": validate_int,
791: (4)                "text.antialiased":    validate_bool,
792: (4)                "text.parse_math":     validate_bool,
793: (4)                "mathtext.cal":            validate_font_properties,
794: (4)                "mathtext.rm":             validate_font_properties,
795: (4)                "mathtext.tt":             validate_font_properties,
796: (4)                "mathtext.it":             validate_font_properties,
797: (4)                "mathtext.bf":             validate_font_properties,
798: (4)                "mathtext.bfit":           validate_font_properties,
799: (4)                "mathtext.sf":             validate_font_properties,
800: (4)                "mathtext.fontset":        ["dejavusans", "dejavuserif", "cm", "stix",
801: (32)                                          "stixsans", "custom"],
802: (4)                "mathtext.default":        ["rm", "cal", "bfit", "it", "tt", "sf", "bf",
"default",
803: (32)                                          "bb", "frak", "scr", "regular"],
804: (4)                "mathtext.fallback":       _validate_mathtext_fallback,
805: (4)                "image.aspect":              validate_aspect,  # equal, auto, a number
806: (4)                "image.interpolation":       validate_string,
807: (4)                "image.interpolation_stage": ["data", "rgba"],
808: (4)                "image.cmap":                _validate_cmap,  # gray, jet, etc.
809: (4)                "image.lut":                 validate_int,  # lookup table
810: (4)                "image.origin":              ["upper", "lower"],
811: (4)                "image.resample":            validate_bool,
812: (4)                "image.composite_image": validate_bool,
813: (4)                "contour.negative_linestyle": _validate_linestyle,
814: (4)                "contour.corner_mask":        validate_bool,
```

```
(4)                 "contour.linewidth":          validate_float_or_None,
(4)                 "contour.algorithm":          ["mpl2005", "mpl2014", "serial",
"threaded"],
(4)                 "errorbar.capsize": validate_float,
(4)                 "xaxis.labellocation": ["left", "center", "right"],
(4)                 "yaxis.labellocation": ["bottom", "center", "top"],
(4)                 "axes.axisbelow":        validate_axisbelow,
(4)                 "axes.facecolor":        validate_color,  # background color
(4)                 "axes.edgecolor":        validate_color,  # edge color
(4)                 "axes.linewidth":        validate_float,  # edge linewidth
(4)                 "axes.spines.left":      validate_bool,  # Set visibility of axes spines,
(4)                 "axes.spines.right":     validate_bool,  # i.e., the lines around the
chart
(4)                 "axes.spines.bottom":    validate_bool,  # denoting data boundary.
(4)                 "axes.spines.top":       validate_bool,
(4)                 "axes.titlesize":     validate_fontsize,  # Axes title fontsize
(4)                 "axes.titlelocation": ["left", "center", "right"],  # Axes title alignment
(4)                 "axes.titleweight":   validate_fontweight,  # Axes title font weight
(4)                 "axes.titlecolor":    validate_color_or_auto,  # Axes title font color
(4)                 "axes.titley":        validate_float_or_None,
(4)                 "axes.titlepad":      validate_float,
(4)                 "axes.grid":          validate_bool,  # display grid or not
(4)                 "axes.grid.which":    ["minor", "both", "major"],  # which grids are drawn
(4)                 "axes.grid.axis":     ["x", "y", "both"],  # grid type
(4)                 "axes.labelsize":     validate_fontsize,  # fontsize of x & y labels
(4)                 "axes.labelpad":      validate_float,  # space between label and axis
(4)                 "axes.labelweight":   validate_fontweight,  # fontsize of x & y labels
(4)                 "axes.labelcolor":    validate_color,  # color of axis label
(4)                 "axes.formatter.limits": _listify_validator(validate_int, n=2),
(4)                 "axes.formatter.use_locale": validate_bool,
(4)                 "axes.formatter.use_mathtext": validate_bool,
(4)                 "axes.formatter.min_exponent": validate_int,
(4)                 "axes.formatter.useoffset": validate_bool,
(4)                 "axes.formatter.offset_threshold": validate_int,
(4)                 "axes.unicode_minus": validate_bool,
(4)                 "axes.prop_cycle": validate_cycler,
(4)                 "axes.autolimit_mode": ["data", "round_numbers"],
(4)                 "axes.xmargin": _validate_greaterthan_minushalf,  # margin added to xaxis
(4)                 "axes.ymargin": _validate_greaterthan_minushalf,  # margin added to yaxis
(4)                 "axes.zmargin": _validate_greaterthan_minushalf,  # margin added to zaxis
(4)                 "polaraxes.grid":    validate_bool,  # display polar grid or not
(4)                 "axes3d.grid":       validate_bool,  # display 3d grid
(4)                 "axes3d.automargin": validate_bool,  # automatically add margin when
(4)                 "axes3d.xaxis.panecolor":    validate_color,  # 3d background pane
(4)                 "axes3d.yaxis.panecolor":    validate_color,  # 3d background pane
(4)                 "axes3d.zaxis.panecolor":    validate_color,  # 3d background pane
(4)                 "scatter.marker":     _validate_marker,
(4)                 "scatter.edgecolors": validate_string,
(4)                 "date.epoch": _validate_date,
(4)                 "date.autoformatter.year":        validate_string,
(4)                 "date.autoformatter.month":       validate_string,
(4)                 "date.autoformatter.day":         validate_string,
(4)                 "date.autoformatter.hour":        validate_string,
(4)                 "date.autoformatter.minute":      validate_string,
(4)                 "date.autoformatter.second":      validate_string,
(4)                 "date.autoformatter.microsecond": validate_string,
(4)                 'date.converter':          ['auto', 'concise'],
(4)                 'date.interval_multiples': validate_bool,
(4)                 "legend.fancybox": validate_bool,
(4)                 "legend.loc": _validate_legend_loc,
(4)                 "legend.numpoints":      validate_int,
(4)                 "legend.scatterpoints":  validate_int,
(4)                 "legend.fontsize":       validate_fontsize,
(4)                 "legend.title_fontsize": validate_fontsize_None,
(4)                 "legend.labelcolor":     _validate_color_or_linecolor,
(4)                 "legend.markerscale":    validate_float,
(4)                 "legend.shadow":         validate_bool,
(4)                 "legend.frameon":        validate_bool,
(4)                 "legend.framealpha":     validate_float_or_None,
```

```
(4)                 "legend.borderpad":      validate_float,  # units are fontsize
(4)                 "legend.labelspacing":   validate_float,
(4)                 "legend.handlelength":   validate_float,
(4)                 "legend.handleheight":   validate_float,
(4)                 "legend.handletextpad":  validate_float,
(4)                 "legend.borderaxespad":  validate_float,
(4)                 "legend.columnspacing":  validate_float,
(4)                 "legend.facecolor":      validate_color_or_inherit,
(4)                 "legend.edgecolor":      validate_color_or_inherit,
(4)                 "xtick.top":           validate_bool,      # draw ticks on top side
(4)                 "xtick.bottom":        validate_bool,      # draw ticks on bottom side
(4)                 "xtick.labeltop":      validate_bool,      # draw label on top
(4)                 "xtick.labelbottom":   validate_bool,      # draw label on bottom
(4)                 "xtick.major.size":    validate_float,     # major xtick size in points
(4)                 "xtick.minor.size":    validate_float,     # minor xtick size in points
(4)                 "xtick.major.width":   validate_float,     # major xtick width in points
(4)                 "xtick.minor.width":   validate_float,     # minor xtick width in points
(4)                 "xtick.major.pad":     validate_float,     # distance to label in points
(4)                 "xtick.minor.pad":     validate_float,     # distance to label in points
(4)                 "xtick.color":         validate_color,     # color of xticks
(4)                 "xtick.labelcolor":    validate_color_or_inherit,  # color of xtick labels
(4)                 "xtick.minor.visible": validate_bool,      # visibility of minor xticks
(4)                 "xtick.minor.top":     validate_bool,      # draw top minor xticks
(4)                 "xtick.minor.bottom":  validate_bool,      # draw bottom minor xticks
(4)                 "xtick.major.top":     validate_bool,      # draw top major xticks
(4)                 "xtick.major.bottom":  validate_bool,      # draw bottom major xticks
(4)                 "xtick.minor.ndivs":   _validate_minor_tick_ndivs,
(4)                 "xtick.labelsize":     validate_fontsize,  # fontsize of xtick labels
(4)                 "xtick.direction":     ["out", "in", "inout"],  # direction of xticks
(4)                 "xtick.alignment":     ["center", "right", "left"],
(4)                 "ytick.left":          validate_bool,      # draw ticks on left side
(4)                 "ytick.right":         validate_bool,      # draw ticks on right side
(4)                 "ytick.labelleft":     validate_bool,      # draw tick labels on left side
(4)                 "ytick.labelright":    validate_bool,      # draw tick labels on right
side
(4)                 "ytick.major.size":    validate_float,     # major ytick size in points
(4)                 "ytick.minor.size":    validate_float,     # minor ytick size in points
(4)                 "ytick.major.width":   validate_float,     # major ytick width in points
(4)                 "ytick.minor.width":   validate_float,     # minor ytick width in points
(4)                 "ytick.major.pad":     validate_float,     # distance to label in points
(4)                 "ytick.minor.pad":     validate_float,     # distance to label in points
(4)                 "ytick.color":         validate_color,     # color of yticks
(4)                 "ytick.labelcolor":    validate_color_or_inherit,  # color of ytick labels
(4)                 "ytick.minor.visible": validate_bool,      # visibility of minor yticks
(4)                 "ytick.minor.left":    validate_bool,      # draw left minor yticks
(4)                 "ytick.minor.right":   validate_bool,      # draw right minor yticks
(4)                 "ytick.major.left":    validate_bool,      # draw left major yticks
(4)                 "ytick.major.right":   validate_bool,      # draw right major yticks
(4)                 "ytick.minor.ndivs":   _validate_minor_tick_ndivs,
(4)                 "ytick.labelsize":     validate_fontsize,  # fontsize of ytick labels
(4)                 "ytick.direction":     ["out", "in", "inout"],  # direction of yticks
(4)                 "ytick.alignment":     [
(8)                     "center", "top", "bottom", "baseline", "center_baseline"],
(4)                 "grid.color":        validate_color,  # grid color
(4)                 "grid.linestyle":    _validate_linestyle,  # solid
(4)                 "grid.linewidth":    validate_float,     # in points
(4)                 "grid.alpha":        validate_float,
(4)                 "figure.titlesize":   validate_fontsize,
(4)                 "figure.titleweight": validate_fontweight,
(4)                 "figure.labelsize":   validate_fontsize,
(4)                 "figure.labelweight": validate_fontweight,
(4)                 "figure.figsize":          _listify_validator(validate_float, n=2),
(4)                 "figure.dpi":              validate_float,
(4)                 "figure.facecolor":        validate_color,
(4)                 "figure.edgecolor":        validate_color,
(4)                 "figure.frameon":          validate_bool,
(4)                 "figure.autolayout":       validate_bool,
(4)                 "figure.max_open_warning": validate_int,
(4)                 "figure.raise_window":     validate_bool,
```

```
(4)                 "macosx.window_mode":      ["system", "tab", "window"],
(4)                 "figure.subplot.left":   validate_float,
(4)                 "figure.subplot.right":  validate_float,
(4)                 "figure.subplot.bottom": validate_float,
(4)                 "figure.subplot.top":    validate_float,
(4)                 "figure.subplot.wspace": validate_float,
(4)                 "figure.subplot.hspace": validate_float,
(4)                 "figure.constrained_layout.use": validate_bool,  # run constrained_layout?
(4)                 "figure.constrained_layout.hspace": validate_float,
(4)                 "figure.constrained_layout.wspace": validate_float,
(4)                 "figure.constrained_layout.h_pad": validate_float,
(4)                 "figure.constrained_layout.w_pad": validate_float,
(4)                 'savefig.dpi':          validate_dpi,
(4)                 'savefig.facecolor':    validate_color_or_auto,
(4)                 'savefig.edgecolor':    validate_color_or_auto,
(4)                 'savefig.orientation':  ['landscape', 'portrait'],
(4)                 "savefig.format":       validate_string,
(4)                 "savefig.bbox":         validate_bbox,  # "tight", or "standard" (= None)
(4)                 "savefig.pad_inches":   validate_float,
(4)                 "savefig.directory":    _validate_pathlike,
(4)                 "savefig.transparent":  validate_bool,
(4)                 "tk.window_focus": validate_bool,  # Maintain shell focus for TkAgg
(4)                 "ps.papersize":       _validate_papersize,
(4)                 "ps.useafm":          validate_bool,
(4)                 "ps.usedistiller":    validate_ps_distiller,
(4)                 "ps.distiller.res":   validate_int,  # dpi
(4)                 "ps.fonttype":        validate_fonttype,  # 3 (Type3) or 42 (Truetype)
(4)                 "pdf.compression":    validate_int,  # 0-9 compression level; 0 to disable
(4)                 "pdf.inheritcolor":   validate_bool,  # skip color setting commands
(4)                 "pdf.use14corefonts": validate_bool,
(4)                 "pdf.fonttype":       validate_fonttype,  # 3 (Type3) or 42 (Truetype)
(4)                 "pgf.texsystem": ["xelatex", "lualatex", "pdflatex"],  # latex variant
used
(4)                 "pgf.rcfonts":   validate_bool,  # use mpl's rc settings for font config
(4)                 "pgf.preamble":  validate_string,  # custom LaTeX preamble
(4)                 "svg.image_inline": validate_bool,
(4)                 "svg.fonttype": ["none", "path"],  # save text as text ("none") or "paths"
(4)                 "svg.hashsalt": validate_string_or_None,
(4)                 "docstring.hardcopy": validate_bool,
(4)                 "path.simplify":           validate_bool,
(4)                 "path.simplify_threshold": _validate_greaterequal0_lessequal1,
(4)                 "path.snap":               validate_bool,
(4)                 "path.sketch":             validate_sketch,
(4)                 "path.effects":            validate_anylist,
(4)                 "agg.path.chunksize":      validate_int,  # 0 to disable chunking
(4)                 "keymap.fullscreen": validate_stringlist,
(4)                 "keymap.home":       validate_stringlist,
(4)                 "keymap.back":       validate_stringlist,
(4)                 "keymap.forward":    validate_stringlist,
(4)                 "keymap.pan":        validate_stringlist,
(4)                 "keymap.zoom":       validate_stringlist,
(4)                 "keymap.save":       validate_stringlist,
(4)                 "keymap.quit":       validate_stringlist,
(4)                 "keymap.quit_all":   validate_stringlist,  # e.g.: "W", "cmd+W", "Q"
(4)                 "keymap.grid":       validate_stringlist,
(4)                 "keymap.grid_minor": validate_stringlist,
(4)                 "keymap.yscale":     validate_stringlist,
(4)                 "keymap.xscale":     validate_stringlist,
(4)                 "keymap.help":       validate_stringlist,
(4)                 "keymap.copy":       validate_stringlist,
(4)                 "animation.html":         ["html5", "jshtml", "none"],
(4)                 "animation.embed_limit":  validate_float,
(4)                 "animation.writer":       validate_string,
(4)                 "animation.codec":        validate_string,
(4)                 "animation.bitrate":      validate_int,
(4)                 "animation.frame_format": ["png", "jpeg", "tiff", "raw", "rgba", "ppm",
(31)                                           "sgi", "bmp", "pbm", "svg"],
(4)                 "animation.ffmpeg_path":  _validate_pathlike,
(4)                 "animation.ffmpeg_args":  validate_stringlist,
```

```
1018: (4)               "animation.convert_path": _validate_pathlike,
1019: (4)               "animation.convert_args": validate_stringlist,
1020: (4)               "_internal.classic_mode": validate_bool
1021: (0)           }
1022: (0)           _hardcoded_defaults = {  # Defaults not inferred from
1023: (4)               "_internal.classic_mode": False,
1024: (0)           }
1025: (0)           _validators = {k: _convert_validator_spec(k, conv)
1026: (15)                         for k, conv in _validators.items()}
```

File 58 - scale.py:

```
1: (0)              """
2: (0)              Scales define the distribution of data values on an axis, e.g. a log scaling.
3: (0)              They are defined as subclasses of `ScaleBase`.
4: (0)              See also `.axes.Axes.set_xscale` and the scales examples in the documentation.
5: (0)              See :doc:`/gallery/scales/custom_scale` for a full example of defining a
custom
6: (0)              scale.
7: (0)              Matplotlib also supports non-separable transformations that operate on both
8: (0)              `~.axis.Axis` at the same time.  They are known as projections, and defined in
9: (0)              `matplotlib.projections`.
10: (0)             """
11: (0)             import inspect
12: (0)             import textwrap
13: (0)             import numpy as np
14: (0)             import matplotlib as mpl
15: (0)             from matplotlib import _api, _docstring
16: (0)             from matplotlib.ticker import (
17: (4)                 NullFormatter, ScalarFormatter, LogFormatterSciNotation, LogitFormatter,
18: (4)                 NullLocator, LogLocator, AutoLocator, AutoMinorLocator,
19: (4)                 SymmetricalLogLocator, AsinhLocator, LogitLocator)
20: (0)             from matplotlib.transforms import Transform, IdentityTransform
21: (0)             class ScaleBase:
22: (4)                 """
23: (4)                 The base class for all scales.
24: (4)                 Scales are separable transformations, working on a single dimension.
25: (4)                 Subclasses should override
26: (4)                 :attr:`name`
27: (8)                     The scale's name.
28: (4)                 :meth:`get_transform`
29: (8)                     A method returning a `.Transform`, which converts data coordinates to
30: (8)                     scaled coordinates.  This transform should be invertible, so that e.g.
31: (8)                     mouse positions can be converted back to data coordinates.
32: (4)                 :meth:`set_default_locators_and_formatters`
33: (8)                     A method that sets default locators and formatters for an
`~.axis.Axis`
34: (8)                     that uses this scale.
35: (4)                 :meth:`limit_range_for_scale`
36: (8)                     An optional method that "fixes" the axis range to acceptable values,
37: (8)                     e.g. restricting log-scaled axes to positive values.
38: (4)                 """
39: (4)                 def __init__(self, axis):
40: (8)                     r"""
41: (8)                     Construct a new scale.
42: (8)                     Notes
43: (8)                     -----
44: (8)                     The following note is for scale implementers.
45: (8)                     For back-compatibility reasons, scales take an `~matplotlib.axis.Axis`
46: (8)                     object as first argument.  However, this argument should not
47: (8)                     be used: a single scale object should be usable by multiple
48: (8)                     `~matplotlib.axis.Axis`\es at the same time.
49: (8)                     """
50: (4)                 def get_transform(self):
51: (8)                     """
52: (8)                     Return the `.Transform` object associated with this scale.
53: (8)                     """
```

```
 54: (8)                     raise NotImplementedError()
 55: (4)                 def set_default_locators_and_formatters(self, axis):
 56: (8)                     """
 57: (8)                     Set the locators and formatters of *axis* to instances suitable for
 58: (8)                     this scale.
 59: (8)                     """
 60: (8)                     raise NotImplementedError()
 61: (4)                 def limit_range_for_scale(self, vmin, vmax, minpos):
 62: (8)                     """
 63: (8)                     Return the range *vmin*, *vmax*, restricted to the
 64: (8)                     domain supported by this scale (if any).
 65: (8)                     *minpos* should be the minimum positive value in the data.
 66: (8)                     This is used by log scales to determine a minimum value.
 67: (8)                     """
 68: (8)                     return vmin, vmax
 69: (0)             class LinearScale(ScaleBase):
 70: (4)                 """
 71: (4)                 The default linear scale.
 72: (4)                 """
 73: (4)                 name = 'linear'
 74: (4)                 def __init__(self, axis):
 75: (8)                     """
 76: (8)                     """  # noqa: D419
 77: (4)                 def set_default_locators_and_formatters(self, axis):
 78: (8)                     axis.set_major_locator(AutoLocator())
 79: (8)                     axis.set_major_formatter(ScalarFormatter())
 80: (8)                     axis.set_minor_formatter(NullFormatter())
 81: (8)                     if (axis.axis_name == 'x' and mpl.rcParams['xtick.minor.visible'] or
 82: (16)                            axis.axis_name == 'y' and
mpl.rcParams['ytick.minor.visible']):
 83: (12)                        axis.set_minor_locator(AutoMinorLocator())
 84: (8)                     else:
 85: (12)                        axis.set_minor_locator(NullLocator())
 86: (4)                 def get_transform(self):
 87: (8)                     """
 88: (8)                     Return the transform for linear scaling, which is just the
 89: (8)                     `~matplotlib.transforms.IdentityTransform`.
 90: (8)                     """
 91: (8)                     return IdentityTransform()
 92: (0)             class FuncTransform(Transform):
 93: (4)                 """
 94: (4)                 A simple transform that takes and arbitrary function for the
 95: (4)                 forward and inverse transform.
 96: (4)                 """
 97: (4)                 input_dims = output_dims = 1
 98: (4)                 def __init__(self, forward, inverse):
 99: (8)                     """
100: (8)                     Parameters
101: (8)                     ----------
102: (8)                     forward : callable
103: (12)                        The forward function for the transform.  This function must have
104: (12)                        an inverse and, for best behavior, be monotonic.
105: (12)                        It must have the signature::
106: (15)                           def forward(values: array-like) -> array-like
107: (8)                     inverse : callable
108: (12)                        The inverse of the forward function.  Signature as ``forward``.
109: (8)                     """
110: (8)                     super().__init__()
111: (8)                     if callable(forward) and callable(inverse):
112: (12)                        self._forward = forward
113: (12)                        self._inverse = inverse
114: (8)                     else:
115: (12)                        raise ValueError('arguments to FuncTransform must be functions')
116: (4)                 def transform_non_affine(self, values):
117: (8)                     return self._forward(values)
118: (4)                 def inverted(self):
119: (8)                     return FuncTransform(self._inverse, self._forward)
120: (0)             class FuncScale(ScaleBase):
121: (4)                 """
```

```
122: (4)                Provide an arbitrary scale with user-supplied function for the axis.
123: (4)                """
124: (4)                name = 'function'
125: (4)                def __init__(self, axis, functions):
126: (8)                    """
127: (8)                    Parameters
128: (8)                    ----------
129: (8)                    axis : `~matplotlib.axis.Axis`
130: (12)                       The axis for the scale.
131: (8)                    functions : (callable, callable)
132: (12)                       two-tuple of the forward and inverse functions for the scale.
133: (12)                       The forward function must be monotonic.
134: (12)                       Both functions must have the signature::
135: (15)                          def forward(values: array-like) -> array-like
136: (8)                    """
137: (8)                    forward, inverse = functions
138: (8)                    transform = FuncTransform(forward, inverse)
139: (8)                    self._transform = transform
140: (4)                def get_transform(self):
141: (8)                    """Return the `.FuncTransform` associated with this scale."""
142: (8)                    return self._transform
143: (4)                def set_default_locators_and_formatters(self, axis):
144: (8)                    axis.set_major_locator(AutoLocator())
145: (8)                    axis.set_major_formatter(ScalarFormatter())
146: (8)                    axis.set_minor_formatter(NullFormatter())
147: (8)                    if (axis.axis_name == 'x' and mpl.rcParams['xtick.minor.visible'] or
148: (16)                           axis.axis_name == 'y' and
mpl.rcParams['ytick.minor.visible']):
149: (12)                       axis.set_minor_locator(AutoMinorLocator())
150: (8)                    else:
151: (12)                       axis.set_minor_locator(NullLocator())
152: (0)            class LogTransform(Transform):
153: (4)                input_dims = output_dims = 1
154: (4)                def __init__(self, base, nonpositive='clip'):
155: (8)                    super().__init__()
156: (8)                    if base <= 0 or base == 1:
157: (12)                       raise ValueError('The log base cannot be <= 0 or == 1')
158: (8)                    self.base = base
159: (8)                    self._clip = _api.check_getitem(
160: (12)                       {"clip": True, "mask": False}, nonpositive=nonpositive)
161: (4)                def __str__(self):
162: (8)                    return "{}(base={}, nonpositive={!r})".format(
163: (12)                       type(self).__name__, self.base, "clip" if self._clip else "mask")
164: (4)                @_api.rename_parameter("3.8", "a", "values")
165: (4)                def transform_non_affine(self, values):
166: (8)                    with np.errstate(divide="ignore", invalid="ignore"):
167: (12)                       log = {np.e: np.log, 2: np.log2, 10: np.log10}.get(self.base)
168: (12)                       if log:  # If possible, do everything in a single call to NumPy.
169: (16)                           out = log(values)
170: (12)                       else:
171: (16)                           out = np.log(values)
172: (16)                           out /= np.log(self.base)
173: (12)                       if self._clip:
174: (16)                           out[values <= 0] = -1000
175: (8)                    return out
176: (4)                def inverted(self):
177: (8)                    return InvertedLogTransform(self.base)
178: (0)            class InvertedLogTransform(Transform):
179: (4)                input_dims = output_dims = 1
180: (4)                def __init__(self, base):
181: (8)                    super().__init__()
182: (8)                    self.base = base
183: (4)                def __str__(self):
184: (8)                    return f"{type(self).__name__}(base={self.base})"
185: (4)                @_api.rename_parameter("3.8", "a", "values")
186: (4)                def transform_non_affine(self, values):
187: (8)                    return np.power(self.base, values)
188: (4)                def inverted(self):
189: (8)                    return LogTransform(self.base)
```

```
(0)  class LogScale(ScaleBase):
(4)      """
(4)      A standard logarithmic scale.  Care is taken to only plot positive values.
(4)      """
(4)      name = 'log'
(4)      def __init__(self, axis, *, base=10, subs=None, nonpositive="clip"):
(8)          """
(8)          Parameters
(8)          ----------
(8)          axis : `~matplotlib.axis.Axis`
(12)             The axis for the scale.
(8)          base : float, default: 10
(12)             The base of the logarithm.
(8)          nonpositive : {'clip', 'mask'}, default: 'clip'
(12)             Determines the behavior for non-positive values. They can either
(12)             be masked as invalid, or clipped to a very small positive number.
(8)          subs : sequence of int, default: None
(12)             Where to place the subticks between each major tick.  For example,
(12)             in a log10 scale, ``[2, 3, 4, 5, 6, 7, 8, 9]`` will place 8
(12)             logarithmically spaced minor ticks between each major tick.
(8)          """
(8)          self._transform = LogTransform(base, nonpositive)
(8)          self.subs = subs
(4)      base = property(lambda self: self._transform.base)
(4)      def set_default_locators_and_formatters(self, axis):
(8)          axis.set_major_locator(LogLocator(self.base))
(8)          axis.set_major_formatter(LogFormatterSciNotation(self.base))
(8)          axis.set_minor_locator(LogLocator(self.base, self.subs))
(8)          axis.set_minor_formatter(
(12)             LogFormatterSciNotation(self.base,
(36)                                     labelOnlyBase=(self.subs is not None)))
(4)      def get_transform(self):
(8)          """Return the `.LogTransform` associated with this scale."""
(8)          return self._transform
(4)      def limit_range_for_scale(self, vmin, vmax, minpos):
(8)          """Limit the domain to positive values."""
(8)          if not np.isfinite(minpos):
(12)             minpos = 1e-300  # Should rarely (if ever) have a visible effect.
(8)          return (minpos if vmin <= 0 else vmin,
(16)                 minpos if vmax <= 0 else vmax)
(0)  class FuncScaleLog(LogScale):
(4)      """
(4)      Provide an arbitrary scale with user-supplied function for the axis and
(4)      then put on a logarithmic axes.
(4)      """
(4)      name = 'functionlog'
(4)      def __init__(self, axis, functions, base=10):
(8)          """
(8)          Parameters
(8)          ----------
(8)          axis : `~matplotlib.axis.Axis`
(12)             The axis for the scale.
(8)          functions : (callable, callable)
(12)             two-tuple of the forward and inverse functions for the scale.
(12)             The forward function must be monotonic.
(12)             Both functions must have the signature::
(16)                 def forward(values: array-like) -> array-like
(8)          base : float, default: 10
(12)             Logarithmic base of the scale.
(8)          """
(8)          forward, inverse = functions
(8)          self.subs = None
(8)          self._transform = FuncTransform(forward, inverse) + LogTransform(base)
(4)      @property
(4)      def base(self):
(8)          return self._transform._b.base  # Base of the LogTransform.
(4)      def get_transform(self):
(8)          """Return the `.Transform` associated with this scale."""
(8)          return self._transform
```

```
(0)             class SymmetricalLogTransform(Transform):
(4)                 input_dims = output_dims = 1
(4)                 def __init__(self, base, linthresh, linscale):
(8)                     super().__init__()
(8)                     if base <= 1.0:
(12)                        raise ValueError("'base' must be larger than 1")
(8)                     if linthresh <= 0.0:
(12)                        raise ValueError("'linthresh' must be positive")
(8)                     if linscale <= 0.0:
(12)                        raise ValueError("'linscale' must be positive")
(8)                     self.base = base
(8)                     self.linthresh = linthresh
(8)                     self.linscale = linscale
(8)                     self._linscale_adj = (linscale / (1.0 - self.base ** -1))
(8)                     self._log_base = np.log(base)
(4)                 @_api.rename_parameter("3.8", "a", "values")
(4)                 def transform_non_affine(self, values):
(8)                     abs_a = np.abs(values)
(8)                     with np.errstate(divide="ignore", invalid="ignore"):
(12)                        out = np.sign(values) * self.linthresh * (
(16)                            self._linscale_adj +
(16)                            np.log(abs_a / self.linthresh) / self._log_base)
(12)                        inside = abs_a <= self.linthresh
(8)                     out[inside] = values[inside] * self._linscale_adj
(8)                     return out
(4)                 def inverted(self):
(8)                     return InvertedSymmetricalLogTransform(self.base, self.linthresh,
(47)                                                            self.linscale)
(0)             class InvertedSymmetricalLogTransform(Transform):
(4)                 input_dims = output_dims = 1
(4)                 def __init__(self, base, linthresh, linscale):
(8)                     super().__init__()
(8)                     symlog = SymmetricalLogTransform(base, linthresh, linscale)
(8)                     self.base = base
(8)                     self.linthresh = linthresh
(8)                     self.invlinthresh = symlog.transform(linthresh)
(8)                     self.linscale = linscale
(8)                     self._linscale_adj = (linscale / (1.0 - self.base ** -1))
(4)                 @_api.rename_parameter("3.8", "a", "values")
(4)                 def transform_non_affine(self, values):
(8)                     abs_a = np.abs(values)
(8)                     with np.errstate(divide="ignore", invalid="ignore"):
(12)                        out = np.sign(values) * self.linthresh * (
(16)                            np.power(self.base,
(25)                                     abs_a / self.linthresh - self._linscale_adj))
(12)                        inside = abs_a <= self.invlinthresh
(8)                     out[inside] = values[inside] / self._linscale_adj
(8)                     return out
(4)                 def inverted(self):
(8)                     return SymmetricalLogTransform(self.base,
(39)                                                   self.linthresh, self.linscale)
(0)             class SymmetricalLogScale(ScaleBase):
(4)                 """
(4)                 The symmetrical logarithmic scale is logarithmic in both the
(4)                 positive and negative directions from the origin.
(4)                 Since the values close to zero tend toward infinity, there is a
(4)                 need to have a range around zero that is linear.  The parameter
(4)                 *linthresh* allows the user to specify the size of this range
(4)                 (-*linthresh*, *linthresh*).
(4)                 Parameters
(4)                 ----------
(4)                 base : float, default: 10
(8)                     The base of the logarithm.
(4)                 linthresh : float, default: 2
(8)                     Defines the range ``(-x, x)``, within which the plot is linear.
(8)                     This avoids having the plot go to infinity around zero.
(4)                 subs : sequence of int
(8)                     Where to place the subticks between each major tick.
(8)                     For example, in a log10 scale: ``[2, 3, 4, 5, 6, 7, 8, 9]`` will place
```

```
(8)                     8 logarithmically spaced minor ticks between each major tick.
(4)                 linscale : float, optional
(8)                     This allows the linear range ``(-linthresh, linthresh)`` to be
(8)                     stretched relative to the logarithmic range. Its value is the number
of
(8)                     decades to use for each half of the linear range. For example, when
(8)                     *linscale* == 1.0 (the default), the space used for the positive and
(8)                     negative halves of the linear range will be equal to one decade in
(8)                     the logarithmic range.
(4)                 """
(4)                 name = 'symlog'
(4)                 def __init__(self, axis, *, base=10, linthresh=2, subs=None, linscale=1):
(8)                     self._transform = SymmetricalLogTransform(base, linthresh, linscale)
(8)                     self.subs = subs
(4)                 base = property(lambda self: self._transform.base)
(4)                 linthresh = property(lambda self: self._transform.linthresh)
(4)                 linscale = property(lambda self: self._transform.linscale)
(4)                 def set_default_locators_and_formatters(self, axis):
(8)                     axis.set_major_locator(SymmetricalLogLocator(self.get_transform()))
(8)                     axis.set_major_formatter(LogFormatterSciNotation(self.base))
(8)                     axis.set_minor_locator(SymmetricalLogLocator(self.get_transform(),
(53)                                                         self.subs))
(8)                     axis.set_minor_formatter(NullFormatter())
(4)                 def get_transform(self):
(8)                     """Return the `.SymmetricalLogTransform` associated with this
scale."""
(8)                     return self._transform
(0)             class AsinhTransform(Transform):
(4)                 """Inverse hyperbolic-sine transformation used by `.AsinhScale`"""
(4)                 input_dims = output_dims = 1
(4)                 def __init__(self, linear_width):
(8)                     super().__init__()
(8)                     if linear_width <= 0.0:
(12)                        raise ValueError("Scale parameter 'linear_width' " +
(29)                                         "must be strictly positive")
(8)                     self.linear_width = linear_width
(4)                 @_api.rename_parameter("3.8", "a", "values")
(4)                 def transform_non_affine(self, values):
(8)                     return self.linear_width * np.arcsinh(values / self.linear_width)
(4)                 def inverted(self):
(8)                     return InvertedAsinhTransform(self.linear_width)
(0)             class InvertedAsinhTransform(Transform):
(4)                 """Hyperbolic sine transformation used by `.AsinhScale`"""
(4)                 input_dims = output_dims = 1
(4)                 def __init__(self, linear_width):
(8)                     super().__init__()
(8)                     self.linear_width = linear_width
(4)                 @_api.rename_parameter("3.8", "a", "values")
(4)                 def transform_non_affine(self, values):
(8)                     return self.linear_width * np.sinh(values / self.linear_width)
(4)                 def inverted(self):
(8)                     return AsinhTransform(self.linear_width)
(0)             class AsinhScale(ScaleBase):
(4)                 """
(4)                 A quasi-logarithmic scale based on the inverse hyperbolic sine (asinh)
(4)                 For values close to zero, this is essentially a linear scale,
(4)                 but for large magnitude values (either positive or negative)
(4)                 it is asymptotically logarithmic. The transition between these
(4)                 linear and logarithmic regimes is smooth, and has no discontinuities
(4)                 in the function gradient in contrast to
(4)                 the `.SymmetricalLogScale` ("symlog") scale.
(4)                 Specifically, the transformation of an axis coordinate :math:`a` is
(4)                 :math:`a \\rightarrow a_0 \\sinh^{-1} (a / a_0)` where :math:`a_0`
(4)                 is the effective width of the linear region of the transformation.
(4)                 In that region, the transformation is
(4)                 :math:`a \\rightarrow a + \\mathcal{O}(a^3)`.
(4)                 For large values of :math:`a` the transformation behaves as
(4)                 :math:`a \\rightarrow a_0 \\, \\mathrm{sgn}(a) \\ln |a| + \\mathcal{O}
(1)`.
```

```
(4)                 .. note::
(7)                    This API is provisional and may be revised in the future
(7)                    based on early user feedback.
(4)                 """
(4)                 name = 'asinh'
(4)                 auto_tick_multipliers = {
(8)                     3: (2, ),
(8)                     4: (2, ),
(8)                     5: (2, ),
(8)                     8: (2, 4),
(8)                     10: (2, 5),
(8)                     16: (2, 4, 8),
(8)                     64: (4, 16),
(8)                     1024: (256, 512)
(4)                 }
(4)                 def __init__(self, axis, *, linear_width=1.0,
(17)                             base=10, subs='auto', **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     linear_width : float, default: 1
(12)                        The scale parameter (elsewhere referred to as :math:`a_0`)
(12)                        defining the extent of the quasi-linear region,
(12)                        and the coordinate values beyond which the transformation
(12)                        becomes asymptotically logarithmic.
(8)                     base : int, default: 10
(12)                        The number base used for rounding tick locations
(12)                        on a logarithmic scale. If this is less than one,
(12)                        then rounding is to the nearest integer multiple
(12)                        of powers of ten.
(8)                     subs : sequence of int
(12)                        Multiples of the number base used for minor ticks.
(12)                        If set to 'auto', this will use built-in defaults,
(12)                        e.g. (2, 5) for base=10.
(8)                     """
(8)                     super().__init__(axis)
(8)                     self._transform = AsinhTransform(linear_width)
(8)                     self._base = int(base)
(8)                     if subs == 'auto':
(12)                        self._subs = self.auto_tick_multipliers.get(self._base)
(8)                     else:
(12)                        self._subs = subs
(4)                 linear_width = property(lambda self: self._transform.linear_width)
(4)                 def get_transform(self):
(8)                     return self._transform
(4)                 def set_default_locators_and_formatters(self, axis):
(8)                     axis.set(major_locator=AsinhLocator(self.linear_width,
(44)                                                         base=self._base),
(17)                             minor_locator=AsinhLocator(self.linear_width,
(44)                                                         base=self._base,
(44)                                                         subs=self._subs),
(17)                             minor_formatter=NullFormatter())
(8)                     if self._base > 1:
(12)                        axis.set_major_formatter(LogFormatterSciNotation(self._base))
(8)                     else:
(12)                        axis.set_major_formatter('{x:.3g}')
(0)             class LogitTransform(Transform):
(4)                 input_dims = output_dims = 1
(4)                 def __init__(self, nonpositive='mask'):
(8)                     super().__init__()
(8)                     _api.check_in_list(['mask', 'clip'], nonpositive=nonpositive)
(8)                     self._nonpositive = nonpositive
(8)                     self._clip = {"clip": True, "mask": False}[nonpositive]
(4)                 @_api.rename_parameter("3.8", "a", "values")
(4)                 def transform_non_affine(self, values):
(8)                     """logit transform (base 10), masked or clipped"""
(8)                     with np.errstate(divide="ignore", invalid="ignore"):
(12)                        out = np.log10(values / (1 - values))
(8)                     if self._clip:  # See LogTransform for choice of clip value.
```

```
463: (12)                        out[values <= 0] = -1000
464: (12)                        out[1 <= values] = 1000
465: (8)                    return out
466: (4)                def inverted(self):
467: (8)                    return LogisticTransform(self._nonpositive)
468: (4)                def __str__(self):
469: (8)                    return f"{type(self).__name__}({self._nonpositive!r})"
470: (0)            class LogisticTransform(Transform):
471: (4)                input_dims = output_dims = 1
472: (4)                def __init__(self, nonpositive='mask'):
473: (8)                    super().__init__()
474: (8)                    self._nonpositive = nonpositive
475: (4)                @_api.rename_parameter("3.8", "a", "values")
476: (4)                def transform_non_affine(self, values):
477: (8)                    """logistic transform (base 10)"""
478: (8)                    return 1.0 / (1 + 10**(-values))
479: (4)                def inverted(self):
480: (8)                    return LogitTransform(self._nonpositive)
481: (4)                def __str__(self):
482: (8)                    return f"{type(self).__name__}({self._nonpositive!r})"
483: (0)            class LogitScale(ScaleBase):
484: (4)                """
485: (4)                Logit scale for data between zero and one, both excluded.
486: (4)                This scale is similar to a log scale close to zero and to one, and almost
487: (4)                linear around 0.5. It maps the interval ]0, 1[ onto ]-infty, +infty[.
488: (4)                """
489: (4)                name = 'logit'
490: (4)                def __init__(self, axis, nonpositive='mask', *,
491: (17)                            one_half=r"\frac{1}{2}", use_overline=False):
492: (8)                    r"""
493: (8)                    Parameters
494: (8)                    ----------
495: (8)                    axis : `~matplotlib.axis.Axis`
496: (12)                       Currently unused.
497: (8)                    nonpositive : {'mask', 'clip'}
498: (12)                       Determines the behavior for values beyond the open interval ]0,
1[.
499: (12)                       They can either be masked as invalid, or clipped to a number very
500: (12)                       close to 0 or 1.
501: (8)                    use_overline : bool, default: False
502: (12)                       Indicate the usage of survival notation (\overline{x}) in place of
503: (12)                       standard notation (1-x) for probability close to one.
504: (8)                    one_half : str, default: r"\frac{1}{2}"
505: (12)                       The string used for ticks formatter to represent 1/2.
506: (8)                    """
507: (8)                    self._transform = LogitTransform(nonpositive)
508: (8)                    self._use_overline = use_overline
509: (8)                    self._one_half = one_half
510: (4)                def get_transform(self):
511: (8)                    """Return the `.LogitTransform` associated with this scale."""
512: (8)                    return self._transform
513: (4)                def set_default_locators_and_formatters(self, axis):
514: (8)                    axis.set_major_locator(LogitLocator())
515: (8)                    axis.set_major_formatter(
516: (12)                       LogitFormatter(
517: (16)                           one_half=self._one_half,
518: (16)                           use_overline=self._use_overline
519: (12)                       )
520: (8)                    )
521: (8)                    axis.set_minor_locator(LogitLocator(minor=True))
522: (8)                    axis.set_minor_formatter(
523: (12)                       LogitFormatter(
524: (16)                           minor=True,
525: (16)                           one_half=self._one_half,
526: (16)                           use_overline=self._use_overline
527: (12)                       )
528: (8)                    )
529: (4)                def limit_range_for_scale(self, vmin, vmax, minpos):
530: (8)                    """
```

```
531: (8)                    Limit the domain to values between 0 and 1 (excluded).
532: (8)                    """
533: (8)                    if not np.isfinite(minpos):
534: (12)                       minpos = 1e-7  # Should rarely (if ever) have a visible effect.
535: (8)                    return (minpos if vmin <= 0 else vmin,
536: (16)                           1 - minpos if vmax >= 1 else vmax)
537: (0)            _scale_mapping = {
538: (4)                'linear': LinearScale,
539: (4)                'log':    LogScale,
540: (4)                'symlog': SymmetricalLogScale,
541: (4)                'asinh':  AsinhScale,
542: (4)                'logit':  LogitScale,
543: (4)                'function': FuncScale,
544: (4)                'functionlog': FuncScaleLog,
545: (4)                }
546: (0)            def get_scale_names():
547: (4)                """Return the names of the available scales."""
548: (4)                return sorted(_scale_mapping)
549: (0)            def scale_factory(scale, axis, **kwargs):
550: (4)                """
551: (4)                Return a scale class by name.
552: (4)                Parameters
553: (4)                ----------
554: (4)                scale : {%(names)s}
555: (4)                axis : `~matplotlib.axis.Axis`
556: (4)                """
557: (4)                scale_cls = _api.check_getitem(_scale_mapping, scale=scale)
558: (4)                return scale_cls(axis, **kwargs)
559: (0)            if scale_factory.__doc__:
560: (4)                scale_factory.__doc__ = scale_factory.__doc__ % {
561: (8)                    "names": ", ".join(map(repr, get_scale_names()))}
562: (0)            def register_scale(scale_class):
563: (4)                """
564: (4)                Register a new kind of scale.
565: (4)                Parameters
566: (4)                ----------
567: (4)                scale_class : subclass of `ScaleBase`
568: (8)                    The scale to register.
569: (4)                """
570: (4)                _scale_mapping[scale_class.name] = scale_class
571: (0)            def _get_scale_docs():
572: (4)                """
573: (4)                Helper function for generating docstrings related to scales.
574: (4)                """
575: (4)                docs = []
576: (4)                for name, scale_class in _scale_mapping.items():
577: (8)                    docstring = inspect.getdoc(scale_class.__init__) or ""
578: (8)                    docs.extend([
579: (12)                       f"    {name!r}",
580: (12)                       "",
581: (12)                       textwrap.indent(docstring, " " * 8),
582: (12)                       ""
583: (8)                    ])
584: (4)                return "\n".join(docs)
585: (0)            _docstring.interpd.update(
586: (4)                scale_type='{%s}' % ', '.join([repr(x) for x in get_scale_names()]),
587: (4)                scale_docs=_get_scale_docs().rstrip(),
588: (4)                )
```

File 59 - sankey.py:

```
1: (0)              """
2: (0)              Module for creating Sankey diagrams using Matplotlib.
3: (0)              """
4: (0)              import logging
5: (0)              from types import SimpleNamespace
6: (0)              import numpy as np
```

```
(0)  import matplotlib as mpl
(0)  from matplotlib.path import Path
(0)  from matplotlib.patches import PathPatch
(0)  from matplotlib.transforms import Affine2D
(0)  from matplotlib import _docstring
(0)  _log = logging.getLogger(__name__)
(0)  __author__ = "Kevin L. Davies"
(0)  __credits__ = ["Yannick Copin"]
(0)  __license__ = "BSD"
(0)  __version__ = "2011/09/16"
(0)  RIGHT = 0
(0)  UP = 1
(0)  DOWN = 3
(0)  class Sankey:
(4)      """
(4)      Sankey diagram.
(6)        Sankey diagrams are a specific type of flow diagram, in which
(6)        the width of the arrows is shown proportionally to the flow
(6)        quantity.  They are typically used to visualize energy or
(6)        material or cost transfers between processes.
(6)        `Wikipedia (6/1/2011) <https://en.wikipedia.org/wiki/Sankey_diagram>`_
(4)      """
(4)      def __init__(self, ax=None, scale=1.0, unit='', format='%G', gap=0.25,
(17)                  radius=0.1, shoulder=0.03, offset=0.15, head_angle=100,
(17)                  margin=0.4, tolerance=1e-6, **kwargs):
(8)          """
(8)          Create a new Sankey instance.
(8)          The optional arguments listed below are applied to all subdiagrams so
(8)          that there is consistent alignment and formatting.
(8)          In order to draw a complex Sankey diagram, create an instance of
(8)          `Sankey` by calling it without any kwargs::
(12)             sankey = Sankey()
(8)          Then add simple Sankey sub-diagrams::
(12)             sankey.add() # 1
(12)             sankey.add() # 2
(12)             sankey.add() # n
(8)          Finally, create the full diagram::
(12)             sankey.finish()
(8)          Or, instead, simply daisy-chain those calls::
(12)             Sankey().add().add...  .add().finish()
(8)          Other Parameters
(8)          ----------------
(8)          ax : `~matplotlib.axes.Axes`
(12)             Axes onto which the data should be plotted.  If *ax* isn't
(12)             provided, new Axes will be created.
(8)          scale : float
(12)             Scaling factor for the flows.  *scale* sizes the width of the
paths
(12)             in order to maintain proper layout.  The same scale is applied to
(12)             all subdiagrams.  The value should be chosen such that the product
(12)             of the scale and the sum of the inputs is approximately 1.0 (and
(12)             the product of the scale and the sum of the outputs is
(12)             approximately -1.0).
(8)          unit : str
(12)             The physical unit associated with the flow quantities.  If *unit*
(12)             is None, then none of the quantities are labeled.
(8)          format : str or callable
(12)             A Python number formatting string or callable used to label the
(12)             flows with their quantities (i.e., a number times a unit, where
the
(12)             unit is given). If a format string is given, the label will be
(12)             ``format % quantity``. If a callable is given, it will be called
(12)             with ``quantity`` as an argument.
(8)          gap : float
(12)             Space between paths that break in/break away to/from the top or
(12)             bottom.
(8)          radius : float
(12)             Inner radius of the vertical paths.
(8)          shoulder : float
```

```
 74: (12)                        Size of the shoulders of output arrows.
 75: (8)                     offset : float
 76: (12)                        Text offset (from the dip or tip of the arrow).
 77: (8)                     head_angle : float
 78: (12)                        Angle, in degrees, of the arrow heads (and negative of the angle
of
 79: (12)                        the tails).
 80: (8)                     margin : float
 81: (12)                        Minimum space between Sankey outlines and the edge of the plot
 82: (12)                        area.
 83: (8)                     tolerance : float
 84: (12)                        Acceptable maximum of the magnitude of the sum of flows.  The
 85: (12)                        magnitude of the sum of connected flows cannot be greater than
 86: (12)                        *tolerance*.
 87: (8)                     **kwargs
 88: (12)                        Any additional keyword arguments will be passed to `add`, which
 89: (12)                        will create the first subdiagram.
 90: (8)                     See Also
 91: (8)                     --------
 92: (8)                     Sankey.add
 93: (8)                     Sankey.finish
 94: (8)                     Examples
 95: (8)                     --------
 96: (8)                     .. plot:: gallery/specialty_plots/sankey_basics.py
 97: (8)                     """
 98: (8)                     if gap < 0:
 99: (12)                        raise ValueError(
100: (16)                            "'gap' is negative, which is not allowed because it would "
101: (16)                            "cause the paths to overlap")
102: (8)                     if radius > gap:
103: (12)                        raise ValueError(
104: (16)                            "'radius' is greater than 'gap', which is not allowed because
"
105: (16)                            "it would cause the paths to overlap")
106: (8)                     if head_angle < 0:
107: (12)                        raise ValueError(
108: (16)                            "'head_angle' is negative, which is not allowed because it "
109: (16)                            "would cause inputs to look like outputs and vice versa")
110: (8)                     if tolerance < 0:
111: (12)                        raise ValueError(
112: (16)                            "'tolerance' is negative, but it must be a magnitude")
113: (8)                     if ax is None:
114: (12)                        import matplotlib.pyplot as plt
115: (12)                        fig = plt.figure()
116: (12)                        ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[])
117: (8)                     self.diagrams = []
118: (8)                     self.ax = ax
119: (8)                     self.unit = unit
120: (8)                     self.format = format
121: (8)                     self.scale = scale
122: (8)                     self.gap = gap
123: (8)                     self.radius = radius
124: (8)                     self.shoulder = shoulder
125: (8)                     self.offset = offset
126: (8)                     self.margin = margin
127: (8)                     self.pitch = np.tan(np.pi * (1 - head_angle / 180.0) / 2.0)
128: (8)                     self.tolerance = tolerance
129: (8)                     self.extent = np.array((np.inf, -np.inf, np.inf, -np.inf))
130: (8)                     if len(kwargs):
131: (12)                        self.add(**kwargs)
132: (4)                 def _arc(self, quadrant=0, cw=True, radius=1, center=(0, 0)):
133: (8)                     """
134: (8)                     Return the codes and vertices for a rotated, scaled, and translated
135: (8)                     90 degree arc.
136: (8)                     Other Parameters
137: (8)                     ----------------
138: (8)                     quadrant : {0, 1, 2, 3}, default: 0
139: (12)                        Uses 0-based indexing (0, 1, 2, or 3).
140: (8)                     cw : bool, default: True
```

```
141: (12)                        If True, the arc vertices are produced clockwise; counter-
clockwise
142: (12)                        otherwise.
143: (8)                    radius : float, default: 1
144: (12)                        The radius of the arc.
145: (8)                    center : (float, float), default: (0, 0)
146: (12)                        (x, y) tuple of the arc's center.
147: (8)                    """
148: (8)                    ARC_CODES = [Path.LINETO,
149: (21)                                 Path.CURVE4,
150: (21)                                 Path.CURVE4,
151: (21)                                 Path.CURVE4,
152: (21)                                 Path.CURVE4,
153: (21)                                 Path.CURVE4,
154: (21)                                 Path.CURVE4]
155: (8)                    ARC_VERTICES = np.array([[1.00000000e+00, 0.00000000e+00],
156: (33)                                            [1.00000000e+00, 2.65114773e-01],
157: (33)                                            [8.94571235e-01, 5.19642327e-01],
158: (33)                                            [7.07106781e-01, 7.07106781e-01],
159: (33)                                            [5.19642327e-01, 8.94571235e-01],
160: (33)                                            [2.65114773e-01, 1.00000000e+00],
161: (33)                                            [0.00000000e+00, 1.00000000e+00]])
162: (8)                    if quadrant in (0, 2):
163: (12)                       if cw:
164: (16)                           vertices = ARC_VERTICES
165: (12)                       else:
166: (16)                           vertices = ARC_VERTICES[:, ::-1]  # Swap x and y.
167: (8)                    else:  # 1, 3
168: (12)                       if cw:
169: (16)                           vertices = np.column_stack((-ARC_VERTICES[:, 1],
170: (45)                                                       ARC_VERTICES[:, 0]))
171: (12)                       else:
172: (16)                           vertices = np.column_stack((-ARC_VERTICES[:, 0],
173: (45)                                                       ARC_VERTICES[:, 1]))
174: (8)                    if quadrant > 1:
175: (12)                       radius = -radius  # Rotate 180 deg.
176: (8)                    return list(zip(ARC_CODES, radius * vertices +
177: (24)                                   np.tile(center, (ARC_VERTICES.shape[0], 1))))
178: (4)                def _add_input(self, path, angle, flow, length):
179: (8)                    """
180: (8)                    Add an input to a path and return its tip and label locations.
181: (8)                    """
182: (8)                    if angle is None:
183: (12)                       return [0, 0], [0, 0]
184: (8)                    else:
185: (12)                       x, y = path[-1][1]  # Use the last point as a reference.
186: (12)                       dipdepth = (flow / 2) * self.pitch
187: (12)                       if angle == RIGHT:
188: (16)                           x -= length
189: (16)                           dip = [x + dipdepth, y + flow / 2.0]
190: (16)                           path.extend([(Path.LINETO, [x, y]),
191: (29)                                        (Path.LINETO, dip),
192: (29)                                        (Path.LINETO, [x, y + flow]),
193: (29)                                        (Path.LINETO, [x + self.gap, y + flow])])
194: (16)                           label_location = [dip[0] - self.offset, dip[1]]
195: (12)                       else:  # Vertical
196: (16)                           x -= self.gap
197: (16)                           if angle == UP:
198: (20)                               sign = 1
199: (16)                           else:
200: (20)                               sign = -1
201: (16)                           dip = [x - flow / 2, y - sign * (length - dipdepth)]
202: (16)                           if angle == DOWN:
203: (20)                               quadrant = 2
204: (16)                           else:
205: (20)                               quadrant = 1
206: (16)                           if self.radius:
207: (20)                               path.extend(self._arc(quadrant=quadrant,
208: (42)                                                     cw=angle == UP,
```

```
209: (42)                                                   radius=self.radius,
210: (42)                                                   center=(x + self.radius,
211: (50)                                                           y - sign * self.radius)))
212: (16)                           else:
213: (20)                               path.append((Path.LINETO, [x, y]))
214: (16)                           path.extend([(Path.LINETO, [x, y - sign * length]),
215: (29)                                        (Path.LINETO, dip),
216: (29)                                        (Path.LINETO, [x - flow, y - sign * length])])
217: (16)                           path.extend(self._arc(quadrant=quadrant,
218: (38)                                                 cw=angle == DOWN,
219: (38)                                                 radius=flow + self.radius,
220: (38)                                                 center=(x + self.radius,
221: (46)                                                         y - sign * self.radius)))
222: (16)                           path.append((Path.LINETO, [x - flow, y + sign * flow]))
223: (16)                           label_location = [dip[0], dip[1] - sign * self.offset]
224: (12)                       return dip, label_location
225: (4)                 def _add_output(self, path, angle, flow, length):
226: (8)                     """
227: (8)                     Append an output to a path and return its tip and label locations.
228: (8)                     .. note:: *flow* is negative for an output.
229: (8)                     """
230: (8)                     if angle is None:
231: (12)                        return [0, 0], [0, 0]
232: (8)                     else:
233: (12)                        x, y = path[-1][1]  # Use the last point as a reference.
234: (12)                        tipheight = (self.shoulder - flow / 2) * self.pitch
235: (12)                        if angle == RIGHT:
236: (16)                            x += length
237: (16)                            tip = [x + tipheight, y + flow / 2.0]
238: (16)                            path.extend([(Path.LINETO, [x, y]),
239: (29)                                        (Path.LINETO, [x, y + self.shoulder]),
240: (29)                                        (Path.LINETO, tip),
241: (29)                                        (Path.LINETO, [x, y - self.shoulder + flow]),
242: (29)                                        (Path.LINETO, [x, y + flow]),
243: (29)                                        (Path.LINETO, [x - self.gap, y + flow])])
244: (16)                            label_location = [tip[0] + self.offset, tip[1]]
245: (12)                        else:  # Vertical
246: (16)                            x += self.gap
247: (16)                            if angle == UP:
248: (20)                                sign, quadrant = 1, 3
249: (16)                            else:
250: (20)                                sign, quadrant = -1, 0
251: (16)                            tip = [x - flow / 2.0, y + sign * (length + tipheight)]
252: (16)                            if self.radius:
253: (20)                                path.extend(self._arc(quadrant=quadrant,
254: (42)                                                      cw=angle == UP,
255: (42)                                                      radius=self.radius,
256: (42)                                                      center=(x - self.radius,
257: (50)                                                              y + sign * self.radius)))
258: (16)                            else:
259: (20)                                path.append((Path.LINETO, [x, y]))
260: (16)                            path.extend([(Path.LINETO, [x, y + sign * length]),
261: (29)                                        (Path.LINETO, [x - self.shoulder,
262: (44)                                                       y + sign * length]),
263: (29)                                        (Path.LINETO, tip),
264: (29)                                        (Path.LINETO, [x + self.shoulder - flow,
265: (44)                                                       y + sign * length]),
266: (29)                                        (Path.LINETO, [x - flow, y + sign * length])])
267: (16)                            path.extend(self._arc(quadrant=quadrant,
268: (38)                                                  cw=angle == DOWN,
269: (38)                                                  radius=self.radius - flow,
270: (38)                                                  center=(x - self.radius,
271: (46)                                                          y + sign * self.radius)))
272: (16)                            path.append((Path.LINETO, [x - flow, y + sign * flow]))
273: (16)                            label_location = [tip[0], tip[1] + sign * self.offset]
274: (12)                        return tip, label_location
275: (4)                 def _revert(self, path, first_action=Path.LINETO):
276: (8)                     """
277: (8)                     A path is not simply reversible by path[::-1] since the code
```

```
(8)                 specifies an action to take from the **previous** point.
(8)                 """
(8)                 reverse_path = []
(8)                 next_code = first_action
(8)                 for code, position in path[::-1]:
(12)                    reverse_path.append((next_code, position))
(12)                    next_code = code
(8)                 return reverse_path
(4)             @_docstring.dedent_interpd
(4)             def add(self, patchlabel='', flows=None, orientations=None, labels='',
(12)                    trunklength=1.0, pathlengths=0.25, prior=None, connect=(0, 0),
(12)                    rotation=0, **kwargs):
(8)                 """
(8)                 Add a simple Sankey diagram with flows at the same hierarchical level.
(8)                 Parameters
(8)                 ----------
(8)                 patchlabel : str
(12)                    Label to be placed at the center of the diagram.
(12)                    Note that *label* (not *patchlabel*) can be passed as keyword
(12)                    argument to create an entry in the legend.
(8)                 flows : list of float
(12)                    Array of flow values.  By convention, inputs are positive and
(12)                    outputs are negative.
(12)                    Flows are placed along the top of the diagram from the inside out
(12)                    in order of their index within *flows*.  They are placed along the
(12)                    sides of the diagram from the top down and along the bottom from
(12)                    the outside in.
(12)                    If the sum of the inputs and outputs is
(12)                    nonzero, the discrepancy will appear as a cubic Bézier curve along
(12)                    the top and bottom edges of the trunk.
(8)                 orientations : list of {-1, 0, 1}
(12)                    List of orientations of the flows (or a single orientation to be
(12)                    used for all flows).  Valid values are 0 (inputs from
(12)                    the left, outputs to the right), 1 (from and to the top) or -1
(12)                    (from and to the bottom).
(8)                 labels : list of (str or None)
(12)                    List of labels for the flows (or a single label to be used for all
(12)                    flows).  Each label may be *None* (no label), or a labeling
string.
(12)                    If an entry is a (possibly empty) string, then the quantity for
the
(12)                    corresponding flow will be shown below the string.  However, if
(12)                    the *unit* of the main diagram is None, then quantities are never
(12)                    shown, regardless of the value of this argument.
(8)                 trunklength : float
(12)                    Length between the bases of the input and output groups (in
(12)                    data-space units).
(8)                 pathlengths : list of float
(12)                    List of lengths of the vertical arrows before break-in or after
(12)                    break-away.  If a single value is given, then it will be applied
to
(12)                    the first (inside) paths on the top and bottom, and the length of
(12)                    all other arrows will be justified accordingly.  The *pathlengths*
(12)                    are not applied to the horizontal inputs and outputs.
(8)                 prior : int
(12)                    Index of the prior diagram to which this diagram should be
(12)                    connected.
(8)                 connect : (int, int)
(12)                    A (prior, this) tuple indexing the flow of the prior diagram and
(12)                    the flow of this diagram which should be connected.  If this is
the
(12)                    first diagram or *prior* is *None*, *connect* will be ignored.
(8)                 rotation : float
(12)                    Angle of rotation of the diagram in degrees.  The interpretation
of
(12)                    the *orientations* argument will be rotated accordingly (e.g., if
(12)                    *rotation* == 90, an *orientations* entry of 1 means to/from the
(12)                    left).  *rotation* is ignored if this diagram is connected to an
(12)                    existing one (using *prior* and *connect*).
```

```
(8)                     Returns
(8)                     -------
(8)                     Sankey
(12)                        The current `.Sankey` instance.
(8)                     Other Parameters
(8)                     ----------------
(8)                     **kwargs
(11)                       Additional keyword arguments set `matplotlib.patches.PathPatch`
(11)                       properties, listed below.  For example, one may want to use
(11)                       ``fill=False`` or ``label="A legend entry"``.
(8)                     %(Patch:kwdoc)s
(8)                     See Also
(8)                     --------
(8)                     Sankey.finish
(8)                     """
(8)                     flows = np.array([1.0, -1.0]) if flows is None else np.array(flows)
(8)                     n = flows.shape[0]  # Number of flows
(8)                     if rotation is None:
(12)                        rotation = 0
(8)                     else:
(12)                        rotation /= 90.0
(8)                     if orientations is None:
(12)                        orientations = 0
(8)                     try:
(12)                        orientations = np.broadcast_to(orientations, n)
(8)                     except ValueError:
(12)                        raise ValueError(
(16)                            f"The shapes of 'flows' {np.shape(flows)} and 'orientations' "
(16)                            f"{np.shape(orientations)} are incompatible"
(12)                        ) from None
(8)                     try:
(12)                        labels = np.broadcast_to(labels, n)
(8)                     except ValueError:
(12)                        raise ValueError(
(16)                            f"The shapes of 'flows' {np.shape(flows)} and 'labels' "
(16)                            f"{np.shape(labels)} are incompatible"
(12)                        ) from None
(8)                     if trunklength < 0:
(12)                        raise ValueError(
(16)                            "'trunklength' is negative, which is not allowed because it "
(16)                            "would cause poor layout")
(8)                     if abs(np.sum(flows)) > self.tolerance:
(12)                        _log.info("The sum of the flows is nonzero (%f; patchlabel=%r); "
(22)                                  "is the system not at steady state?",
(22)                                  np.sum(flows), patchlabel)
(8)                     scaled_flows = self.scale * flows
(8)                     gain = sum(max(flow, 0) for flow in scaled_flows)
(8)                     loss = sum(min(flow, 0) for flow in scaled_flows)
(8)                     if prior is not None:
(12)                        if prior < 0:
(16)                            raise ValueError("The index of the prior diagram is negative")
(12)                        if min(connect) < 0:
(16)                            raise ValueError(
(20)                                "At least one of the connection indices is negative")
(12)                        if prior >= len(self.diagrams):
(16)                            raise ValueError(
(20)                                f"The index of the prior diagram is {prior}, but there "
(20)                                f"are only {len(self.diagrams)} other diagrams")
(12)                        if connect[0] >= len(self.diagrams[prior].flows):
(16)                            raise ValueError(
(20)                                "The connection index to the source diagram is {}, but "
(20)                                "that diagram has only {} flows".format(
(24)                                    connect[0], len(self.diagrams[prior].flows)))
(12)                        if connect[1] >= n:
(16)                            raise ValueError(
(20)                                f"The connection index to this diagram is {connect[1]}, "
(20)                                f"but this diagram has only {n} flows")
(12)                        if self.diagrams[prior].angles[connect[0]] is None:
(16)                            raise ValueError(
```

```
411: (20)                                f"The connection cannot be made, which may occur if the "
412: (20)                                f"magnitude of flow {connect[0]} of diagram {prior} is "
413: (20)                                f"less than the specified tolerance")
414: (12)                        flow_error = (self.diagrams[prior].flows[connect[0]] +
415: (26)                                      flows[connect[1]])
416: (12)                        if abs(flow_error) >= self.tolerance:
417: (16)                            raise ValueError(
418: (20)                                f"The scaled sum of the connected flows is {flow_error}, "
419: (20)                                f"which is not within the tolerance ({self.tolerance})")
420: (8)                    are_inputs = [None] * n
421: (8)                    for i, flow in enumerate(flows):
422: (12)                        if flow >= self.tolerance:
423: (16)                            are_inputs[i] = True
424: (12)                        elif flow <= -self.tolerance:
425: (16)                            are_inputs[i] = False
426: (12)                        else:
427: (16)                            _log.info(
428: (20)                                "The magnitude of flow %d (%f) is below the tolerance "
429: (20)                                "(%f).\nIt will not be shown, and it cannot be used in a "
430: (20)                                "connection.", i, flow, self.tolerance)
431: (8)                    angles = [None] * n
432: (8)                    for i, (orient, is_input) in enumerate(zip(orientations, are_inputs)):
433: (12)                        if orient == 1:
434: (16)                            if is_input:
435: (20)                                angles[i] = DOWN
436: (16)                            elif is_input is False:
437: (20)                                angles[i] = UP
438: (12)                        elif orient == 0:
439: (16)                            if is_input is not None:
440: (20)                                angles[i] = RIGHT
441: (12)                        else:
442: (16)                            if orient != -1:
443: (20)                                raise ValueError(
444: (24)                                    f"The value of orientations[{i}] is {orient}, "
445: (24)                                    f"but it must be -1, 0, or 1")
446: (16)                            if is_input:
447: (20)                                angles[i] = UP
448: (16)                            elif is_input is False:
449: (20)                                angles[i] = DOWN
450: (8)                    if np.iterable(pathlengths):
451: (12)                        if len(pathlengths) != n:
452: (16)                            raise ValueError(
453: (20)                                f"The lengths of 'flows' ({n}) and 'pathlengths' "
454: (20)                                f"({len(pathlengths)}) are incompatible")
455: (8)                    else:  # Make pathlengths into a list.
456: (12)                        urlength = pathlengths
457: (12)                        ullength = pathlengths
458: (12)                        lrlength = pathlengths
459: (12)                        lllength = pathlengths
460: (12)                        d = dict(RIGHT=pathlengths)
461: (12)                        pathlengths = [d.get(angle, 0) for angle in angles]
462: (12)                        for i, (angle, is_input, flow) in enumerate(zip(angles,
are_inputs,
463: (60)                                                                        scaled_flows)):
464: (16)                            if angle == DOWN and is_input:
465: (20)                                pathlengths[i] = ullength
466: (20)                                ullength += flow
467: (16)                            elif angle == UP and is_input is False:
468: (20)                                pathlengths[i] = urlength
469: (20)                                urlength -= flow  # Flow is negative for outputs.
470: (12)                        for i, (angle, is_input, flow) in enumerate(reversed(list(zip(
471: (18)                              angles, are_inputs, scaled_flows)))):
472: (16)                            if angle == UP and is_input:
473: (20)                                pathlengths[n - i - 1] = lllength
474: (20)                                lllength += flow
475: (16)                            elif angle == DOWN and is_input is False:
476: (20)                                pathlengths[n - i - 1] = lrlength
477: (20)                                lrlength -= flow
478: (12)                        has_left_input = False
```

```
(12)                        for i, (angle, is_input, spec) in enumerate(reversed(list(zip(
(18)                              angles, are_inputs, zip(scaled_flows, pathlengths))))):
(16)                            if angle == RIGHT:
(20)                                if is_input:
(24)                                    if has_left_input:
(28)                                        pathlengths[n - i - 1] = 0
(24)                                    else:
(28)                                        has_left_input = True
(12)                        has_right_output = False
(12)                        for i, (angle, is_input, spec) in enumerate(zip(
(18)                              angles, are_inputs, list(zip(scaled_flows, pathlengths)))):
(16)                            if angle == RIGHT:
(20)                                if is_input is False:
(24)                                    if has_right_output:
(28)                                        pathlengths[i] = 0
(24)                                    else:
(28)                                        has_right_output = True
(8)                     urpath = [(Path.MOVETO, [(self.gap - trunklength / 2.0),  # Upper
right
(33)                                             gain / 2.0]),
(18)                              (Path.LINETO, [(self.gap - trunklength / 2.0) / 2.0,
(33)                                             gain / 2.0]),
(18)                              (Path.CURVE4, [(self.gap - trunklength / 2.0) / 8.0,
(33)                                             gain / 2.0]),
(18)                              (Path.CURVE4, [(trunklength / 2.0 - self.gap) / 8.0,
(33)                                             -loss / 2.0]),
(18)                              (Path.LINETO, [(trunklength / 2.0 - self.gap) / 2.0,
(33)                                             -loss / 2.0]),
(18)                              (Path.LINETO, [(trunklength / 2.0 - self.gap),
(33)                                             -loss / 2.0])]
(8)                     llpath = [(Path.LINETO, [(trunklength / 2.0 - self.gap),  # Lower left
(33)                                             loss / 2.0]),
(18)                              (Path.LINETO, [(trunklength / 2.0 - self.gap) / 2.0,
(33)                                             loss / 2.0]),
(18)                              (Path.CURVE4, [(trunklength / 2.0 - self.gap) / 8.0,
(33)                                             loss / 2.0]),
(18)                              (Path.CURVE4, [(self.gap - trunklength / 2.0) / 8.0,
(33)                                             -gain / 2.0]),
(18)                              (Path.LINETO, [(self.gap - trunklength / 2.0) / 2.0,
(33)                                             -gain / 2.0]),
(18)                              (Path.LINETO, [(self.gap - trunklength / 2.0),
(33)                                             -gain / 2.0])]
(8)                     lrpath = [(Path.LINETO, [(trunklength / 2.0 - self.gap),  # Lower
right
(33)                                             loss / 2.0])]
(8)                     ulpath = [(Path.LINETO, [self.gap - trunklength / 2.0,  # Upper left
(33)                                             gain / 2.0])]
(8)                     tips = np.zeros((n, 2))
(8)                     label_locations = np.zeros((n, 2))
(8)                     for i, (angle, is_input, spec) in enumerate(zip(
(14)                          angles, are_inputs, list(zip(scaled_flows, pathlengths)))):
(12)                        if angle == DOWN and is_input:
(16)                            tips[i, :], label_locations[i, :] = self._add_input(
(20)                                ulpath, angle, *spec)
(12)                        elif angle == UP and is_input is False:
(16)                            tips[i, :], label_locations[i, :] = self._add_output(
(20)                                urpath, angle, *spec)
(8)                     for i, (angle, is_input, spec) in enumerate(reversed(list(zip(
(14)                          angles, are_inputs, list(zip(scaled_flows, pathlengths))))):
(12)                        if angle == UP and is_input:
(16)                            tip, label_location = self._add_input(llpath, angle, *spec)
(16)                            tips[n - i - 1, :] = tip
(16)                            label_locations[n - i - 1, :] = label_location
(12)                        elif angle == DOWN and is_input is False:
(16)                            tip, label_location = self._add_output(lrpath, angle, *spec)
(16)                            tips[n - i - 1, :] = tip
(16)                            label_locations[n - i - 1, :] = label_location
(8)                     has_left_input = False
(8)                     for i, (angle, is_input, spec) in enumerate(reversed(list(zip(
```

```
(14)                        angles, are_inputs, list(zip(scaled_flows, pathlengths))))):
(12)                    if angle == RIGHT and is_input:
(16)                        if not has_left_input:
(20)                            if llpath[-1][1][0] > ulpath[-1][1][0]:
(24)                                llpath.append((Path.LINETO, [ulpath[-1][1][0],
(53)                                                             llpath[-1][1][1]]))
(20)                            has_left_input = True
(16)                        tip, label_location = self._add_input(llpath, angle, *spec)
(16)                        tips[n - i - 1, :] = tip
(16)                        label_locations[n - i - 1, :] = label_location
(8)                 has_right_output = False
(8)                 for i, (angle, is_input, spec) in enumerate(zip(
(14)                      angles, are_inputs, list(zip(scaled_flows, pathlengths)))):
(12)                    if angle == RIGHT and is_input is False:
(16)                        if not has_right_output:
(20)                            if urpath[-1][1][0] < lrpath[-1][1][0]:
(24)                                urpath.append((Path.LINETO, [lrpath[-1][1][0],
(53)                                                             urpath[-1][1][1]]))
(20)                            has_right_output = True
(16)                        tips[i, :], label_locations[i, :] = self._add_output(
(20)                            urpath, angle, *spec)
(8)                 if not has_left_input:
(12)                    ulpath.pop()
(12)                    llpath.pop()
(8)                 if not has_right_output:
(12)                    lrpath.pop()
(12)                    urpath.pop()
(8)                 path = (urpath + self._revert(lrpath) + llpath + self._revert(ulpath)
+
(16)                        [(Path.CLOSEPOLY, urpath[0][1])])
(8)                 codes, vertices = zip(*path)
(8)                 vertices = np.array(vertices)
(8)                 def _get_angle(a, r):
(12)                    if a is None:
(16)                        return None
(12)                    else:
(16)                        return a + r
(8)                 if prior is None:
(12)                    if rotation != 0:  # By default, none of this is needed.
(16)                        angles = [_get_angle(angle, rotation) for angle in angles]
(16)                        rotate = Affine2D().rotate_deg(rotation * 90).transform_affine
(16)                        tips = rotate(tips)
(16)                        label_locations = rotate(label_locations)
(16)                        vertices = rotate(vertices)
(12)                    text = self.ax.text(0, 0, s=patchlabel, ha='center', va='center')
(8)                 else:
(12)                    rotation = (self.diagrams[prior].angles[connect[0]] -
(24)                                angles[connect[1]])
(12)                    angles = [_get_angle(angle, rotation) for angle in angles]
(12)                    rotate = Affine2D().rotate_deg(rotation * 90).transform_affine
(12)                    tips = rotate(tips)
(12)                    offset = self.diagrams[prior].tips[connect[0]] - tips[connect[1]]
(12)                    translate = Affine2D().translate(*offset).transform_affine
(12)                    tips = translate(tips)
(12)                    label_locations = translate(rotate(label_locations))
(12)                    vertices = translate(rotate(vertices))
(12)                    kwds = dict(s=patchlabel, ha='center', va='center')
(12)                    text = self.ax.text(*offset, **kwds)
(8)                 if mpl.rcParams['_internal.classic_mode']:
(12)                    fc = kwargs.pop('fc', kwargs.pop('facecolor', '#bfd1d4'))
(12)                    lw = kwargs.pop('lw', kwargs.pop('linewidth', 0.5))
(8)                 else:
(12)                    fc = kwargs.pop('fc', kwargs.pop('facecolor', None))
(12)                    lw = kwargs.pop('lw', kwargs.pop('linewidth', None))
(8)                 if fc is None:
(12)                    fc = self.ax._get_patches_for_fill.get_next_color()
(8)                 patch = PathPatch(Path(vertices, codes), fc=fc, lw=lw, **kwargs)
(8)                 self.ax.add_patch(patch)
(8)                 texts = []
```

```
614: (8)                    for number, angle, label, location in zip(flows, angles, labels,
615: (50)                                                             label_locations):
616: (12)                       if label is None or angle is None:
617: (16)                           label = ''
618: (12)                       elif self.unit is not None:
619: (16)                           if isinstance(self.format, str):
620: (20)                               quantity = self.format % abs(number) + self.unit
621: (16)                           elif callable(self.format):
622: (20)                               quantity = self.format(number)
623: (16)                           else:
624: (20)                               raise TypeError(
625: (24)                                   'format must be callable or a format string')
626: (16)                           if label != '':
627: (20)                               label += "\n"
628: (16)                           label += quantity
629: (12)                       texts.append(self.ax.text(x=location[0], y=location[1],
630: (38)                                                 s=label,
631: (38)                                                 ha='center', va='center'))
632: (8)                    self.extent = (min(np.min(vertices[:, 0]),
633: (27)                                       np.min(label_locations[:, 0]),
634: (27)                                       self.extent[0]),
635: (23)                                   max(np.max(vertices[:, 0]),
636: (27)                                       np.max(label_locations[:, 0]),
637: (27)                                       self.extent[1]),
638: (23)                                   min(np.min(vertices[:, 1]),
639: (27)                                       np.min(label_locations[:, 1]),
640: (27)                                       self.extent[2]),
641: (23)                                   max(np.max(vertices[:, 1]),
642: (27)                                       np.max(label_locations[:, 1]),
643: (27)                                       self.extent[3]))
644: (8)                    self.diagrams.append(
645: (12)                       SimpleNamespace(patch=patch, flows=flows, angles=angles,
tips=tips,
646: (28)                                        text=text, texts=texts))
647: (8)                    return self
648: (4)                def finish(self):
649: (8)                    """
650: (8)                    Adjust the Axes and return a list of information about the Sankey
651: (8)                    subdiagram(s).
652: (8)                    Returns a list of subdiagrams with the following fields:
653: (8)                    ========
============================================================
654: (8)                    Field     Description
655: (8)                    ========
============================================================
656: (8)                    *patch*   Sankey outline (a `~matplotlib.patches.PathPatch`).
657: (8)                    *flows*   Flow values (positive for input, negative for output).
658: (8)                    *angles*  List of angles of the arrows [deg/90].
659: (18)                             For example, if the diagram has not been rotated,
660: (18)                             an input to the top side has an angle of 3 (DOWN),
661: (18)                             and an output from the top side has an angle of 1 (UP).
662: (18)                             If a flow has been skipped (because its magnitude is less
663: (18)                             than *tolerance*), then its angle will be *None*.
664: (8)                    *tips*    (N, 2)-array of the (x, y) positions of the tips (or "dips")
665: (18)                             of the flow paths.
666: (18)                             If the magnitude of a flow is less the *tolerance* of this
667: (18)                             `Sankey` instance, the flow is skipped and its tip will be
at
668: (18)                             the center of the diagram.
669: (8)                    *text*    `.Text` instance for the diagram label.
670: (8)                    *texts*   List of `.Text` instances for the flow labels.
671: (8)                    ========
============================================================
672: (8)                    See Also
673: (8)                    --------
674: (8)                    Sankey.add
675: (8)                    """
676: (8)                    self.ax.axis([self.extent[0] - self.margin,
677: (22)                                 self.extent[1] + self.margin,
```

```
678: (22)                                self.extent[2] - self.margin,
679: (22)                                self.extent[3] + self.margin])
680: (8)                    self.ax.set_aspect('equal', adjustable='datalim')
681: (8)                    return self.diagrams
```

File 60 - spines.py:

```
1: (0)              from collections.abc import MutableMapping
2: (0)              import functools
3: (0)              import numpy as np
4: (0)              import matplotlib as mpl
5: (0)              from matplotlib import _api, _docstring
6: (0)              from matplotlib.artist import allow_rasterization
7: (0)              import matplotlib.transforms as mtransforms
8: (0)              import matplotlib.patches as mpatches
9: (0)              import matplotlib.path as mpath
10: (0)             class Spine(mpatches.Patch):
11: (4)                 """
12: (4)                 An axis spine -- the line noting the data area boundaries.
13: (4)                 Spines are the lines connecting the axis tick marks and noting the
14: (4)                 boundaries of the data area. They can be placed at arbitrary
15: (4)                 positions. See `~.Spine.set_position` for more information.
16: (4)                 The default position is ``('outward', 0)``.
17: (4)                 Spines are subclasses of `.Patch`, and inherit much of their behavior.
18: (4)                 Spines draw a line, a circle, or an arc depending on if
19: (4)                 `~.Spine.set_patch_line`, `~.Spine.set_patch_circle`, or
20: (4)                 `~.Spine.set_patch_arc` has been called. Line-like is the default.
21: (4)                 For examples see :ref:`spines_examples`.
22: (4)                 """
23: (4)                 def __str__(self):
24: (8)                     return "Spine"
25: (4)                 @_docstring.dedent_interpd
26: (4)                 def __init__(self, axes, spine_type, path, **kwargs):
27: (8)                     """
28: (8)                     Parameters
29: (8)                     ----------
30: (8)                     axes : `~matplotlib.axes.Axes`
31: (12)                        The `~.axes.Axes` instance containing the spine.
32: (8)                     spine_type : str
33: (12)                        The spine type.
34: (8)                     path : `~matplotlib.path.Path`
35: (12)                        The `.Path` instance used to draw the spine.
36: (8)                     Other Parameters
37: (8)                     ----------------
38: (8)                     **kwargs
39: (12)                        Valid keyword arguments are:
40: (12)                        %(Patch:kwdoc)s
41: (8)                     """
42: (8)                     super().__init__(**kwargs)
43: (8)                     self.axes = axes
44: (8)                     self.set_figure(self.axes.figure)
45: (8)                     self.spine_type = spine_type
46: (8)                     self.set_facecolor('none')
47: (8)                     self.set_edgecolor(mpl.rcParams['axes.edgecolor'])
48: (8)                     self.set_linewidth(mpl.rcParams['axes.linewidth'])
49: (8)                     self.set_capstyle('projecting')
50: (8)                     self.axis = None
51: (8)                     self.set_zorder(2.5)
52: (8)                     self.set_transform(self.axes.transData)  # default transform
53: (8)                     self._bounds = None  # default bounds
54: (8)                     self._position = None
55: (8)                     _api.check_isinstance(mpath.Path, path=path)
56: (8)                     self._path = path
57: (8)                     self._patch_type = 'line'
58: (8)                     self._patch_transform = mtransforms.IdentityTransform()
59: (4)                 def set_patch_arc(self, center, radius, theta1, theta2):
60: (8)                     """Set the spine to be arc-like."""
```

```
 61: (8)                     self._patch_type = 'arc'
 62: (8)                     self._center = center
 63: (8)                     self._width = radius * 2
 64: (8)                     self._height = radius * 2
 65: (8)                     self._theta1 = theta1
 66: (8)                     self._theta2 = theta2
 67: (8)                     self._path = mpath.Path.arc(theta1, theta2)
 68: (8)                     self.set_transform(self.axes.transAxes)
 69: (8)                     self.stale = True
 70: (4)                 def set_patch_circle(self, center, radius):
 71: (8)                     """Set the spine to be circular."""
 72: (8)                     self._patch_type = 'circle'
 73: (8)                     self._center = center
 74: (8)                     self._width = radius * 2
 75: (8)                     self._height = radius * 2
 76: (8)                     self.set_transform(self.axes.transAxes)
 77: (8)                     self.stale = True
 78: (4)                 def set_patch_line(self):
 79: (8)                     """Set the spine to be linear."""
 80: (8)                     self._patch_type = 'line'
 81: (8)                     self.stale = True
 82: (4)                 def _recompute_transform(self):
 83: (8)                     """
 84: (8)                     Notes
 85: (8)                     -----
 86: (8)                     This cannot be called until after this has been added to an Axes,
 87: (8)                     otherwise unit conversion will fail. This makes it very important to
 88: (8)                     call the accessor method and not directly access the transformation
 89: (8)                     member variable.
 90: (8)                     """
 91: (8)                     assert self._patch_type in ('arc', 'circle')
 92: (8)                     center = (self.convert_xunits(self._center[0]),
 93: (18)                              self.convert_yunits(self._center[1]))
 94: (8)                     width = self.convert_xunits(self._width)
 95: (8)                     height = self.convert_yunits(self._height)
 96: (8)                     self._patch_transform = mtransforms.Affine2D() \
 97: (12)                        .scale(width * 0.5, height * 0.5) \
 98: (12)                        .translate(*center)
 99: (4)                 def get_patch_transform(self):
 100: (8)                    if self._patch_type in ('arc', 'circle'):
 101: (12)                       self._recompute_transform()
 102: (12)                       return self._patch_transform
 103: (8)                    else:
 104: (12)                       return super().get_patch_transform()
 105: (4)                def get_window_extent(self, renderer=None):
 106: (8)                    """
 107: (8)                    Return the window extent of the spines in display space, including
 108: (8)                    padding for ticks (but not their labels)
 109: (8)                    See Also
 110: (8)                    --------
 111: (8)                    matplotlib.axes.Axes.get_tightbbox
 112: (8)                    matplotlib.axes.Axes.get_window_extent
 113: (8)                    """
 114: (8)                    self._adjust_location()
 115: (8)                    bb = super().get_window_extent(renderer=renderer)
 116: (8)                    if self.axis is None or not self.axis.get_visible():
 117: (12)                       return bb
 118: (8)                    bboxes = [bb]
 119: (8)                    drawn_ticks = self.axis._update_ticks()
 120: (8)                    major_tick = next(iter({*drawn_ticks} & {*self.axis.majorTicks}),
 None)
 121: (8)                    minor_tick = next(iter({*drawn_ticks} & {*self.axis.minorTicks}),
 None)
 122: (8)                    for tick in [major_tick, minor_tick]:
 123: (12)                       if tick is None:
 124: (16)                           continue
 125: (12)                       bb0 = bb.frozen()
 126: (12)                       tickl = tick._size
 127: (12)                       tickdir = tick._tickdir
```

```
(12)                        if tickdir == 'out':
(16)                            padout = 1
(16)                            padin = 0
(12)                        elif tickdir == 'in':
(16)                            padout = 0
(16)                            padin = 1
(12)                        else:
(16)                            padout = 0.5
(16)                            padin = 0.5
(12)                        padout = padout * tickl / 72 * self.figure.dpi
(12)                        padin = padin * tickl / 72 * self.figure.dpi
(12)                        if tick.tick1line.get_visible():
(16)                            if self.spine_type == 'left':
(20)                                bb0.x0 = bb0.x0 - padout
(20)                                bb0.x1 = bb0.x1 + padin
(16)                            elif self.spine_type == 'bottom':
(20)                                bb0.y0 = bb0.y0 - padout
(20)                                bb0.y1 = bb0.y1 + padin
(12)                        if tick.tick2line.get_visible():
(16)                            if self.spine_type == 'right':
(20)                                bb0.x1 = bb0.x1 + padout
(20)                                bb0.x0 = bb0.x0 - padin
(16)                            elif self.spine_type == 'top':
(20)                                bb0.y1 = bb0.y1 + padout
(20)                                bb0.y0 = bb0.y0 - padout
(12)                        bboxes.append(bb0)
(8)                     return mtransforms.Bbox.union(bboxes)
(4)                 def get_path(self):
(8)                     return self._path
(4)                 def _ensure_position_is_set(self):
(8)                     if self._position is None:
(12)                        self._position = ('outward', 0.0)  # in points
(12)                        self.set_position(self._position)
(4)                 def register_axis(self, axis):
(8)                     """
(8)                     Register an axis.
(8)                     An axis should be registered with its corresponding spine from
(8)                     the Axes instance. This allows the spine to clear any axis
(8)                     properties when needed.
(8)                     """
(8)                     self.axis = axis
(8)                     self.stale = True
(4)                 def clear(self):
(8)                     """Clear the current spine."""
(8)                     self._clear()
(8)                     if self.axis is not None:
(12)                        self.axis.clear()
(4)                 def _clear(self):
(8)                     """
(8)                     Clear things directly related to the spine.
(8)                     In this way it is possible to avoid clearing the Axis as well when
calling
(8)                     from library code where it is known that the Axis is cleared
separately.
(8)                     """
(8)                     self._position = None  # clear position
(4)                 def _adjust_location(self):
(8)                     """Automatically set spine bounds to the view interval."""
(8)                     if self.spine_type == 'circle':
(12)                        return
(8)                     if self._bounds is not None:
(12)                        low, high = self._bounds
(8)                     elif self.spine_type in ('left', 'right'):
(12)                        low, high = self.axes.viewLim.intervaly
(8)                     elif self.spine_type in ('top', 'bottom'):
(12)                        low, high = self.axes.viewLim.intervalx
(8)                     else:
(12)                        raise ValueError(f'unknown spine spine_type: {self.spine_type}')
(8)                     if self._patch_type == 'arc':
```

```
(12)                        if self.spine_type in ('bottom', 'top'):
(16)                            try:
(20)                                direction = self.axes.get_theta_direction()
(16)                            except AttributeError:
(20)                                direction = 1
(16)                            try:
(20)                                offset = self.axes.get_theta_offset()
(16)                            except AttributeError:
(20)                                offset = 0
(16)                            low = low * direction + offset
(16)                            high = high * direction + offset
(16)                            if low > high:
(20)                                low, high = high, low
(16)                            self._path = mpath.Path.arc(np.rad2deg(low), np.rad2deg(high))
(16)                            if self.spine_type == 'bottom':
(20)                                rmin, rmax = self.axes.viewLim.intervaly
(20)                                try:
(24)                                    rorigin = self.axes.get_rorigin()
(20)                                except AttributeError:
(24)                                    rorigin = rmin
(20)                                scaled_diameter = (rmin - rorigin) / (rmax - rorigin)
(20)                                self._height = scaled_diameter
(20)                                self._width = scaled_diameter
(12)                        else:
(16)                            raise ValueError('unable to set bounds for spine "%s"' %
(33)                                             self.spine_type)
(8)                     else:
(12)                        v1 = self._path.vertices
(12)                        assert v1.shape == (2, 2), 'unexpected vertices shape'
(12)                        if self.spine_type in ['left', 'right']:
(16)                            v1[0, 1] = low
(16)                            v1[1, 1] = high
(12)                        elif self.spine_type in ['bottom', 'top']:
(16)                            v1[0, 0] = low
(16)                            v1[1, 0] = high
(12)                        else:
(16)                            raise ValueError('unable to set bounds for spine "%s"' %
(33)                                             self.spine_type)
(4)                 @allow_rasterization
(4)                 def draw(self, renderer):
(8)                     self._adjust_location()
(8)                     ret = super().draw(renderer)
(8)                     self.stale = False
(8)                     return ret
(4)                 def set_position(self, position):
(8)                     """
(8)                     Set the position of the spine.
(8)                     Spine position is specified by a 2 tuple of (position type,
(8)                     amount). The position types are:
(8)                     * 'outward': place the spine out from the data area by the specified
(10)                      number of points. (Negative values place the spine inwards.)
(8)                     * 'axes': place the spine at the specified Axes coordinate (0 to 1).
(8)                     * 'data': place the spine at the specified data coordinate.
(8)                     Additionally, shorthand notations define a special positions:
(8)                     * 'center' -> ``('axes', 0.5)``
(8)                     * 'zero' -> ``('data', 0.0)``
(8)                     Examples
(8)                     --------
(8)                     :doc:`/gallery/spines/spine_placement_demo`
(8)                     """
(8)                     if position in ('center', 'zero'):  # special positions
(12)                        pass
(8)                     else:
(12)                        if len(position) != 2:
(16)                            raise ValueError("position should be 'center' or 2-tuple")
(12)                        if position[0] not in ['outward', 'axes', 'data']:
(16)                            raise ValueError("position[0] should be one of 'outward', "
(33)                                             "'axes', or 'data' ")
(8)                     self._position = position
```

```
264: (8)                    self.set_transform(self.get_spine_transform())
265: (8)                    if self.axis is not None:
266: (12)                       self.axis.reset_ticks()
267: (8)                    self.stale = True
268: (4)                def get_position(self):
269: (8)                    """Return the spine position."""
270: (8)                    self._ensure_position_is_set()
271: (8)                    return self._position
272: (4)                def get_spine_transform(self):
273: (8)                    """Return the spine transform."""
274: (8)                    self._ensure_position_is_set()
275: (8)                    position = self._position
276: (8)                    if isinstance(position, str):
277: (12)                       if position == 'center':
278: (16)                           position = ('axes', 0.5)
279: (12)                       elif position == 'zero':
280: (16)                           position = ('data', 0)
281: (8)                    assert len(position) == 2, 'position should be 2-tuple'
282: (8)                    position_type, amount = position
283: (8)                    _api.check_in_list(['axes', 'outward', 'data'],
284: (27)                                      position_type=position_type)
285: (8)                    if self.spine_type in ['left', 'right']:
286: (12)                       base_transform = self.axes.get_yaxis_transform(which='grid')
287: (8)                    elif self.spine_type in ['top', 'bottom']:
288: (12)                       base_transform = self.axes.get_xaxis_transform(which='grid')
289: (8)                    else:
290: (12)                       raise ValueError(f'unknown spine spine_type: {self.spine_type!r}')
291: (8)                    if position_type == 'outward':
292: (12)                       if amount == 0:  # short circuit commonest case
293: (16)                           return base_transform
294: (12)                       else:
295: (16)                           offset_vec = {'left': (-1, 0), 'right': (1, 0),
296: (30)                                         'bottom': (0, -1), 'top': (0, 1),
297: (30)                                         }[self.spine_type]
298: (16)                           offset_dots = amount * np.array(offset_vec) / 72
299: (16)                           return (base_transform
300: (24)                                   + mtransforms.ScaledTranslation(
301: (28)                                       *offset_dots, self.figure.dpi_scale_trans))
302: (8)                    elif position_type == 'axes':
303: (12)                       if self.spine_type in ['left', 'right']:
304: (16)                           return (mtransforms.Affine2D.from_values(0, 0, 0, 1, amount,
0)
305: (24)                                   + base_transform)
306: (12)                       elif self.spine_type in ['bottom', 'top']:
307: (16)                           return (mtransforms.Affine2D.from_values(1, 0, 0, 0, 0,
amount)
308: (24)                                   + base_transform)
309: (8)                    elif position_type == 'data':
310: (12)                       if self.spine_type in ('right', 'top'):
311: (16)                           amount -= 1
312: (12)                       if self.spine_type in ('left', 'right'):
313: (16)                           return mtransforms.blended_transform_factory(
314: (20)                               mtransforms.Affine2D().translate(amount, 0)
315: (20)                               + self.axes.transData,
316: (20)                               self.axes.transData)
317: (12)                       elif self.spine_type in ('bottom', 'top'):
318: (16)                           return mtransforms.blended_transform_factory(
319: (20)                               self.axes.transData,
320: (20)                               mtransforms.Affine2D().translate(0, amount)
321: (20)                               + self.axes.transData)
322: (4)                def set_bounds(self, low=None, high=None):
323: (8)                    """
324: (8)                    Set the spine bounds.
325: (8)                    Parameters
326: (8)                    ----------
327: (8)                    low : float or None, optional
328: (12)                       The lower spine bound. Passing *None* leaves the limit unchanged.
329: (12)                       The bounds may also be passed as the tuple (*low*, *high*) as the
330: (12)                       first positional argument.
```

```
331: (12)                        .. ACCEPTS: (low: float, high: float)
332: (8)                    high : float or None, optional
333: (12)                       The higher spine bound. Passing *None* leaves the limit unchanged.
334: (8)                    """
335: (8)                    if self.spine_type == 'circle':
336: (12)                       raise ValueError(
337: (16)                           'set_bounds() method incompatible with circular spines')
338: (8)                    if high is None and np.iterable(low):
339: (12)                       low, high = low
340: (8)                    old_low, old_high = self.get_bounds() or (None, None)
341: (8)                    if low is None:
342: (12)                       low = old_low
343: (8)                    if high is None:
344: (12)                       high = old_high
345: (8)                    self._bounds = (low, high)
346: (8)                    self.stale = True
347: (4)                def get_bounds(self):
348: (8)                    """Get the bounds of the spine."""
349: (8)                    return self._bounds
350: (4)                @classmethod
351: (4)                def linear_spine(cls, axes, spine_type, **kwargs):
352: (8)                    """Create and return a linear `Spine`."""
353: (8)                    if spine_type == 'left':
354: (12)                       path = mpath.Path([(0.0, 0.999), (0.0, 0.999)])
355: (8)                    elif spine_type == 'right':
356: (12)                       path = mpath.Path([(1.0, 0.999), (1.0, 0.999)])
357: (8)                    elif spine_type == 'bottom':
358: (12)                       path = mpath.Path([(0.999, 0.0), (0.999, 0.0)])
359: (8)                    elif spine_type == 'top':
360: (12)                       path = mpath.Path([(0.999, 1.0), (0.999, 1.0)])
361: (8)                    else:
362: (12)                       raise ValueError('unable to make path for spine "%s"' %
spine_type)
363: (8)                    result = cls(axes, spine_type, path, **kwargs)
364: (8)                    result.set_visible(mpl.rcParams[f'axes.spines.{spine_type}'])
365: (8)                    return result
366: (4)                @classmethod
367: (4)                def arc_spine(cls, axes, spine_type, center, radius, theta1, theta2,
368: (18)                             **kwargs):
369: (8)                    """Create and return an arc `Spine`."""
370: (8)                    path = mpath.Path.arc(theta1, theta2)
371: (8)                    result = cls(axes, spine_type, path, **kwargs)
372: (8)                    result.set_patch_arc(center, radius, theta1, theta2)
373: (8)                    return result
374: (4)                @classmethod
375: (4)                def circular_spine(cls, axes, center, radius, **kwargs):
376: (8)                    """Create and return a circular `Spine`."""
377: (8)                    path = mpath.Path.unit_circle()
378: (8)                    spine_type = 'circle'
379: (8)                    result = cls(axes, spine_type, path, **kwargs)
380: (8)                    result.set_patch_circle(center, radius)
381: (8)                    return result
382: (4)                def set_color(self, c):
383: (8)                    """
384: (8)                    Set the edgecolor.
385: (8)                    Parameters
386: (8)                    ----------
387: (8)                    c : :mpltype:`color`
388: (8)                    Notes
389: (8)                    -----
390: (8)                    This method does not modify the facecolor (which defaults to "none"),
391: (8)                    unlike the `.Patch.set_color` method defined in the parent class.  Use
392: (8)                    `.Patch.set_facecolor` to set the facecolor.
393: (8)                    """
394: (8)                    self.set_edgecolor(c)
395: (8)                    self.stale = True
396: (0)            class SpinesProxy:
397: (4)                """
398: (4)                A proxy to broadcast ``set_*()`` and ``set()`` method calls to contained
```

```
`.Spines`.
399: (4)                The proxy cannot be used for any other operations on its members.
400: (4)                The supported methods are determined dynamically based on the contained
401: (4)                spines. If not all spines support a given method, it's executed only on
402: (4)                the subset of spines that support it.
403: (4)                """
404: (4)                def __init__(self, spine_dict):
405: (8)                    self._spine_dict = spine_dict
406: (4)                def __getattr__(self, name):
407: (8)                    broadcast_targets = [spine for spine in self._spine_dict.values()
408: (29)                                         if hasattr(spine, name)]
409: (8)                    if (name != 'set' and not name.startswith('set_')) or not
broadcast_targets:
410: (12)                       raise AttributeError(
411: (16)                           f"'SpinesProxy' object has no attribute '{name}'")
412: (8)                    def x(_targets, _funcname, *args, **kwargs):
413: (12)                       for spine in _targets:
414: (16)                           getattr(spine, _funcname)(*args, **kwargs)
415: (8)                    x = functools.partial(x, broadcast_targets, name)
416: (8)                    x.__doc__ = broadcast_targets[0].__doc__
417: (8)                    return x
418: (4)                def __dir__(self):
419: (8)                    names = []
420: (8)                    for spine in self._spine_dict.values():
421: (12)                       names.extend(name
422: (25)                                    for name in dir(spine) if name.startswith('set_'))
423: (8)                    return list(sorted(set(names)))
424: (0)            class Spines(MutableMapping):
425: (4)                r"""
426: (4)                The container of all `.Spine`\s in an Axes.
427: (4)                The interface is dict-like mapping names (e.g. 'left') to `.Spine`
objects.
428: (4)                Additionally, it implements some pandas.Series-like features like
accessing
429: (4)                elements by attribute::
430: (8)                    spines['top'].set_visible(False)
431: (8)                    spines.top.set_visible(False)
432: (4)                Multiple spines can be addressed simultaneously by passing a list::
433: (8)                    spines[['top', 'right']].set_visible(False)
434: (4)                Use an open slice to address all spines::
435: (8)                    spines[:].set_visible(False)
436: (4)                The latter two indexing methods will return a `SpinesProxy` that
broadcasts all
437: (4)                ``set_*()`` and ``set()`` calls to its members, but cannot be used for any
other
438: (4)                operation.
439: (4)                """
440: (4)                def __init__(self, **kwargs):
441: (8)                    self._dict = kwargs
442: (4)                @classmethod
443: (4)                def from_dict(cls, d):
444: (8)                    return cls(**d)
445: (4)                def __getstate__(self):
446: (8)                    return self._dict
447: (4)                def __setstate__(self, state):
448: (8)                    self.__init__(**state)
449: (4)                def __getattr__(self, name):
450: (8)                    try:
451: (12)                       return self._dict[name]
452: (8)                    except KeyError:
453: (12)                       raise AttributeError(
454: (16)                           f"'Spines' object does not contain a '{name}' spine")
455: (4)                def __getitem__(self, key):
456: (8)                    if isinstance(key, list):
457: (12)                       unknown_keys = [k for k in key if k not in self._dict]
458: (12)                       if unknown_keys:
459: (16)                           raise KeyError(', '.join(unknown_keys))
460: (12)                       return SpinesProxy({k: v for k, v in self._dict.items()
461: (32)                                           if k in key})
```

```
462: (8)                    if isinstance(key, tuple):
463: (12)                       raise ValueError('Multiple spines must be passed as a single
list')
464: (8)                    if isinstance(key, slice):
465: (12)                       if key.start is None and key.stop is None and key.step is None:
466: (16)                           return SpinesProxy(self._dict)
467: (12)                       else:
468: (16)                           raise ValueError(
469: (20)                               'Spines does not support slicing except for the fully '
470: (20)                               'open slice [:] to access all spines.')
471: (8)                    return self._dict[key]
472: (4)                def __setitem__(self, key, value):
473: (8)                    self._dict[key] = value
474: (4)                def __delitem__(self, key):
475: (8)                    del self._dict[key]
476: (4)                def __iter__(self):
477: (8)                    return iter(self._dict)
478: (4)                def __len__(self):
479: (8)                    return len(self._dict)
```

File 61 - stackplot.py:

```
1: (0)              """
2: (0)              Stacked area plot for 1D arrays inspired by Douglas Y'barbo's stackoverflow
3: (0)              answer:
4: (0)              https://stackoverflow.com/q/2225995/
5: (0)              (https://stackoverflow.com/users/66549/doug)
6: (0)              """
7: (0)              import itertools
8: (0)              import numpy as np
9: (0)              from matplotlib import _api
10: (0)             __all__ = ['stackplot']
11: (0)             def stackplot(axes, x, *args,
12: (14)                          labels=(), colors=None, hatch=None, baseline='zero',
13: (14)                          **kwargs):
14: (4)                 """
15: (4)                 Draw a stacked area plot or a streamgraph.
16: (4)                 Parameters
17: (4)                 ----------
18: (4)                 x : (N,) array-like
19: (4)                 y : (M, N) array-like
20: (8)                     The data is assumed to be unstacked. Each of the following
21: (8)                     calls is legal::
22: (12)                        stackplot(x, y)           # where y has shape (M, N)
23: (12)                        stackplot(x, y1, y2, y3)  # where y1, y2, y3, y4 have length N
24: (4)                 baseline : {'zero', 'sym', 'wiggle', 'weighted_wiggle'}
25: (8)                     Method used to calculate the baseline:
26: (8)                     - ``'zero'``: Constant zero baseline, i.e. a simple stacked plot.
27: (8)                     - ``'sym'``:  Symmetric around zero and is sometimes called
28: (10)                      'ThemeRiver'.
29: (8)                     - ``'wiggle'``: Minimizes the sum of the squared slopes.
30: (8)                     - ``'weighted_wiggle'``: Does the same but weights to account for
31: (10)                      size of each layer. It is also called 'Streamgraph'-layout. More
32: (10)                      details can be found at http://leebyron.com/streamgraph/.
33: (4)                 labels : list of str, optional
34: (8)                     A sequence of labels to assign to each data series. If unspecified,
35: (8)                     then no labels will be applied to artists.
36: (4)                 colors : list of :mpltype:`color`, optional
37: (8)                     A sequence of colors to be cycled through and used to color the
stacked
38: (8)                     areas. The sequence need not be exactly the same length as the number
39: (8)                     of provided *y*, in which case the colors will repeat from the
40: (8)                     beginning.
41: (8)                     If not specified, the colors from the Axes property cycle will be
used.
42: (4)                 hatch : list of str, default: None
43: (8)                     A sequence of hatching styles.  See
```

```
44: (8)                     :doc:`/gallery/shapes_and_collections/hatch_style_reference`.
45: (8)                     The sequence will be cycled through for filling the
46: (8)                     stacked areas from bottom to top.
47: (8)                     It need not be exactly the same length as the number
48: (8)                     of provided *y*, in which case the styles will repeat from the
49: (8)                     beginning.
50: (8)                     .. versionadded:: 3.9
51: (11)                       Support for list input
52: (4)                 data : indexable object, optional
53: (8)                     DATA_PARAMETER_PLACEHOLDER
54: (4)                 **kwargs
55: (8)                     All other keyword arguments are passed to `.Axes.fill_between`.
56: (4)                 Returns
57: (4)                 -------
58: (4)                 list of `.PolyCollection`
59: (8)                     A list of `.PolyCollection` instances, one for each element in the
60: (8)                     stacked area plot.
61: (4)                 """
62: (4)                 y = np.vstack(args)
63: (4)                 labels = iter(labels)
64: (4)                 if colors is not None:
65: (8)                     colors = itertools.cycle(colors)
66: (4)                 else:
67: (8)                     colors = (axes._get_lines.get_next_color() for _ in y)
68: (4)                 if hatch is None or isinstance(hatch, str):
69: (8)                     hatch = itertools.cycle([hatch])
70: (4)                 else:
71: (8)                     hatch = itertools.cycle(hatch)
72: (4)                 stack = np.cumsum(y, axis=0, dtype=np.promote_types(y.dtype, np.float32))
73: (4)                 _api.check_in_list(['zero', 'sym', 'wiggle', 'weighted_wiggle'],
74: (23)                                   baseline=baseline)
75: (4)                 if baseline == 'zero':
76: (8)                     first_line = 0.
77: (4)                 elif baseline == 'sym':
78: (8)                     first_line = -np.sum(y, 0) * 0.5
79: (8)                     stack += first_line[None, :]
80: (4)                 elif baseline == 'wiggle':
81: (8)                     m = y.shape[0]
82: (8)                     first_line = (y * (m - 0.5 - np.arange(m)[:, None])).sum(0)
83: (8)                     first_line /= -m
84: (8)                     stack += first_line
85: (4)                 elif baseline == 'weighted_wiggle':
86: (8)                     total = np.sum(y, 0)
87: (8)                     inv_total = np.zeros_like(total)
88: (8)                     mask = total > 0
89: (8)                     inv_total[mask] = 1.0 / total[mask]
90: (8)                     increase = np.hstack((y[:, 0:1], np.diff(y)))
91: (8)                     below_size = total - stack
92: (8)                     below_size += 0.5 * y
93: (8)                     move_up = below_size * inv_total
94: (8)                     move_up[:, 0] = 0.5
95: (8)                     center = (move_up - 0.5) * increase
96: (8)                     center = np.cumsum(center.sum(0))
97: (8)                     first_line = center - 0.5 * total
98: (8)                     stack += first_line
99: (4)                 coll = axes.fill_between(x, first_line, stack[0, :],
100: (29)                                        facecolor=next(colors),
101: (29)                                        hatch=next(hatch),
102: (29)                                        label=next(labels, None),
103: (29)                                        **kwargs)
104: (4)                coll.sticky_edges.y[:] = [0]
105: (4)                r = [coll]
106: (4)                for i in range(len(y) - 1):
107: (8)                    r.append(axes.fill_between(x, stack[i, :], stack[i + 1, :],
108: (35)                                              facecolor=next(colors),
109: (35)                                              hatch=next(hatch),
110: (35)                                              label=next(labels, None),
111: (35)                                              **kwargs))
112: (4)                return r
```

```
----------------------------------------

File 62 - streamplot.py:

1: (0)              """
2: (0)              Streamline plotting for 2D vector fields.
3: (0)              """
4: (0)              import numpy as np
5: (0)              import matplotlib as mpl
6: (0)              from matplotlib import _api, cm, patches
7: (0)              import matplotlib.colors as mcolors
8: (0)              import matplotlib.collections as mcollections
9: (0)              import matplotlib.lines as mlines
10: (0)             __all__ = ['streamplot']
11: (0)             def streamplot(axes, x, y, u, v, density=1, linewidth=None, color=None,
12: (15)                           cmap=None, norm=None, arrowsize=1, arrowstyle='-|>',
13: (15)                           minlength=0.1, transform=None, zorder=None, start_points=None,
14: (15)                           maxlength=4.0, integration_direction='both',
15: (15)                           broken_streamlines=True):
16: (4)                 """
17: (4)                 Draw streamlines of a vector flow.
18: (4)                 Parameters
19: (4)                 ----------
20: (4)                 x, y : 1D/2D arrays
21: (8)                     Evenly spaced strictly increasing arrays to make a grid.  If 2D, all
22: (8)                     rows of *x* must be equal and all columns of *y* must be equal; i.e.,
23: (8)                     they must be as if generated by ``np.meshgrid(x_1d, y_1d)``.
24: (4)                 u, v : 2D arrays
25: (8)                     *x* and *y*-velocities. The number of rows and columns must match
26: (8)                     the length of *y* and *x*, respectively.
27: (4)                 density : float or (float, float)
28: (8)                     Controls the closeness of streamlines. When ``density = 1``, the
domain

29: (8)                     is divided into a 30x30 grid. *density* linearly scales this grid.
30: (8)                     Each cell in the grid can have, at most, one traversing streamline.
31: (8)                     For different densities in each direction, use a tuple
32: (8)                     (density_x, density_y).
33: (4)                 linewidth : float or 2D array
34: (8)                     The width of the streamlines. With a 2D array the line width can be
35: (8)                     varied across the grid. The array must have the same shape as *u*
36: (8)                     and *v*.
37: (4)                 color : :mpltype:`color` or 2D array
38: (8)                     The streamline color. If given an array, its values are converted to
39: (8)                     colors using *cmap* and *norm*.  The array must have the same shape
40: (8)                     as *u* and *v*.
41: (4)                 cmap, norm
42: (8)                     Data normalization and colormapping parameters for *color*; only used
43: (8)                     if *color* is an array of floats. See `~.Axes.imshow` for a detailed
44: (8)                     description.
45: (4)                 arrowsize : float
46: (8)                     Scaling factor for the arrow size.
47: (4)                 arrowstyle : str
48: (8)                     Arrow style specification.
49: (8)                     See `~matplotlib.patches.FancyArrowPatch`.
50: (4)                 minlength : float
51: (8)                     Minimum length of streamline in axes coordinates.
52: (4)                 start_points : (N, 2) array
53: (8)                     Coordinates of starting points for the streamlines in data coordinates
54: (8)                     (the same coordinates as the *x* and *y* arrays).
55: (4)                 zorder : float
56: (8)                     The zorder of the streamlines and arrows.
57: (8)                     Artists with lower zorder values are drawn first.
58: (4)                 maxlength : float
59: (8)                     Maximum length of streamline in axes coordinates.
60: (4)                 integration_direction : {'forward', 'backward', 'both'}, default: 'both'
61: (8)                     Integrate the streamline in forward, backward or both directions.
62: (4)                 data : indexable object, optional
63: (8)                     DATA_PARAMETER_PLACEHOLDER
```

```
(4)                 broken_streamlines : boolean, default: True
(8)                     If False, forces streamlines to continue until they
(8)                     leave the plot domain.  If True, they may be terminated if they
(8)                     come too close to another streamline.
(4)                 Returns
(4)                 -------
(4)                 StreamplotSet
(8)                     Container object with attributes
(8)                     - ``lines``: `.LineCollection` of streamlines
(8)                     - ``arrows``: `.PatchCollection` containing `.FancyArrowPatch`
(10)                      objects representing the arrows half-way along streamlines.
(8)                     This container will probably change in the future to allow changes
(8)                     to the colormap, alpha, etc. for both lines and arrows, but these
(8)                     changes should be backward compatible.
(4)                 """
(4)                 grid = Grid(x, y)
(4)                 mask = StreamMask(density)
(4)                 dmap = DomainMap(grid, mask)
(4)                 if zorder is None:
(8)                     zorder = mlines.Line2D.zorder
(4)                 if transform is None:
(8)                     transform = axes.transData
(4)                 if color is None:
(8)                     color = axes._get_lines.get_next_color()
(4)                 if linewidth is None:
(8)                     linewidth = mpl.rcParams['lines.linewidth']
(4)                 line_kw = {}
(4)                 arrow_kw = dict(arrowstyle=arrowstyle, mutation_scale=10 * arrowsize)
(4)                 _api.check_in_list(['both', 'forward', 'backward'],
(23)                                   integration_direction=integration_direction)
(4)                 if integration_direction == 'both':
(8)                     maxlength /= 2.
(4)                 use_multicolor_lines = isinstance(color, np.ndarray)
(4)                 if use_multicolor_lines:
(8)                     if color.shape != grid.shape:
(12)                        raise ValueError("If 'color' is given, it must match the shape of
"
(29)                                         "the (x, y) grid")
(8)                     line_colors = [[]]  # Empty entry allows concatenation of zero arrays.
(8)                     color = np.ma.masked_invalid(color)
(4)                 else:
(8)                     line_kw['color'] = color
(8)                     arrow_kw['color'] = color
(4)                 if isinstance(linewidth, np.ndarray):
(8)                     if linewidth.shape != grid.shape:
(12)                        raise ValueError("If 'linewidth' is given, it must match the "
(29)                                         "shape of the (x, y) grid")
(8)                     line_kw['linewidth'] = []
(4)                 else:
(8)                     line_kw['linewidth'] = linewidth
(8)                     arrow_kw['linewidth'] = linewidth
(4)                 line_kw['zorder'] = zorder
(4)                 arrow_kw['zorder'] = zorder
(4)                 if u.shape != grid.shape or v.shape != grid.shape:
(8)                     raise ValueError("'u' and 'v' must match the shape of the (x, y)
grid")
(4)                 u = np.ma.masked_invalid(u)
(4)                 v = np.ma.masked_invalid(v)
(4)                 integrate = _get_integrator(u, v, dmap, minlength, maxlength,
(32)                                            integration_direction)
(4)                 trajectories = []
(4)                 if start_points is None:
(8)                     for xm, ym in _gen_starting_points(mask.shape):
(12)                        if mask[ym, xm] == 0:
(16)                            xg, yg = dmap.mask2grid(xm, ym)
(16)                            t = integrate(xg, yg, broken_streamlines)
(16)                            if t is not None:
(20)                                trajectories.append(t)
(4)                 else:
```

```
(8)                     sp2 = np.asanyarray(start_points, dtype=float).copy()
(8)                     for xs, ys in sp2:
(12)                        if not (grid.x_origin <= xs <= grid.x_origin + grid.width and
(20)                                grid.y_origin <= ys <= grid.y_origin + grid.height):
(16)                            raise ValueError(f"Starting point ({xs}, {ys}) outside of "
(33)                                             "data boundaries")
(8)                     sp2[:, 0] -= grid.x_origin
(8)                     sp2[:, 1] -= grid.y_origin
(8)                     for xs, ys in sp2:
(12)                        xg, yg = dmap.data2grid(xs, ys)
(12)                        xg = np.clip(xg, 0, grid.nx - 1)
(12)                        yg = np.clip(yg, 0, grid.ny - 1)
(12)                        t = integrate(xg, yg, broken_streamlines)
(12)                        if t is not None:
(16)                            trajectories.append(t)
(4)                 if use_multicolor_lines:
(8)                     if norm is None:
(12)                        norm = mcolors.Normalize(color.min(), color.max())
(8)                     cmap = cm._ensure_cmap(cmap)
(4)                 streamlines = []
(4)                 arrows = []
(4)                 for t in trajectories:
(8)                     tgx, tgy = t.T
(8)                     tx, ty = dmap.grid2data(tgx, tgy)
(8)                     tx += grid.x_origin
(8)                     ty += grid.y_origin
(8)                     if isinstance(linewidth, np.ndarray) or use_multicolor_lines:
(12)                        points = np.transpose([tx, ty]).reshape(-1, 1, 2)
(12)                        streamlines.extend(np.hstack([points[:-1], points[1:]]))
(8)                     else:
(12)                        points = np.transpose([tx, ty])
(12)                        streamlines.append(points)
(8)                     s = np.cumsum(np.hypot(np.diff(tx), np.diff(ty)))
(8)                     n = np.searchsorted(s, s[-1] / 2.)
(8)                     arrow_tail = (tx[n], ty[n])
(8)                     arrow_head = (np.mean(tx[n:n + 2]), np.mean(ty[n:n + 2]))
(8)                     if isinstance(linewidth, np.ndarray):
(12)                        line_widths = interpgrid(linewidth, tgx, tgy)[:-1]
(12)                        line_kw['linewidth'].extend(line_widths)
(12)                        arrow_kw['linewidth'] = line_widths[n]
(8)                     if use_multicolor_lines:
(12)                        color_values = interpgrid(color, tgx, tgy)[:-1]
(12)                        line_colors.append(color_values)
(12)                        arrow_kw['color'] = cmap(norm(color_values[n]))
(8)                     p = patches.FancyArrowPatch(
(12)                        arrow_tail, arrow_head, transform=transform, **arrow_kw)
(8)                     arrows.append(p)
(4)                 lc = mcollections.LineCollection(
(8)                     streamlines, transform=transform, **line_kw)
(4)                 lc.sticky_edges.x[:] = [grid.x_origin, grid.x_origin + grid.width]
(4)                 lc.sticky_edges.y[:] = [grid.y_origin, grid.y_origin + grid.height]
(4)                 if use_multicolor_lines:
(8)                     lc.set_array(np.ma.hstack(line_colors))
(8)                     lc.set_cmap(cmap)
(8)                     lc.set_norm(norm)
(4)                 axes.add_collection(lc)
(4)                 ac = mcollections.PatchCollection(arrows)
(4)                 for p in arrows:
(8)                     axes.add_patch(p)
(4)                 axes.autoscale_view()
(4)                 stream_container = StreamplotSet(lc, ac)
(4)                 return stream_container
(0)             class StreamplotSet:
(4)                 def __init__(self, lines, arrows):
(8)                     self.lines = lines
(8)                     self.arrows = arrows
(0)             class DomainMap:
(4)                 """
(4)                 Map representing different coordinate systems.
```

```
200: (4)                Coordinate definitions:
201: (4)                * axes-coordinates goes from 0 to 1 in the domain.
202: (4)                * data-coordinates are specified by the input x-y coordinates.
203: (4)                * grid-coordinates goes from 0 to N and 0 to M for an N x M grid,
204: (6)                  where N and M match the shape of the input data.
205: (4)                * mask-coordinates goes from 0 to N and 0 to M for an N x M mask,
206: (6)                  where N and M are user-specified to control the density of streamlines.
207: (4)                This class also has methods for adding trajectories to the StreamMask.
208: (4)                Before adding a trajectory, run `start_trajectory` to keep track of
regions
209: (4)                crossed by a given trajectory. Later, if you decide the trajectory is bad
210: (4)                (e.g., if the trajectory is very short) just call `undo_trajectory`.
211: (4)                """
212: (4)                def __init__(self, grid, mask):
213: (8)                    self.grid = grid
214: (8)                    self.mask = mask
215: (8)                    self.x_grid2mask = (mask.nx - 1) / (grid.nx - 1)
216: (8)                    self.y_grid2mask = (mask.ny - 1) / (grid.ny - 1)
217: (8)                    self.x_mask2grid = 1. / self.x_grid2mask
218: (8)                    self.y_mask2grid = 1. / self.y_grid2mask
219: (8)                    self.x_data2grid = 1. / grid.dx
220: (8)                    self.y_data2grid = 1. / grid.dy
221: (4)                def grid2mask(self, xi, yi):
222: (8)                    """Return nearest space in mask-coords from given grid-coords."""
223: (8)                    return round(xi * self.x_grid2mask), round(yi * self.y_grid2mask)
224: (4)                def mask2grid(self, xm, ym):
225: (8)                    return xm * self.x_mask2grid, ym * self.y_mask2grid
226: (4)                def data2grid(self, xd, yd):
227: (8)                    return xd * self.x_data2grid, yd * self.y_data2grid
228: (4)                def grid2data(self, xg, yg):
229: (8)                    return xg / self.x_data2grid, yg / self.y_data2grid
230: (4)                def start_trajectory(self, xg, yg, broken_streamlines=True):
231: (8)                    xm, ym = self.grid2mask(xg, yg)
232: (8)                    self.mask._start_trajectory(xm, ym, broken_streamlines)
233: (4)                def reset_start_point(self, xg, yg):
234: (8)                    xm, ym = self.grid2mask(xg, yg)
235: (8)                    self.mask._current_xy = (xm, ym)
236: (4)                def update_trajectory(self, xg, yg, broken_streamlines=True):
237: (8)                    if not self.grid.within_grid(xg, yg):
238: (12)                       raise InvalidIndexError
239: (8)                    xm, ym = self.grid2mask(xg, yg)
240: (8)                    self.mask._update_trajectory(xm, ym, broken_streamlines)
241: (4)                def undo_trajectory(self):
242: (8)                    self.mask._undo_trajectory()
243: (0)            class Grid:
244: (4)                """Grid of data."""
245: (4)                def __init__(self, x, y):
246: (8)                    if np.ndim(x) == 1:
247: (12)                       pass
248: (8)                    elif np.ndim(x) == 2:
249: (12)                       x_row = x[0]
250: (12)                       if not np.allclose(x_row, x):
251: (16)                           raise ValueError("The rows of 'x' must be equal")
252: (12)                       x = x_row
253: (8)                    else:
254: (12)                       raise ValueError("'x' can have at maximum 2 dimensions")
255: (8)                    if np.ndim(y) == 1:
256: (12)                       pass
257: (8)                    elif np.ndim(y) == 2:
258: (12)                       yt = np.transpose(y)  # Also works for nested lists.
259: (12)                       y_col = yt[0]
260: (12)                       if not np.allclose(y_col, yt):
261: (16)                           raise ValueError("The columns of 'y' must be equal")
262: (12)                       y = y_col
263: (8)                    else:
264: (12)                       raise ValueError("'y' can have at maximum 2 dimensions")
265: (8)                    if not (np.diff(x) > 0).all():
266: (12)                       raise ValueError("'x' must be strictly increasing")
267: (8)                    if not (np.diff(y) > 0).all():
```

```
268: (12)                        raise ValueError("'y' must be strictly increasing")
269: (8)                    self.nx = len(x)
270: (8)                    self.ny = len(y)
271: (8)                    self.dx = x[1] - x[0]
272: (8)                    self.dy = y[1] - y[0]
273: (8)                    self.x_origin = x[0]
274: (8)                    self.y_origin = y[0]
275: (8)                    self.width = x[-1] - x[0]
276: (8)                    self.height = y[-1] - y[0]
277: (8)                    if not np.allclose(np.diff(x), self.width / (self.nx - 1)):
278: (12)                       raise ValueError("'x' values must be equally spaced")
279: (8)                    if not np.allclose(np.diff(y), self.height / (self.ny - 1)):
280: (12)                       raise ValueError("'y' values must be equally spaced")
281: (4)                @property
282: (4)                def shape(self):
283: (8)                    return self.ny, self.nx
284: (4)                def within_grid(self, xi, yi):
285: (8)                    """Return whether (*xi*, *yi*) is a valid index of the grid."""
286: (8)                    return 0 <= xi <= self.nx - 1 and 0 <= yi <= self.ny - 1
287: (0)            class StreamMask:
288: (4)                """
289: (4)                Mask to keep track of discrete regions crossed by streamlines.
290: (4)                The resolution of this grid determines the approximate spacing between
291: (4)                trajectories. Streamlines are only allowed to pass through zeroed cells:
292: (4)                When a streamline enters a cell, that cell is set to 1, and no new
293: (4)                streamlines are allowed to enter.
294: (4)                """
295: (4)                def __init__(self, density):
296: (8)                    try:
297: (12)                       self.nx, self.ny = (30 * np.broadcast_to(density, 2)).astype(int)
298: (8)                    except ValueError as err:
299: (12)                       raise ValueError("'density' must be a scalar or be of length "
300: (29)                                        "2") from err
301: (8)                    if self.nx < 0 or self.ny < 0:
302: (12)                       raise ValueError("'density' must be positive")
303: (8)                    self._mask = np.zeros((self.ny, self.nx))
304: (8)                    self.shape = self._mask.shape
305: (8)                    self._current_xy = None
306: (4)                def __getitem__(self, args):
307: (8)                    return self._mask[args]
308: (4)                def _start_trajectory(self, xm, ym, broken_streamlines=True):
309: (8)                    """Start recording streamline trajectory"""
310: (8)                    self._traj = []
311: (8)                    self._update_trajectory(xm, ym, broken_streamlines)
312: (4)                def _undo_trajectory(self):
313: (8)                    """Remove current trajectory from mask"""
314: (8)                    for t in self._traj:
315: (12)                       self._mask[t] = 0
316: (4)                def _update_trajectory(self, xm, ym, broken_streamlines=True):
317: (8)                    """
318: (8)                    Update current trajectory position in mask.
319: (8)                    If the new position has already been filled, raise
`InvalidIndexError`.
320: (8)                    """
321: (8)                    if self._current_xy != (xm, ym):
322: (12)                       if self[ym, xm] == 0:
323: (16)                           self._traj.append((ym, xm))
324: (16)                           self._mask[ym, xm] = 1
325: (16)                           self._current_xy = (xm, ym)
326: (12)                       else:
327: (16)                           if broken_streamlines:
328: (20)                               raise InvalidIndexError
329: (16)                           else:
330: (20)                               pass
331: (0)            class InvalidIndexError(Exception):
332: (4)                pass
333: (0)            class TerminateTrajectory(Exception):
334: (4)                pass
335: (0)            def _get_integrator(u, v, dmap, minlength, maxlength, integration_direction):
```

```
336: (4)                u, v = dmap.data2grid(u, v)
337: (4)                u_ax = u / (dmap.grid.nx - 1)
338: (4)                v_ax = v / (dmap.grid.ny - 1)
339: (4)                speed = np.ma.sqrt(u_ax ** 2 + v_ax ** 2)
340: (4)                def forward_time(xi, yi):
341: (8)                    if not dmap.grid.within_grid(xi, yi):
342: (12)                       raise OutOfBounds
343: (8)                    ds_dt = interpgrid(speed, xi, yi)
344: (8)                    if ds_dt == 0:
345: (12)                       raise TerminateTrajectory()
346: (8)                    dt_ds = 1. / ds_dt
347: (8)                    ui = interpgrid(u, xi, yi)
348: (8)                    vi = interpgrid(v, xi, yi)
349: (8)                    return ui * dt_ds, vi * dt_ds
350: (4)                def backward_time(xi, yi):
351: (8)                    dxi, dyi = forward_time(xi, yi)
352: (8)                    return -dxi, -dyi
353: (4)                def integrate(x0, y0, broken_streamlines=True):
354: (8)                    """
355: (8)                    Return x, y grid-coordinates of trajectory based on starting point.
356: (8)                    Integrate both forward and backward in time from starting point in
357: (8)                    grid coordinates.
358: (8)                    Integration is terminated when a trajectory reaches a domain boundary
359: (8)                    or when it crosses into an already occupied cell in the StreamMask.
The
360: (8)                    resulting trajectory is None if it is shorter than `minlength`.
361: (8)                    """
362: (8)                    stotal, xy_traj = 0., []
363: (8)                    try:
364: (12)                       dmap.start_trajectory(x0, y0, broken_streamlines)
365: (8)                    except InvalidIndexError:
366: (12)                       return None
367: (8)                    if integration_direction in ['both', 'backward']:
368: (12)                       s, xyt = _integrate_rk12(x0, y0, dmap, backward_time, maxlength,
369: (37)                                                broken_streamlines)
370: (12)                       stotal += s
371: (12)                       xy_traj += xyt[::-1]
372: (8)                    if integration_direction in ['both', 'forward']:
373: (12)                       dmap.reset_start_point(x0, y0)
374: (12)                       s, xyt = _integrate_rk12(x0, y0, dmap, forward_time, maxlength,
375: (37)                                                broken_streamlines)
376: (12)                       stotal += s
377: (12)                       xy_traj += xyt[1:]
378: (8)                    if stotal > minlength:
379: (12)                       return np.broadcast_arrays(xy_traj, np.empty((1, 2)))[0]
380: (8)                    else:  # reject short trajectories
381: (12)                       dmap.undo_trajectory()
382: (12)                       return None
383: (4)                return integrate
384: (0)            class OutOfBounds(IndexError):
385: (4)                pass
386: (0)            def _integrate_rk12(x0, y0, dmap, f, maxlength, broken_streamlines=True):
387: (4)                """
388: (4)                2nd-order Runge-Kutta algorithm with adaptive step size.
389: (4)                This method is also referred to as the improved Euler's method, or Heun's
390: (4)                method. This method is favored over higher-order methods because:
391: (4)                1. To get decent looking trajectories and to sample every mask cell
392: (7)                   on the trajectory we need a small timestep, so a lower order
393: (7)                   solver doesn't hurt us unless the data is *very* high resolution.
394: (7)                   In fact, for cases where the user inputs
395: (7)                   data smaller or of similar grid size to the mask grid, the higher
396: (7)                   order corrections are negligible because of the very fast linear
397: (7)                   interpolation used in `interpgrid`.
398: (4)                2. For high resolution input data (i.e. beyond the mask
399: (7)                   resolution), we must reduce the timestep. Therefore, an adaptive
400: (7)                   timestep is more suited to the problem as this would be very hard
401: (7)                   to judge automatically otherwise.
402: (4)                This integrator is about 1.5 - 2x as fast as RK4 and RK45 solvers (using
403: (4)                similar Python implementations) in most setups.
```

```
(4)                 """
(4)                 maxerror = 0.003
(4)                 maxds = min(1. / dmap.mask.nx, 1. / dmap.mask.ny, 0.1)
(4)                 ds = maxds
(4)                 stotal = 0
(4)                 xi = x0
(4)                 yi = y0
(4)                 xyf_traj = []
(4)                 while True:
(8)                     try:
(12)                        if dmap.grid.within_grid(xi, yi):
(16)                            xyf_traj.append((xi, yi))
(12)                        else:
(16)                            raise OutOfBounds
(12)                        k1x, k1y = f(xi, yi)
(12)                        k2x, k2y = f(xi + ds * k1x, yi + ds * k1y)
(8)                     except OutOfBounds:
(12)                        if xyf_traj:
(16)                            ds, xyf_traj = _euler_step(xyf_traj, dmap, f)
(16)                            stotal += ds
(12)                        break
(8)                     except TerminateTrajectory:
(12)                        break
(8)                     dx1 = ds * k1x
(8)                     dy1 = ds * k1y
(8)                     dx2 = ds * 0.5 * (k1x + k2x)
(8)                     dy2 = ds * 0.5 * (k1y + k2y)
(8)                     ny, nx = dmap.grid.shape
(8)                     error = np.hypot((dx2 - dx1) / (nx - 1), (dy2 - dy1) / (ny - 1))
(8)                     if error < maxerror:
(12)                        xi += dx2
(12)                        yi += dy2
(12)                        try:
(16)                            dmap.update_trajectory(xi, yi, broken_streamlines)
(12)                        except InvalidIndexError:
(16)                            break
(12)                        if stotal + ds > maxlength:
(16)                            break
(12)                        stotal += ds
(8)                     if error == 0:
(12)                        ds = maxds
(8)                     else:
(12)                        ds = min(maxds, 0.85 * ds * (maxerror / error) ** 0.5)
(4)                 return stotal, xyf_traj
(0)             def _euler_step(xyf_traj, dmap, f):
(4)                 """Simple Euler integration step that extends streamline to boundary."""
(4)                 ny, nx = dmap.grid.shape
(4)                 xi, yi = xyf_traj[-1]
(4)                 cx, cy = f(xi, yi)
(4)                 if cx == 0:
(8)                     dsx = np.inf
(4)                 elif cx < 0:
(8)                     dsx = xi / -cx
(4)                 else:
(8)                     dsx = (nx - 1 - xi) / cx
(4)                 if cy == 0:
(8)                     dsy = np.inf
(4)                 elif cy < 0:
(8)                     dsy = yi / -cy
(4)                 else:
(8)                     dsy = (ny - 1 - yi) / cy
(4)                 ds = min(dsx, dsy)
(4)                 xyf_traj.append((xi + cx * ds, yi + cy * ds))
(4)                 return ds, xyf_traj
(0)             def interpgrid(a, xi, yi):
(4)                 """Fast 2D, linear interpolation on an integer grid"""
(4)                 Ny, Nx = np.shape(a)
(4)                 if isinstance(xi, np.ndarray):
(8)                     x = xi.astype(int)
```

```
473: (8)                    y = yi.astype(int)
474: (8)                    xn = np.clip(x + 1, 0, Nx - 1)
475: (8)                    yn = np.clip(y + 1, 0, Ny - 1)
476: (4)                else:
477: (8)                    x = int(xi)
478: (8)                    y = int(yi)
479: (8)                    if x == (Nx - 1):
480: (12)                       xn = x
481: (8)                    else:
482: (12)                       xn = x + 1
483: (8)                    if y == (Ny - 1):
484: (12)                       yn = y
485: (8)                    else:
486: (12)                       yn = y + 1
487: (4)                a00 = a[y, x]
488: (4)                a01 = a[y, xn]
489: (4)                a10 = a[yn, x]
490: (4)                a11 = a[yn, xn]
491: (4)                xt = xi - x
492: (4)                yt = yi - y
493: (4)                a0 = a00 * (1 - xt) + a01 * xt
494: (4)                a1 = a10 * (1 - xt) + a11 * xt
495: (4)                ai = a0 * (1 - yt) + a1 * yt
496: (4)                if not isinstance(xi, np.ndarray):
497: (8)                    if np.ma.is_masked(ai):
498: (12)                       raise TerminateTrajectory
499: (4)                return ai
500: (0)            def _gen_starting_points(shape):
501: (4)                """
502: (4)                Yield starting points for streamlines.
503: (4)                Trying points on the boundary first gives higher quality streamlines.
504: (4)                This algorithm starts with a point on the mask corner and spirals inward.
505: (4)                This algorithm is inefficient, but fast compared to rest of streamplot.
506: (4)                """
507: (4)                ny, nx = shape
508: (4)                xfirst = 0
509: (4)                yfirst = 1
510: (4)                xlast = nx - 1
511: (4)                ylast = ny - 1
512: (4)                x, y = 0, 0
513: (4)                direction = 'right'
514: (4)                for i in range(nx * ny):
515: (8)                    yield x, y
516: (8)                    if direction == 'right':
517: (12)                       x += 1
518: (12)                       if x >= xlast:
519: (16)                           xlast -= 1
520: (16)                           direction = 'up'
521: (8)                    elif direction == 'up':
522: (12)                       y += 1
523: (12)                       if y >= ylast:
524: (16)                           ylast -= 1
525: (16)                           direction = 'left'
526: (8)                    elif direction == 'left':
527: (12)                       x -= 1
528: (12)                       if x <= xfirst:
529: (16)                           xfirst += 1
530: (16)                           direction = 'down'
531: (8)                    elif direction == 'down':
532: (12)                       y -= 1
533: (12)                       if y <= yfirst:
534: (16)                           yfirst += 1
535: (16)                           direction = 'right'
```

File 63 - table.py:

```
1: (0)              """
```

```
2: (0)              Tables drawing.
3: (0)              .. note::
4: (4)                  The table implementation in Matplotlib is lightly maintained. For a more
5: (4)                  featureful table implementation, you may wish to try `blume
6: (4)                  <https://github.com/swfiua/blume>`_.
7: (0)              Use the factory function `~matplotlib.table.table` to create a ready-made
8: (0)              table from texts. If you need more control, use the `.Table` class and its
9: (0)              methods.
10: (0)             The table consists of a grid of cells, which are indexed by (row, column).
11: (0)             The cell (0, 0) is positioned at the top left.
12: (0)             Thanks to John Gill for providing the class and table.
13: (0)             """
14: (0)             import numpy as np
15: (0)             from . import _api, _docstring
16: (0)             from .artist import Artist, allow_rasterization
17: (0)             from .patches import Rectangle
18: (0)             from .text import Text
19: (0)             from .transforms import Bbox
20: (0)             from .path import Path
21: (0)             class Cell(Rectangle):
22: (4)                 """
23: (4)                 A cell is a `.Rectangle` with some associated `.Text`.
24: (4)                 As a user, you'll most likely not creates cells yourself. Instead, you
25: (4)                 should use either the `~matplotlib.table.table` factory function or
26: (4)                 `.Table.add_cell`.
27: (4)                 """
28: (4)                 PAD = 0.1
29: (4)                 """Padding between text and rectangle."""
30: (4)                 _edges = 'BRTL'
31: (4)                 _edge_aliases = {'open':         '',
32: (21)                                 'closed':       _edges,  # default
33: (21)                                 'horizontal':   'BT',
34: (21)                                 'vertical':     'RL'
35: (21)                                 }
36: (4)                 def __init__(self, xy, width, height, *,
37: (17)                             edgecolor='k', facecolor='w',
38: (17)                             fill=True,
39: (17)                             text='',
40: (17)                             loc='right',
41: (17)                             fontproperties=None,
42: (17)                             visible_edges='closed',
43: (17)                             ):
44: (8)                     """
45: (8)                     Parameters
46: (8)                     ----------
47: (8)                     xy : 2-tuple
48: (12)                        The position of the bottom left corner of the cell.
49: (8)                     width : float
50: (12)                        The cell width.
51: (8)                     height : float
52: (12)                        The cell height.
53: (8)                     edgecolor : :mpltype:`color`, default: 'k'
54: (12)                        The color of the cell border.
55: (8)                     facecolor : :mpltype:`color`, default: 'w'
56: (12)                        The cell facecolor.
57: (8)                     fill : bool, default: True
58: (12)                        Whether the cell background is filled.
59: (8)                     text : str, optional
60: (12)                        The cell text.
61: (8)                     loc : {'right', 'center', 'left'}
62: (12)                        The alignment of the text within the cell.
63: (8)                     fontproperties : dict, optional
64: (12)                        A dict defining the font properties of the text. Supported keys
and
65: (12)                        values are the keyword arguments accepted by `.FontProperties`.
66: (8)                     visible_edges : {'closed', 'open', 'horizontal', 'vertical'} or \
67: (0)             substring of 'BRTL'
68: (12)                        The cell edges to be drawn with a line: a substring of 'BRTL'
69: (12)                        (bottom, right, top, left), or one of 'open' (no edges drawn),
```

```
 70: (12)                        'closed' (all edges drawn), 'horizontal' (bottom and top),
 71: (12)                        'vertical' (right and left).
 72: (8)                     """
 73: (8)                     super().__init__(xy, width=width, height=height, fill=fill,
 74: (25)                                     edgecolor=edgecolor, facecolor=facecolor)
 75: (8)                     self.set_clip_on(False)
 76: (8)                     self.visible_edges = visible_edges
 77: (8)                     self._loc = loc
 78: (8)                     self._text = Text(x=xy[0], y=xy[1], clip_on=False,
 79: (26)                                      text=text, fontproperties=fontproperties,
 80: (26)                                      horizontalalignment=loc, verticalalignment='center')
 81: (4)                 @_api.rename_parameter("3.8", "trans", "t")
 82: (4)                 def set_transform(self, t):
 83: (8)                     super().set_transform(t)
 84: (8)                     self.stale = True
 85: (4)                 def set_figure(self, fig):
 86: (8)                     super().set_figure(fig)
 87: (8)                     self._text.set_figure(fig)
 88: (4)                 def get_text(self):
 89: (8)                     """Return the cell `.Text` instance."""
 90: (8)                     return self._text
 91: (4)                 def set_fontsize(self, size):
 92: (8)                     """Set the text fontsize."""
 93: (8)                     self._text.set_fontsize(size)
 94: (8)                     self.stale = True
 95: (4)                 def get_fontsize(self):
 96: (8)                     """Return the cell fontsize."""
 97: (8)                     return self._text.get_fontsize()
 98: (4)                 def auto_set_font_size(self, renderer):
 99: (8)                     """Shrink font size until the text fits into the cell width."""
 100: (8)                    fontsize = self.get_fontsize()
 101: (8)                    required = self.get_required_width(renderer)
 102: (8)                    while fontsize > 1 and required > self.get_width():
 103: (12)                       fontsize -= 1
 104: (12)                       self.set_fontsize(fontsize)
 105: (12)                       required = self.get_required_width(renderer)
 106: (8)                    return fontsize
 107: (4)                @allow_rasterization
 108: (4)                def draw(self, renderer):
 109: (8)                    if not self.get_visible():
 110: (12)                       return
 111: (8)                    super().draw(renderer)
 112: (8)                    self._set_text_position(renderer)
 113: (8)                    self._text.draw(renderer)
 114: (8)                    self.stale = False
 115: (4)                def _set_text_position(self, renderer):
 116: (8)                    """Set text up so it is drawn in the right place."""
 117: (8)                    bbox = self.get_window_extent(renderer)
 118: (8)                    y = bbox.y0 + bbox.height / 2
 119: (8)                    loc = self._text.get_horizontalalignment()
 120: (8)                    if loc == 'center':
 121: (12)                       x = bbox.x0 + bbox.width / 2
 122: (8)                    elif loc == 'left':
 123: (12)                       x = bbox.x0 + bbox.width * self.PAD
 124: (8)                    else:  # right.
 125: (12)                       x = bbox.x0 + bbox.width * (1 - self.PAD)
 126: (8)                    self._text.set_position((x, y))
 127: (4)                def get_text_bounds(self, renderer):
 128: (8)                    """
 129: (8)                    Return the text bounds as *(x, y, width, height)* in table
coordinates.
 130: (8)                    """
 131: (8)                    return (self._text.get_window_extent(renderer)
 132: (16)                           .transformed(self.get_data_transform().inverted())
 133: (16)                           .bounds)
 134: (4)                def get_required_width(self, renderer):
 135: (8)                    """Return the minimal required width for the cell."""
 136: (8)                    l, b, w, h = self.get_text_bounds(renderer)
 137: (8)                    return w * (1.0 + (2.0 * self.PAD))
```

```
(4)                 @_docstring.dedent_interpd
(4)                 def set_text_props(self, **kwargs):
(8)                     """
(8)                     Update the text properties.
(8)                     Valid keyword arguments are:
(8)                     %(Text:kwdoc)s
(8)                     """
(8)                     self._text._internal_update(kwargs)
(8)                     self.stale = True
(4)                 @property
(4)                 def visible_edges(self):
(8)                     """
(8)                     The cell edges to be drawn with a line.
(8)                     Reading this property returns a substring of 'BRTL' (bottom, right,
(8)                     top, left').
(8)                     When setting this property, you can use a substring of 'BRTL' or one
(8)                     of {'open', 'closed', 'horizontal', 'vertical'}.
(8)                     """
(8)                     return self._visible_edges
(4)                 @visible_edges.setter
(4)                 def visible_edges(self, value):
(8)                     if value is None:
(12)                        self._visible_edges = self._edges
(8)                     elif value in self._edge_aliases:
(12)                        self._visible_edges = self._edge_aliases[value]
(8)                     else:
(12)                        if any(edge not in self._edges for edge in value):
(16)                            raise ValueError('Invalid edge param {}, must only be one of '
(33)                                             '{} or string of {}'.format(
(37)                                                 value,
(37)                                                 ", ".join(self._edge_aliases),
(37)                                                 ", ".join(self._edges)))
(12)                        self._visible_edges = value
(8)                     self.stale = True
(4)                 def get_path(self):
(8)                     """Return a `.Path` for the `.visible_edges`."""
(8)                     codes = [Path.MOVETO]
(8)                     codes.extend(
(12)                        Path.LINETO if edge in self._visible_edges else Path.MOVETO
(12)                        for edge in self._edges)
(8)                     if Path.MOVETO not in codes[1:]:  # All sides are visible
(12)                        codes[-1] = Path.CLOSEPOLY
(8)                     return Path(
(12)                        [[0.0, 0.0], [1.0, 0.0], [1.0, 1.0], [0.0, 1.0], [0.0, 0.0]],
(12)                        codes,
(12)                        readonly=True
(12)                        )
(0)             CustomCell = Cell  # Backcompat. alias.
(0)             class Table(Artist):
(4)                 """
(4)                 A table of cells.
(4)                 The table consists of a grid of cells, which are indexed by (row, column).
(4)                 For a simple table, you'll have a full grid of cells with indices from
(4)                 (0, 0) to (num_rows-1, num_cols-1), in which the cell (0, 0) is positioned
(4)                 at the top left. However, you can also add cells with negative indices.
(4)                 You don't have to add a cell to every grid position, so you can create
(4)                 tables that have holes.
(4)                 *Note*: You'll usually not create an empty table from scratch. Instead use
(4)                 `~matplotlib.table.table` to create a table from data.
(4)                 """
(4)                 codes = {'best': 0,
(13)                         'upper right':  1,  # default
(13)                         'upper left':   2,
(13)                         'lower left':   3,
(13)                         'lower right':  4,
(13)                         'center left':  5,
(13)                         'center right': 6,
(13)                         'lower center': 7,
(13)                         'upper center': 8,
```

```
(13)                         'center':       9,
(13)                         'top right':    10,
(13)                         'top left':     11,
(13)                         'bottom left':  12,
(13)                         'bottom right': 13,
(13)                         'right':        14,
(13)                         'left':         15,
(13)                         'top':          16,
(13)                         'bottom':       17,
(13)                         }
(4)                 """Possible values where to place the table relative to the Axes."""
(4)                 FONTSIZE = 10
(4)                 AXESPAD = 0.02
(4)                 """The border between the Axes and the table edge in Axes units."""
(4)                 def __init__(self, ax, loc=None, bbox=None, **kwargs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     ax : `~matplotlib.axes.Axes`
(12)                        The `~.axes.Axes` to plot the table into.
(8)                     loc : str, optional
(12)                        The position of the cell with respect to *ax*. This must be one of
(12)                        the `~.Table.codes`.
(8)                     bbox : `.Bbox` or [xmin, ymin, width, height], optional
(12)                        A bounding box to draw the table into. If this is not *None*, this
(12)                        overrides *loc*.
(8)                     Other Parameters
(8)                     ----------------
(8)                     **kwargs
(12)                        `.Artist` properties.
(8)                     """
(8)                     super().__init__()
(8)                     if isinstance(loc, str):
(12)                        if loc not in self.codes:
(16)                            raise ValueError(
(20)                                "Unrecognized location {!r}. Valid locations are\n\t{}"
(20)                                .format(loc, '\n\t'.join(self.codes)))
(12)                        loc = self.codes[loc]
(8)                     self.set_figure(ax.figure)
(8)                     self._axes = ax
(8)                     self._loc = loc
(8)                     self._bbox = bbox
(8)                     ax._unstale_viewLim()
(8)                     self.set_transform(ax.transAxes)
(8)                     self._cells = {}
(8)                     self._edges = None
(8)                     self._autoColumns = []
(8)                     self._autoFontsize = True
(8)                     self._internal_update(kwargs)
(8)                     self.set_clip_on(False)
(4)                 def add_cell(self, row, col, *args, **kwargs):
(8)                     """
(8)                     Create a cell and add it to the table.
(8)                     Parameters
(8)                     ----------
(8)                     row : int
(12)                        Row index.
(8)                     col : int
(12)                        Column index.
(8)                     *args, **kwargs
(12)                        All other parameters are passed on to `Cell`.
(8)                     Returns
(8)                     -------
(8)                     `.Cell`
(12)                        The created cell.
(8)                     """
(8)                     xy = (0, 0)
(8)                     cell = Cell(xy, visible_edges=self.edges, *args, **kwargs)
(8)                     self[row, col] = cell
```

```
(8)                 return cell
(4)             def __setitem__(self, position, cell):
(8)                 """
(8)                 Set a custom cell in a given position.
(8)                 """
(8)                 _api.check_isinstance(Cell, cell=cell)
(8)                 try:
(12)                    row, col = position[0], position[1]
(8)                 except Exception as err:
(12)                    raise KeyError('Only tuples length 2 are accepted as '
(27)                                   'coordinates') from err
(8)                 cell.set_figure(self.figure)
(8)                 cell.set_transform(self.get_transform())
(8)                 cell.set_clip_on(False)
(8)                 self._cells[row, col] = cell
(8)                 self.stale = True
(4)             def __getitem__(self, position):
(8)                 """Retrieve a custom cell from a given position."""
(8)                 return self._cells[position]
(4)             @property
(4)             def edges(self):
(8)                 """
(8)                 The default value of `~.Cell.visible_edges` for newly added
(8)                 cells using `.add_cell`.
(8)                 Notes
(8)                 -----
(8)                 This setting does currently only affect newly created cells using
(8)                 `.add_cell`.
(8)                 To change existing cells, you have to set their edges explicitly::
(12)                    for c in tab.get_celld().values():
(16)                        c.visible_edges = 'horizontal'
(8)                 """
(8)                 return self._edges
(4)             @edges.setter
(4)             def edges(self, value):
(8)                 self._edges = value
(8)                 self.stale = True
(4)             def _approx_text_height(self):
(8)                 return (self.FONTSIZE / 72.0 * self.figure.dpi /
(16)                        self._axes.bbox.height * 1.2)
(4)             @allow_rasterization
(4)             def draw(self, renderer):
(8)                 if renderer is None:
(12)                    renderer = self.figure._get_renderer()
(8)                 if renderer is None:
(12)                    raise RuntimeError('No renderer defined')
(8)                 if not self.get_visible():
(12)                    return
(8)                 renderer.open_group('table', gid=self.get_gid())
(8)                 self._update_positions(renderer)
(8)                 for key in sorted(self._cells):
(12)                    self._cells[key].draw(renderer)
(8)                 renderer.close_group('table')
(8)                 self.stale = False
(4)             def _get_grid_bbox(self, renderer):
(8)                 """
(8)                 Get a bbox, in axes coordinates for the cells.
(8)                 Only include those in the range (0, 0) to (maxRow, maxCol).
(8)                 """
(8)                 boxes = [cell.get_window_extent(renderer)
(17)                         for (row, col), cell in self._cells.items()
(17)                         if row >= 0 and col >= 0]
(8)                 bbox = Bbox.union(boxes)
(8)                 return bbox.transformed(self.get_transform().inverted())
(4)             def contains(self, mouseevent):
(8)                 if self._different_canvas(mouseevent):
(12)                    return False, {}
(8)                 renderer = self.figure._get_renderer()
(8)                 if renderer is not None:
```

```
345: (12)                        boxes = [cell.get_window_extent(renderer)
346: (21)                                 for (row, col), cell in self._cells.items()
347: (21)                                 if row >= 0 and col >= 0]
348: (12)                        bbox = Bbox.union(boxes)
349: (12)                        return bbox.contains(mouseevent.x, mouseevent.y), {}
350: (8)                    else:
351: (12)                        return False, {}
352: (4)                def get_children(self):
353: (8)                    """Return the Artists contained by the table."""
354: (8)                    return list(self._cells.values())
355: (4)                def get_window_extent(self, renderer=None):
356: (8)                    if renderer is None:
357: (12)                       renderer = self.figure._get_renderer()
358: (8)                    self._update_positions(renderer)
359: (8)                    boxes = [cell.get_window_extent(renderer)
360: (17)                            for cell in self._cells.values()]
361: (8)                    return Bbox.union(boxes)
362: (4)                def _do_cell_alignment(self):
363: (8)                    """
364: (8)                    Calculate row heights and column widths; position cells accordingly.
365: (8)                    """
366: (8)                    widths = {}
367: (8)                    heights = {}
368: (8)                    for (row, col), cell in self._cells.items():
369: (12)                       height = heights.setdefault(row, 0.0)
370: (12)                       heights[row] = max(height, cell.get_height())
371: (12)                       width = widths.setdefault(col, 0.0)
372: (12)                       widths[col] = max(width, cell.get_width())
373: (8)                    xpos = 0
374: (8)                    lefts = {}
375: (8)                    for col in sorted(widths):
376: (12)                       lefts[col] = xpos
377: (12)                       xpos += widths[col]
378: (8)                    ypos = 0
379: (8)                    bottoms = {}
380: (8)                    for row in sorted(heights, reverse=True):
381: (12)                       bottoms[row] = ypos
382: (12)                       ypos += heights[row]
383: (8)                    for (row, col), cell in self._cells.items():
384: (12)                       cell.set_x(lefts[col])
385: (12)                       cell.set_y(bottoms[row])
386: (4)                def auto_set_column_width(self, col):
387: (8)                    """
388: (8)                    Automatically set the widths of given columns to optimal sizes.
389: (8)                    Parameters
390: (8)                    ----------
391: (8)                    col : int or sequence of ints
392: (12)                       The indices of the columns to auto-scale.
393: (8)                    """
394: (8)                    col1d = np.atleast_1d(col)
395: (8)                    if not np.issubdtype(col1d.dtype, np.integer):
396: (12)                       _api.warn_deprecated("3.8", name="col",
397: (33)                                            message="%(name)r must be an int or sequence
of ints. "
398: (33)                                            "Passing other types is deprecated since %
(since)s "
399: (33)                                            "and will be removed %(removal)s.")
400: (12)                       return
401: (8)                    for cell in col1d:
402: (12)                       self._autoColumns.append(cell)
403: (8)                    self.stale = True
404: (4)                def _auto_set_column_width(self, col, renderer):
405: (8)                    """Automatically set width for column."""
406: (8)                    cells = [cell for key, cell in self._cells.items() if key[1] == col]
407: (8)                    max_width = max((cell.get_required_width(renderer) for cell in cells),
408: (24)                                   default=0)
409: (8)                    for cell in cells:
410: (12)                       cell.set_width(max_width)
411: (4)                def auto_set_font_size(self, value=True):
```

```
(8)                 """Automatically set font size."""
(8)                 self._autoFontsize = value
(8)                 self.stale = True
(4)             def _auto_set_font_size(self, renderer):
(8)                 if len(self._cells) == 0:
(12)                    return
(8)                 fontsize = next(iter(self._cells.values())).get_fontsize()
(8)                 cells = []
(8)                 for key, cell in self._cells.items():
(12)                    if key[1] in self._autoColumns:
(16)                        continue
(12)                    size = cell.auto_set_font_size(renderer)
(12)                    fontsize = min(fontsize, size)
(12)                    cells.append(cell)
(8)                 for cell in self._cells.values():
(12)                    cell.set_fontsize(fontsize)
(4)             def scale(self, xscale, yscale):
(8)                 """Scale column widths by *xscale* and row heights by *yscale*."""
(8)                 for c in self._cells.values():
(12)                    c.set_width(c.get_width() * xscale)
(12)                    c.set_height(c.get_height() * yscale)
(4)             def set_fontsize(self, size):
(8)                 """
(8)                 Set the font size, in points, of the cell text.
(8)                 Parameters
(8)                 ----------
(8)                 size : float
(8)                 Notes
(8)                 -----
(8)                 As long as auto font size has not been disabled, the value will be
(8)                 clipped such that the text fits horizontally into the cell.
(8)                 You can disable this behavior using `.auto_set_font_size`.
(8)                 >>> the_table.auto_set_font_size(False)
(8)                 >>> the_table.set_fontsize(20)
(8)                 However, there is no automatic scaling of the row height so that the
(8)                 text may exceed the cell boundary.
(8)                 """
(8)                 for cell in self._cells.values():
(12)                    cell.set_fontsize(size)
(8)                 self.stale = True
(4)             def _offset(self, ox, oy):
(8)                 """Move all the artists by ox, oy (axes coords)."""
(8)                 for c in self._cells.values():
(12)                    x, y = c.get_x(), c.get_y()
(12)                    c.set_x(x + ox)
(12)                    c.set_y(y + oy)
(4)             def _update_positions(self, renderer):
(8)                 for col in self._autoColumns:
(12)                    self._auto_set_column_width(col, renderer)
(8)                 if self._autoFontsize:
(12)                    self._auto_set_font_size(renderer)
(8)                 self._do_cell_alignment()
(8)                 bbox = self._get_grid_bbox(renderer)
(8)                 l, b, w, h = bbox.bounds
(8)                 if self._bbox is not None:
(12)                    if isinstance(self._bbox, Bbox):
(16)                        rl, rb, rw, rh = self._bbox.bounds
(12)                    else:
(16)                        rl, rb, rw, rh = self._bbox
(12)                    self.scale(rw / w, rh / h)
(12)                    ox = rl - l
(12)                    oy = rb - b
(12)                    self._do_cell_alignment()
(8)                 else:
(12)                    (BEST, UR, UL, LL, LR, CL, CR, LC, UC, C,
(13)                     TR, TL, BL, BR, R, L, T, B) = range(len(self.codes))
(12)                    ox = (0.5 - w / 2) - l
(12)                    oy = (0.5 - h / 2) - b
(12)                    if self._loc in (UL, LL, CL):   # left
```

```
481: (16)                            ox = self.AXESPAD - l
482: (12)                        if self._loc in (BEST, UR, LR, R, CR):  # right
483: (16)                            ox = 1 - (l + w + self.AXESPAD)
484: (12)                        if self._loc in (BEST, UR, UL, UC):     # upper
485: (16)                            oy = 1 - (b + h + self.AXESPAD)
486: (12)                        if self._loc in (LL, LR, LC):           # lower
487: (16)                            oy = self.AXESPAD - b
488: (12)                        if self._loc in (LC, UC, C):            # center x
489: (16)                            ox = (0.5 - w / 2) - l
490: (12)                        if self._loc in (CL, CR, C):            # center y
491: (16)                            oy = (0.5 - h / 2) - b
492: (12)                        if self._loc in (TL, BL, L):            # out left
493: (16)                            ox = - (l + w)
494: (12)                        if self._loc in (TR, BR, R):            # out right
495: (16)                            ox = 1.0 - l
496: (12)                        if self._loc in (TR, TL, T):            # out top
497: (16)                            oy = 1.0 - b
498: (12)                        if self._loc in (BL, BR, B):           # out bottom
499: (16)                            oy = - (b + h)
500: (8)                    self._offset(ox, oy)
501: (4)                def get_celld(self):
502: (8)                    r"""
503: (8)                    Return a dict of cells in the table mapping *(row, column)* to
504: (8)                    `.Cell`\s.
505: (8)                    Notes
506: (8)                    -----
507: (8)                    You can also directly index into the Table object to access individual
508: (8)                    cells::
509: (12)                       cell = table[row, col]
510: (8)                    """
511: (8)                    return self._cells
512: (0)            @_docstring.dedent_interpd
513: (0)            def table(ax,
514: (10)                     cellText=None, cellColours=None,
515: (10)                     cellLoc='right', colWidths=None,
516: (10)                     rowLabels=None, rowColours=None, rowLoc='left',
517: (10)                     colLabels=None, colColours=None, colLoc='center',
518: (10)                     loc='bottom', bbox=None, edges='closed',
519: (10)                     **kwargs):
520: (4)                """
521: (4)                Add a table to an `~.axes.Axes`.
522: (4)                At least one of *cellText* or *cellColours* must be specified. These
523: (4)                parameters must be 2D lists, in which the outer lists define the rows and
524: (4)                the inner list define the column values per row. Each row must have the
525: (4)                same number of elements.
526: (4)                The table can optionally have row and column headers, which are configured
527: (4)                using *rowLabels*, *rowColours*, *rowLoc* and *colLabels*, *colColours*,
528: (4)                *colLoc* respectively.
529: (4)                For finer grained control over tables, use the `.Table` class and add it
to
530: (4)                the Axes with `.Axes.add_table`.
531: (4)                Parameters
532: (4)                ----------
533: (4)                cellText : 2D list of str, optional
534: (8)                    The texts to place into the table cells.
535: (8)                    *Note*: Line breaks in the strings are currently not accounted for and
536: (8)                    will result in the text exceeding the cell boundaries.
537: (4)                cellColours : 2D list of :mpltype:`color`, optional
538: (8)                    The background colors of the cells.
539: (4)                cellLoc : {'right', 'center', 'left'}
540: (8)                    The alignment of the text within the cells.
541: (4)                colWidths : list of float, optional
542: (8)                    The column widths in units of the axes. If not given, all columns will
543: (8)                    have a width of *1 / ncols*.
544: (4)                rowLabels : list of str, optional
545: (8)                    The text of the row header cells.
546: (4)                rowColours : list of :mpltype:`color`, optional
547: (8)                    The colors of the row header cells.
548: (4)                rowLoc : {'left', 'center', 'right'}
```

```
549: (8)                    The text alignment of the row header cells.
550: (4)                colLabels : list of str, optional
551: (8)                    The text of the column header cells.
552: (4)                colColours : list of :mpltype:`color`, optional
553: (8)                    The colors of the column header cells.
554: (4)                colLoc : {'center', 'left', 'right'}
555: (8)                    The text alignment of the column header cells.
556: (4)                loc : str, default: 'bottom'
557: (8)                    The position of the cell with respect to *ax*. This must be one of
558: (8)                    the `~.Table.codes`.
559: (4)                bbox : `.Bbox` or [xmin, ymin, width, height], optional
560: (8)                    A bounding box to draw the table into. If this is not *None*, this
561: (8)                    overrides *loc*.
562: (4)                edges : {'closed', 'open', 'horizontal', 'vertical'} or substring of
'BRTL'
563: (8)                    The cell edges to be drawn with a line. See also
564: (8)                    `~.Cell.visible_edges`.
565: (4)                Returns
566: (4)                -------
567: (4)                `~matplotlib.table.Table`
568: (8)                    The created table.
569: (4)                Other Parameters
570: (4)                ----------------
571: (4)                **kwargs
572: (8)                    `.Table` properties.
573: (4)                %(Table:kwdoc)s
574: (4)                """
575: (4)                if cellColours is None and cellText is None:
576: (8)                    raise ValueError('At least one argument from "cellColours" or '
577: (25)                                     '"cellText" must be provided to create a table.')
578: (4)                if cellText is None:
579: (8)                    rows = len(cellColours)
580: (8)                    cols = len(cellColours[0])
581: (8)                    cellText = [['''] * cols] * rows
582: (4)                rows = len(cellText)
583: (4)                cols = len(cellText[0])
584: (4)                for row in cellText:
585: (8)                    if len(row) != cols:
586: (12)                       raise ValueError(f"Each row in 'cellText' must have {cols} "
587: (29)                                        "columns")
588: (4)                if cellColours is not None:
589: (8)                    if len(cellColours) != rows:
590: (12)                       raise ValueError(f"'cellColours' must have {rows} rows")
591: (8)                    for row in cellColours:
592: (12)                       if len(row) != cols:
593: (16)                           raise ValueError("Each row in 'cellColours' must have "
594: (33)                                            f"{cols} columns")
595: (4)                else:
596: (8)                    cellColours = ['w' * cols] * rows
597: (4)                if colWidths is None:
598: (8)                    colWidths = [1.0 / cols] * cols
599: (4)                rowLabelWidth = 0
600: (4)                if rowLabels is None:
601: (8)                    if rowColours is not None:
602: (12)                       rowLabels = [''] * rows
603: (12)                       rowLabelWidth = colWidths[0]
604: (4)                elif rowColours is None:
605: (8)                    rowColours = 'w' * rows
606: (4)                if rowLabels is not None:
607: (8)                    if len(rowLabels) != rows:
608: (12)                       raise ValueError(f"'rowLabels' must be of length {rows}")
609: (4)                offset = 1
610: (4)                if colLabels is None:
611: (8)                    if colColours is not None:
612: (12)                       colLabels = [''] * cols
613: (8)                    else:
614: (12)                       offset = 0
615: (4)                elif colColours is None:
616: (8)                    colColours = 'w' * cols
```

```
617: (4)                if cellColours is None:
618: (8)                    cellColours = ['w' * cols] * rows
619: (4)                table = Table(ax, loc, bbox, **kwargs)
620: (4)                table.edges = edges
621: (4)                height = table._approx_text_height()
622: (4)                for row in range(rows):
623: (8)                    for col in range(cols):
624: (12)                       table.add_cell(row + offset, col,
625: (27)                                      width=colWidths[col], height=height,
626: (27)                                      text=cellText[row][col],
627: (27)                                      facecolor=cellColours[row][col],
628: (27)                                      loc=cellLoc)
629: (4)                if colLabels is not None:
630: (8)                    for col in range(cols):
631: (12)                       table.add_cell(0, col,
632: (27)                                      width=colWidths[col], height=height,
633: (27)                                      text=colLabels[col], facecolor=colColours[col],
634: (27)                                      loc=colLoc)
635: (4)                if rowLabels is not None:
636: (8)                    for row in range(rows):
637: (12)                       table.add_cell(row + offset, -1,
638: (27)                                      width=rowLabelWidth or 1e-15, height=height,
639: (27)                                      text=rowLabels[row], facecolor=rowColours[row],
640: (27)                                      loc=rowLoc)
641: (8)                    if rowLabelWidth == 0:
642: (12)                       table.auto_set_column_width(-1)
643: (4)                ax.add_table(table)
644: (4)                return table
```

File 64 - texmanager.py:

```
1: (0)              r"""
2: (0)              Support for embedded TeX expressions in Matplotlib.
3: (0)              Requirements:
4: (0)              * LaTeX.
5: (0)              * \*Agg backends: dvipng>=1.6.
6: (0)              * PS backend: PSfrag, dvips, and Ghostscript>=9.0.
7: (0)              * PDF and SVG backends: if LuaTeX is present, it will be used to speed up some
8: (2)                post-processing steps, but note that it is not used to parse the TeX string
9: (2)                itself (only LaTeX is supported).
10: (0)             To enable TeX rendering of all text in your Matplotlib figure, set
11: (0)             :rc:`text.usetex` to True.
12: (0)             TeX and dvipng/dvips processing results are cached
13: (0)             in ~/.matplotlib/tex.cache for reuse between sessions.
14: (0)             `TexManager.get_rgba` can also be used to directly obtain raster output as
RGBA
15: (0)             NumPy arrays.
16: (0)             """
17: (0)             import functools
18: (0)             import hashlib
19: (0)             import logging
20: (0)             import os
21: (0)             from pathlib import Path
22: (0)             import subprocess
23: (0)             from tempfile import TemporaryDirectory
24: (0)             import numpy as np
25: (0)             import matplotlib as mpl
26: (0)             from matplotlib import _api, cbook, dviread
27: (0)             _log = logging.getLogger(__name__)
28: (0)             def _usepackage_if_not_loaded(package, *, option=None):
29: (4)                 """
30: (4)                 Output LaTeX code that loads a package (possibly with an option) if it
31: (4)                 hasn't been loaded yet.
32: (4)                 LaTeX cannot load twice a package with different options, so this helper
33: (4)                 can be used to protect against users loading arbitrary packages/options in
34: (4)                 their custom preamble.
35: (4)                 """
```

```
36: (4)                 option = f"[{option}]" if option is not None else ""
37: (4)                 return (
38: (8)                     r"\makeatletter"
39: (8)                     r"\@ifpackageloaded{%(package)s}{}{\usepackage%(option)s{%
(package)s}}"
40: (8)                     r"\makeatother"
41: (4)                 ) % {"package": package, "option": option}
42: (0)             class TexManager:
43: (4)                 """
44: (4)                 Convert strings to dvi files using TeX, caching the results to a
directory.
45: (4)                 The cache directory is called ``tex.cache`` and is located in the
directory
46: (4)                 returned by `.get_cachedir`.
47: (4)                 Repeated calls to this constructor always return the same instance.
48: (4)                 """
49: (4)                 texcache = _api.deprecate_privatize_attribute("3.8")
50: (4)                 _texcache = os.path.join(mpl.get_cachedir(), 'tex.cache')
51: (4)                 _grey_arrayd = {}
52: (4)                 _font_families = ('serif', 'sans-serif', 'cursive', 'monospace')
53: (4)                 _font_preambles = {
54: (8)                     'new century schoolbook': r'\renewcommand{\rmdefault}{pnc}',
55: (8)                     'bookman': r'\renewcommand{\rmdefault}{pbk}',
56: (8)                     'times': r'\usepackage{mathptmx}',
57: (8)                     'palatino': r'\usepackage{mathpazo}',
58: (8)                     'zapf chancery': r'\usepackage{chancery}',
59: (8)                     'cursive': r'\usepackage{chancery}',
60: (8)                     'charter': r'\usepackage{charter}',
61: (8)                     'serif': '',
62: (8)                     'sans-serif': '',
63: (8)                     'helvetica': r'\usepackage{helvet}',
64: (8)                     'avant garde': r'\usepackage{avant}',
65: (8)                     'courier': r'\usepackage{courier}',
66: (8)                     'monospace': r'\usepackage{type1ec}',
67: (8)                     'computer modern roman': r'\usepackage{type1ec}',
68: (8)                     'computer modern sans serif': r'\usepackage{type1ec}',
69: (8)                     'computer modern typewriter': r'\usepackage{type1ec}',
70: (4)                 }
71: (4)                 _font_types = {
72: (8)                     'new century schoolbook': 'serif',
73: (8)                     'bookman': 'serif',
74: (8)                     'times': 'serif',
75: (8)                     'palatino': 'serif',
76: (8)                     'zapf chancery': 'cursive',
77: (8)                     'charter': 'serif',
78: (8)                     'helvetica': 'sans-serif',
79: (8)                     'avant garde': 'sans-serif',
80: (8)                     'courier': 'monospace',
81: (8)                     'computer modern roman': 'serif',
82: (8)                     'computer modern sans serif': 'sans-serif',
83: (8)                     'computer modern typewriter': 'monospace',
84: (4)                 }
85: (4)                 @functools.lru_cache  # Always return the same instance.
86: (4)                 def __new__(cls):
87: (8)                     Path(cls._texcache).mkdir(parents=True, exist_ok=True)
88: (8)                     return object.__new__(cls)
89: (4)                 @classmethod
90: (4)                 def _get_font_family_and_reduced(cls):
91: (8)                     """Return the font family name and whether the font is reduced."""
92: (8)                     ff = mpl.rcParams['font.family']
93: (8)                     ff_val = ff[0].lower() if len(ff) == 1 else None
94: (8)                     if len(ff) == 1 and ff_val in cls._font_families:
95: (12)                        return ff_val, False
96: (8)                     elif len(ff) == 1 and ff_val in cls._font_preambles:
97: (12)                        return cls._font_types[ff_val], True
98: (8)                     else:
99: (12)                        _log.info('font.family must be one of (%s) when text.usetex is '
100: (22)                                 'True. serif will be used by default.',
101: (22)                                 ', '.join(cls._font_families))
```

```
102: (12)                        return 'serif', False
103: (4)                 @classmethod
104: (4)                 def _get_font_preamble_and_command(cls):
105: (8)                     requested_family, is_reduced_font = cls._get_font_family_and_reduced()
106: (8)                     preambles = {}
107: (8)                     for font_family in cls._font_families:
108: (12)                        if is_reduced_font and font_family == requested_family:
109: (16)                            preambles[font_family] = cls._font_preambles[
110: (20)                                mpl.rcParams['font.family'][0].lower()]
111: (12)                        else:
112: (16)                            for font in mpl.rcParams['font.' + font_family]:
113: (20)                                if font.lower() in cls._font_preambles:
114: (24)                                    preambles[font_family] = \
115: (28)                                        cls._font_preambles[font.lower()]
116: (24)                                    _log.debug(
117: (28)                                        'family: %s, font: %s, info: %s',
118: (28)                                        font_family, font,
119: (28)                                        cls._font_preambles[font.lower()])
120: (24)                                    break
121: (20)                                else:
122: (24)                                    _log.debug('%s font is not compatible with usetex.',
123: (35)                                               font)
124: (16)                            else:
125: (20)                                _log.info('No LaTeX-compatible font found for the %s font'
126: (30)                                          'family in rcParams. Using default.',
127: (30)                                          font_family)
128: (20)                                preambles[font_family] = cls._font_preambles[font_family]
129: (8)                     cmd = {preambles[family]
130: (15)                            for family in ['serif', 'sans-serif', 'monospace']}
131: (8)                     if requested_family == 'cursive':
132: (12)                        cmd.add(preambles['cursive'])
133: (8)                     cmd.add(r'\usepackage{type1cm}')
134: (8)                     preamble = '\n'.join(sorted(cmd))
135: (8)                     fontcmd = (r'\sffamily' if requested_family == 'sans-serif' else
136: (19)                               r'\ttfamily' if requested_family == 'monospace' else
137: (19)                               r'\rmfamily')
138: (8)                     return preamble, fontcmd
139: (4)                 @classmethod
140: (4)                 def get_basefile(cls, tex, fontsize, dpi=None):
141: (8)                     """
142: (8)                     Return a filename based on a hash of the string, fontsize, and dpi.
143: (8)                     """
144: (8)                     src = cls._get_tex_source(tex, fontsize) + str(dpi)
145: (8)                     filehash = hashlib.md5(src.encode('utf-8')).hexdigest()
146: (8)                     filepath = Path(cls._texcache)
147: (8)                     num_letters, num_levels = 2, 2
148: (8)                     for i in range(0, num_letters*num_levels, num_letters):
149: (12)                        filepath = filepath / Path(filehash[i:i+2])
150: (8)                     filepath.mkdir(parents=True, exist_ok=True)
151: (8)                     return os.path.join(filepath, filehash)
152: (4)                 @classmethod
153: (4)                 def get_font_preamble(cls):
154: (8)                     """
155: (8)                     Return a string containing font configuration for the tex preamble.
156: (8)                     """
157: (8)                     font_preamble, command = cls._get_font_preamble_and_command()
158: (8)                     return font_preamble
159: (4)                 @classmethod
160: (4)                 def get_custom_preamble(cls):
161: (8)                     """Return a string containing user additions to the tex preamble."""
162: (8)                     return mpl.rcParams['text.latex.preamble']
163: (4)                 @classmethod
164: (4)                 def _get_tex_source(cls, tex, fontsize):
165: (8)                     """Return the complete TeX source for processing a TeX string."""
166: (8)                     font_preamble, fontcmd = cls._get_font_preamble_and_command()
167: (8)                     baselineskip = 1.25 * fontsize
168: (8)                     return "\n".join([
169: (12)                        r"\documentclass{article}",
170: (12)                        r"% Pass-through \mathdefault, which is used in non-usetex mode",
```

```
171: (12)                        r"% to use the default text font but was historically suppressed",
172: (12)                        r"% in usetex mode.",
173: (12)                        r"\newcommand{\mathdefault}[1]{#1}",
174: (12)                        font_preamble,
175: (12)                        r"\usepackage[utf8]{inputenc}",
176: (12)                        r"\DeclareUnicodeCharacter{2212}{\ensuremath{-}}",
177: (12)                        r"% geometry is loaded before the custom preamble as ",
178: (12)                        r"% convert_psfrags relies on a custom preamble to change the ",
179: (12)                        r"% geometry.",
180: (12)                        r"\usepackage[papersize=72in, margin=1in]{geometry}",
181: (12)                        cls.get_custom_preamble(),
182: (12)                        r"% Use `underscore` package to take care of underscores in
text.",
183: (12)                        r"% The [strings] option allows to use underscores in file
names.",
184: (12)                        _usepackage_if_not_loaded("underscore", option="strings"),
185: (12)                        r"% Custom packages (e.g. newtxtext) may already have loaded ",
186: (12)                        r"% textcomp with different options.",
187: (12)                        _usepackage_if_not_loaded("textcomp"),
188: (12)                        r"\pagestyle{empty}",
189: (12)                        r"\begin{document}",
190: (12)                        r"% The empty hbox ensures that a page is printed even for empty",
191: (12)                        r"% inputs, except when using psfrag which gets confused by it.",
192: (12)                        r"% matplotlibbaselinemarker is used by dviread to detect the",
193: (12)                        r"% last line's baseline.",
194: (12)                        rf"\fontsize{{{fontsize}}}{{{baselineskip}}}%",
195: (12)                        r"\ifdefined\psfrag\else\hbox{}\fi%",
196: (12)                        rf"{{{fontcmd} {tex}}}%",
197: (12)                        r"\end{document}",
198: (8)                    ])
199: (4)                @classmethod
200: (4)                def make_tex(cls, tex, fontsize):
201: (8)                    """
202: (8)                    Generate a tex file to render the tex string at a specific font size.
203: (8)                    Return the file name.
204: (8)                    """
205: (8)                    texfile = cls.get_basefile(tex, fontsize) + ".tex"
206: (8)                    Path(texfile).write_text(cls._get_tex_source(tex, fontsize),
207: (33)                                             encoding='utf-8')
208: (8)                    return texfile
209: (4)                @classmethod
210: (4)                def _run_checked_subprocess(cls, command, tex, *, cwd=None):
211: (8)                    _log.debug(cbook._pformat_subprocess(command))
212: (8)                    try:
213: (12)                       report = subprocess.check_output(
214: (16)                           command, cwd=cwd if cwd is not None else cls._texcache,
215: (16)                           stderr=subprocess.STDOUT)
216: (8)                    except FileNotFoundError as exc:
217: (12)                       raise RuntimeError(
218: (16)                           f'Failed to process string with tex because {command[0]} '
219: (16)                           'could not be found') from exc
220: (8)                    except subprocess.CalledProcessError as exc:
221: (12)                       raise RuntimeError(
222: (16)                           '{prog} was not able to process the following string:\n'
223: (16)                           '{tex!r}\n\n'
224: (16)                           'Here is the full command invocation and its output:\n\n'
225: (16)                           '{format_command}\n\n'
226: (16)                           '{exc}\n\n'.format(
227: (20)                               prog=command[0],
228: (20)                               format_command=cbook._pformat_subprocess(command),
229: (20)                               tex=tex.encode('unicode_escape'),
230: (20)                               exc=exc.output.decode('utf-8', 'backslashreplace'))
231: (16)                           ) from None
232: (8)                    _log.debug(report)
233: (8)                    return report
234: (4)                @classmethod
235: (4)                def make_dvi(cls, tex, fontsize):
236: (8)                    """
237: (8)                    Generate a dvi file containing latex's layout of tex string.
```

```
(8)                     Return the file name.
(8)                     """
(8)                     basefile = cls.get_basefile(tex, fontsize)
(8)                     dvifile = '%s.dvi' % basefile
(8)                     if not os.path.exists(dvifile):
(12)                        texfile = Path(cls.make_tex(tex, fontsize))
(12)                        cwd = Path(dvifile).parent
(12)                        with TemporaryDirectory(dir=cwd) as tmpdir:
(16)                            tmppath = Path(tmpdir)
(16)                            cls._run_checked_subprocess(
(20)                                ["latex", "-interaction=nonstopmode", "--halt-on-error",
(21)                                 f"--output-directory={tmppath.name}",
(21)                                 f"{texfile.name}"], tex, cwd=cwd)
(16)                            (tmppath / Path(dvifile).name).replace(dvifile)
(8)                     return dvifile
(4)                 @classmethod
(4)                 def make_png(cls, tex, fontsize, dpi):
(8)                     """
(8)                     Generate a png file containing latex's rendering of tex string.
(8)                     Return the file name.
(8)                     """
(8)                     basefile = cls.get_basefile(tex, fontsize, dpi)
(8)                     pngfile = '%s.png' % basefile
(8)                     if not os.path.exists(pngfile):
(12)                        dvifile = cls.make_dvi(tex, fontsize)
(12)                        cmd = ["dvipng", "-bg", "Transparent", "-D", str(dpi),
(19)                               "-T", "tight", "-o", pngfile, dvifile]
(12)                        if (getattr(mpl, "_called_from_pytest", False) and
(20)                                mpl._get_executable_info("dvipng").raw_version != "1.16"):
(16)                            cmd.insert(1, "--freetype0")
(12)                        cls._run_checked_subprocess(cmd, tex)
(8)                     return pngfile
(4)                 @classmethod
(4)                 def get_grey(cls, tex, fontsize=None, dpi=None):
(8)                     """Return the alpha channel."""
(8)                     if not fontsize:
(12)                        fontsize = mpl.rcParams['font.size']
(8)                     if not dpi:
(12)                        dpi = mpl.rcParams['savefig.dpi']
(8)                     key = cls._get_tex_source(tex, fontsize), dpi
(8)                     alpha = cls._grey_arrayd.get(key)
(8)                     if alpha is None:
(12)                        pngfile = cls.make_png(tex, fontsize, dpi)
(12)                        rgba = mpl.image.imread(os.path.join(cls._texcache, pngfile))
(12)                        cls._grey_arrayd[key] = alpha = rgba[:, :, -1]
(8)                     return alpha
(4)                 @classmethod
(4)                 def get_rgba(cls, tex, fontsize=None, dpi=None, rgb=(0, 0, 0)):
(8)                     r"""
(8)                     Return latex's rendering of the tex string as an RGBA array.
(8)                     Examples
(8)                     --------
(8)                     >>> texmanager = TexManager()
(8)                     >>> s = r"\TeX\ is $\displaystyle\sum_n\frac{-e^{i\pi}}{2^n}$!"
(8)                     >>> Z = texmanager.get_rgba(s, fontsize=12, dpi=80, rgb=(1, 0, 0))
(8)                     """
(8)                     alpha = cls.get_grey(tex, fontsize, dpi)
(8)                     rgba = np.empty((*alpha.shape, 4))
(8)                     rgba[..., :3] = mpl.colors.to_rgb(rgb)
(8)                     rgba[..., -1] = alpha
(8)                     return rgba
(4)                 @classmethod
(4)                 def get_text_width_height_descent(cls, tex, fontsize, renderer=None):
(8)                     """Return width, height and descent of the text."""
(8)                     if tex.strip() == '':
(12)                        return 0, 0, 0
(8)                     dvifile = cls.make_dvi(tex, fontsize)
(8)                     dpi_fraction = renderer.points_to_pixels(1.) if renderer else 1
(8)                     with dviread.Dvi(dvifile, 72 * dpi_fraction) as dvi:
```

```
307: (12)                        page, = dvi
308: (8)                     return page.width, page.height + page.descent, page.descent

----------------------------------------

File 65 - textpath.py:

1: (0)              from collections import OrderedDict
2: (0)              import logging
3: (0)              import urllib.parse
4: (0)              import numpy as np
5: (0)              from matplotlib import _text_helpers, dviread
6: (0)              from matplotlib.font_manager import (
7: (4)                  FontProperties, get_font, fontManager as _fontManager
8: (0)              )
9: (0)              from matplotlib.ft2font import LOAD_NO_HINTING, LOAD_TARGET_LIGHT
10: (0)             from matplotlib.mathtext import MathTextParser
11: (0)             from matplotlib.path import Path
12: (0)             from matplotlib.texmanager import TexManager
13: (0)             from matplotlib.transforms import Affine2D
14: (0)             _log = logging.getLogger(__name__)
15: (0)             class TextToPath:
16: (4)                 """A class that converts strings to paths."""
17: (4)                 FONT_SCALE = 100.
18: (4)                 DPI = 72
19: (4)                 def __init__(self):
20: (8)                     self.mathtext_parser = MathTextParser('path')
21: (8)                     self._texmanager = None
22: (4)                 def _get_font(self, prop):
23: (8)                     """
24: (8)                     Find the `FT2Font` matching font properties *prop*, with its size set.
25: (8)                     """
26: (8)                     filenames = _fontManager._find_fonts_by_props(prop)
27: (8)                     font = get_font(filenames)
28: (8)                     font.set_size(self.FONT_SCALE, self.DPI)
29: (8)                     return font
30: (4)                 def _get_hinting_flag(self):
31: (8)                     return LOAD_NO_HINTING
32: (4)                 def _get_char_id(self, font, ccode):
33: (8)                     """
34: (8)                     Return a unique id for the given font and character-code set.
35: (8)                     """
36: (8)                     return urllib.parse.quote(f"{font.postscript_name}-{ccode:x}")
37: (4)                 def get_text_width_height_descent(self, s, prop, ismath):
38: (8)                     fontsize = prop.get_size_in_points()
39: (8)                     if ismath == "TeX":
40: (12)                        return TexManager().get_text_width_height_descent(s, fontsize)
41: (8)                     scale = fontsize / self.FONT_SCALE
42: (8)                     if ismath:
43: (12)                        prop = prop.copy()
44: (12)                        prop.set_size(self.FONT_SCALE)
45: (12)                        width, height, descent, *_ = \
46: (16)                            self.mathtext_parser.parse(s, 72, prop)
47: (12)                        return width * scale, height * scale, descent * scale
48: (8)                     font = self._get_font(prop)
49: (8)                     font.set_text(s, 0.0, flags=LOAD_NO_HINTING)
50: (8)                     w, h = font.get_width_height()
51: (8)                     w /= 64.0  # convert from subpixels
52: (8)                     h /= 64.0
53: (8)                     d = font.get_descent()
54: (8)                     d /= 64.0
55: (8)                     return w * scale, h * scale, d * scale
56: (4)                 def get_text_path(self, prop, s, ismath=False):
57: (8)                     """
58: (8)                     Convert text *s* to path (a tuple of vertices and codes for
59: (8)                     matplotlib.path.Path).
60: (8)                     Parameters
61: (8)                     ----------
62: (8)                     prop : `~matplotlib.font_manager.FontProperties`
```

```
63: (12)                        The font properties for the text.
64: (8)                     s : str
65: (12)                        The text to be converted.
66: (8)                     ismath : {False, True, "TeX"}
67: (12)                        If True, use mathtext parser.  If "TeX", use tex for rendering.
68: (8)                     Returns
69: (8)                     -------
70: (8)                     verts : list
71: (12)                        A list of arrays containing the (x, y) coordinates of the
vertices.
72: (8)                     codes : list
73: (12)                        A list of path codes.
74: (8)                     Examples
75: (8)                     --------
76: (8)                     Create a list of vertices and codes from a text, and create a `.Path`
77: (8)                     from those::
78: (12)                        from matplotlib.path import Path
79: (12)                        from matplotlib.text import TextToPath
80: (12)                        from matplotlib.font_manager import FontProperties
81: (12)                        fp = FontProperties(family="Comic Neue", style="italic")
82: (12)                        verts, codes = TextToPath().get_text_path(fp, "ABC")
83: (12)                        path = Path(verts, codes, closed=False)
84: (8)                     Also see `TextPath` for a more direct way to create a path from a
text.
85: (8)                     """
86: (8)                     if ismath == "TeX":
87: (12)                        glyph_info, glyph_map, rects = self.get_glyphs_tex(prop, s)
88: (8)                     elif not ismath:
89: (12)                        font = self._get_font(prop)
90: (12)                        glyph_info, glyph_map, rects = self.get_glyphs_with_font(font, s)
91: (8)                     else:
92: (12)                        glyph_info, glyph_map, rects = self.get_glyphs_mathtext(prop, s)
93: (8)                     verts, codes = [], []
94: (8)                     for glyph_id, xposition, yposition, scale in glyph_info:
95: (12)                        verts1, codes1 = glyph_map[glyph_id]
96: (12)                        verts.extend(verts1 * scale + [xposition, yposition])
97: (12)                        codes.extend(codes1)
98: (8)                     for verts1, codes1 in rects:
99: (12)                        verts.extend(verts1)
100: (12)                       codes.extend(codes1)
101: (8)                    if not verts:
102: (12)                       verts = np.empty((0, 2))
103: (8)                    return verts, codes
104: (4)                def get_glyphs_with_font(self, font, s, glyph_map=None,
105: (29)                                        return_new_glyphs_only=False):
106: (8)                    """
107: (8)                    Convert string *s* to vertices and codes using the provided ttf font.
108: (8)                    """
109: (8)                    if glyph_map is None:
110: (12)                       glyph_map = OrderedDict()
111: (8)                    if return_new_glyphs_only:
112: (12)                       glyph_map_new = OrderedDict()
113: (8)                    else:
114: (12)                       glyph_map_new = glyph_map
115: (8)                    xpositions = []
116: (8)                    glyph_ids = []
117: (8)                    for item in _text_helpers.layout(s, font):
118: (12)                       char_id = self._get_char_id(item.ft_object, ord(item.char))
119: (12)                       glyph_ids.append(char_id)
120: (12)                       xpositions.append(item.x)
121: (12)                       if char_id not in glyph_map:
122: (16)                           glyph_map_new[char_id] = item.ft_object.get_path()
123: (8)                    ypositions = [0] * len(xpositions)
124: (8)                    sizes = [1.] * len(xpositions)
125: (8)                    rects = []
126: (8)                    return (list(zip(glyph_ids, xpositions, ypositions, sizes)),
127: (16)                           glyph_map_new, rects)
128: (4)                def get_glyphs_mathtext(self, prop, s, glyph_map=None,
129: (28)                                       return_new_glyphs_only=False):
```

```
(8)                 """
(8)                 Parse mathtext string *s* and convert it to a (vertices, codes) pair.
(8)                 """
(8)                 prop = prop.copy()
(8)                 prop.set_size(self.FONT_SCALE)
(8)                 width, height, descent, glyphs, rects = self.mathtext_parser.parse(
(12)                    s, self.DPI, prop)
(8)                 if not glyph_map:
(12)                    glyph_map = OrderedDict()
(8)                 if return_new_glyphs_only:
(12)                    glyph_map_new = OrderedDict()
(8)                 else:
(12)                    glyph_map_new = glyph_map
(8)                 xpositions = []
(8)                 ypositions = []
(8)                 glyph_ids = []
(8)                 sizes = []
(8)                 for font, fontsize, ccode, ox, oy in glyphs:
(12)                    char_id = self._get_char_id(font, ccode)
(12)                    if char_id not in glyph_map:
(16)                        font.clear()
(16)                        font.set_size(self.FONT_SCALE, self.DPI)
(16)                        font.load_char(ccode, flags=LOAD_NO_HINTING)
(16)                        glyph_map_new[char_id] = font.get_path()
(12)                    xpositions.append(ox)
(12)                    ypositions.append(oy)
(12)                    glyph_ids.append(char_id)
(12)                    size = fontsize / self.FONT_SCALE
(12)                    sizes.append(size)
(8)                 myrects = []
(8)                 for ox, oy, w, h in rects:
(12)                    vert1 = [(ox, oy), (ox, oy + h), (ox + w, oy + h),
(21)                             (ox + w, oy), (ox, oy), (0, 0)]
(12)                    code1 = [Path.MOVETO,
(21)                             Path.LINETO, Path.LINETO, Path.LINETO, Path.LINETO,
(21)                             Path.CLOSEPOLY]
(12)                    myrects.append((vert1, code1))
(8)                 return (list(zip(glyph_ids, xpositions, ypositions, sizes)),
(16)                        glyph_map_new, myrects)
(4)             def get_glyphs_tex(self, prop, s, glyph_map=None,
(23)                               return_new_glyphs_only=False):
(8)                 """Convert the string *s* to vertices and codes using usetex mode."""
(8)                 dvifile = TexManager().make_dvi(s, self.FONT_SCALE)
(8)                 with dviread.Dvi(dvifile, self.DPI) as dvi:
(12)                    page, = dvi
(8)                 if glyph_map is None:
(12)                    glyph_map = OrderedDict()
(8)                 if return_new_glyphs_only:
(12)                    glyph_map_new = OrderedDict()
(8)                 else:
(12)                    glyph_map_new = glyph_map
(8)                 glyph_ids, xpositions, ypositions, sizes = [], [], [], []
(8)                 for text in page.text:
(12)                    font = get_font(text.font_path)
(12)                    char_id = self._get_char_id(font, text.glyph)
(12)                    if char_id not in glyph_map:
(16)                        font.clear()
(16)                        font.set_size(self.FONT_SCALE, self.DPI)
(16)                        glyph_name_or_index = text.glyph_name_or_index
(16)                        if isinstance(glyph_name_or_index, str):
(20)                            index = font.get_name_index(glyph_name_or_index)
(20)                            font.load_glyph(index, flags=LOAD_TARGET_LIGHT)
(16)                        elif isinstance(glyph_name_or_index, int):
(20)                            self._select_native_charmap(font)
(20)                            font.load_char(
(24)                                glyph_name_or_index, flags=LOAD_TARGET_LIGHT)
(16)                        else:  # Should not occur.
(20)                            raise TypeError(f"Glyph spec of unexpected type: "
(36)                                            f"{glyph_name_or_index!r}")
```

```
(16)                            glyph_map_new[char_id] = font.get_path()
(12)                        glyph_ids.append(char_id)
(12)                        xpositions.append(text.x)
(12)                        ypositions.append(text.y)
(12)                        sizes.append(text.font_size / self.FONT_SCALE)
(8)                     myrects = []
(8)                     for ox, oy, h, w in page.boxes:
(12)                        vert1 = [(ox, oy), (ox + w, oy), (ox + w, oy + h),
(21)                                 (ox, oy + h), (ox, oy), (0, 0)]
(12)                        code1 = [Path.MOVETO,
(21)                                 Path.LINETO, Path.LINETO, Path.LINETO, Path.LINETO,
(21)                                 Path.CLOSEPOLY]
(12)                        myrects.append((vert1, code1))
(8)                     return (list(zip(glyph_ids, xpositions, ypositions, sizes)),
(16)                            glyph_map_new, myrects)
(4)                 @staticmethod
(4)                 def _select_native_charmap(font):
(8)                     for charmap_code in [
(16)                            1094992451,  # ADOBE_CUSTOM.
(16)                            1094995778,  # ADOBE_STANDARD.
(8)                     ]:
(12)                        try:
(16)                            font.select_charmap(charmap_code)
(12)                        except (ValueError, RuntimeError):
(16)                            pass
(12)                        else:
(16)                            break
(8)                     else:
(12)                        _log.warning("No supported encoding in font (%s).", font.fname)
(0)             text_to_path = TextToPath()
(0)             class TextPath(Path):
(4)                 """
(4)                 Create a path from the text.
(4)                 """
(4)                 def __init__(self, xy, s, size=None, prop=None,
(17)                             _interpolation_steps=1, usetex=False):
(8)                     r"""
(8)                     Create a path from the text. Note that it simply is a path,
(8)                     not an artist. You need to use the `.PathPatch` (or other artists)
(8)                     to draw this path onto the canvas.
(8)                     Parameters
(8)                     ----------
(8)                     xy : tuple or array of two float values
(12)                        Position of the text. For no offset, use ``xy=(0, 0)``.
(8)                     s : str
(12)                        The text to convert to a path.
(8)                     size : float, optional
(12)                        Font size in points. Defaults to the size specified via the font
(12)                        properties *prop*.
(8)                     prop : `~matplotlib.font_manager.FontProperties`, optional
(12)                        Font property. If not provided, will use a default
(12)                        `.FontProperties` with parameters from the
(12)                        :ref:`rcParams<customizing-with-dynamic-rc-settings>`.
(8)                     _interpolation_steps : int, optional
(12)                        (Currently ignored)
(8)                     usetex : bool, default: False
(12)                        Whether to use tex rendering.
(8)                     Examples
(8)                     --------
(8)                     The following creates a path from the string "ABC" with Helvetica
(8)                     font face; and another path from the latex fraction 1/2::
(12)                        from matplotlib.text import TextPath
(12)                        from matplotlib.font_manager import FontProperties
(12)                        fp = FontProperties(family="Helvetica", style="italic")
(12)                        path1 = TextPath((12, 12), "ABC", size=12, prop=fp)
(12)                        path2 = TextPath((0, 0), r"$\frac{1}{2}$", size=12, usetex=True)
(8)                     Also see :doc:`/gallery/text_labels_and_annotations/demo_text_path`.
(8)                     """
(8)                     from matplotlib.text import Text
```

```
268: (8)                    prop = FontProperties._from_any(prop)
269: (8)                    if size is None:
270: (12)                       size = prop.get_size_in_points()
271: (8)                    self._xy = xy
272: (8)                    self.set_size(size)
273: (8)                    self._cached_vertices = None
274: (8)                    s, ismath = Text(usetex=usetex)._preprocess_math(s)
275: (8)                    super().__init__(
276: (12)                       *text_to_path.get_text_path(prop, s, ismath=ismath),
277: (12)                       _interpolation_steps=_interpolation_steps,
278: (12)                       readonly=True)
279: (8)                    self._should_simplify = False
280: (4)                def set_size(self, size):
281: (8)                    """Set the text size."""
282: (8)                    self._size = size
283: (8)                    self._invalid = True
284: (4)                def get_size(self):
285: (8)                    """Get the text size."""
286: (8)                    return self._size
287: (4)                @property
288: (4)                def vertices(self):
289: (8)                    """
290: (8)                    Return the cached path after updating it if necessary.
291: (8)                    """
292: (8)                    self._revalidate_path()
293: (8)                    return self._cached_vertices
294: (4)                @property
295: (4)                def codes(self):
296: (8)                    """
297: (8)                    Return the codes
298: (8)                    """
299: (8)                    return self._codes
300: (4)                def _revalidate_path(self):
301: (8)                    """
302: (8)                    Update the path if necessary.
303: (8)                    The path for the text is initially create with the font size of
304: (8)                    `.FONT_SCALE`, and this path is rescaled to other size when necessary.
305: (8)                    """
306: (8)                    if self._invalid or self._cached_vertices is None:
307: (12)                       tr = (Affine2D()
308: (18)                             .scale(self._size / text_to_path.FONT_SCALE)
309: (18)                             .translate(*self._xy))
310: (12)                       self._cached_vertices = tr.transform(self._vertices)
311: (12)                       self._cached_vertices.flags.writeable = False
312: (12)                       self._invalid = False
```

File 66 - text.py:

```
1: (0)              """
2: (0)              Classes for including text in a figure.
3: (0)              """
4: (0)              import functools
5: (0)              import logging
6: (0)              import math
7: (0)              from numbers import Real
8: (0)              import weakref
9: (0)              import numpy as np
10: (0)             import matplotlib as mpl
11: (0)             from . import _api, artist, cbook, _docstring
12: (0)             from .artist import Artist
13: (0)             from .font_manager import FontProperties
14: (0)             from .patches import FancyArrowPatch, FancyBboxPatch, Rectangle
15: (0)             from .textpath import TextPath, TextToPath  # noqa # Logically located here
16: (0)             from .transforms import (
17: (4)                 Affine2D, Bbox, BboxBase, BboxTransformTo, IdentityTransform, Transform)
18: (0)             _log = logging.getLogger(__name__)
19: (0)             def _get_textbox(text, renderer):
```

```
 20: (4)                 """
 21: (4)                 Calculate the bounding box of the text.
 22: (4)                 The bbox position takes text rotation into account, but the width and
 23: (4)                 height are those of the unrotated box (unlike `.Text.get_window_extent`).
 24: (4)                 """
 25: (4)                 projected_xs = []
 26: (4)                 projected_ys = []
 27: (4)                 theta = np.deg2rad(text.get_rotation())
 28: (4)                 tr = Affine2D().rotate(-theta)
 29: (4)                 _, parts, d = text._get_layout(renderer)
 30: (4)                 for t, wh, x, y in parts:
 31: (8)                     w, h = wh
 32: (8)                     xt1, yt1 = tr.transform((x, y))
 33: (8)                     yt1 -= d
 34: (8)                     xt2, yt2 = xt1 + w, yt1 + h
 35: (8)                     projected_xs.extend([xt1, xt2])
 36: (8)                     projected_ys.extend([yt1, yt2])
 37: (4)                 xt_box, yt_box = min(projected_xs), min(projected_ys)
 38: (4)                 w_box, h_box = max(projected_xs) - xt_box, max(projected_ys) - yt_box
 39: (4)                 x_box, y_box = Affine2D().rotate(theta).transform((xt_box, yt_box))
 40: (4)                 return x_box, y_box, w_box, h_box
 41: (0)             def _get_text_metrics_with_cache(renderer, text, fontprop, ismath, dpi):
 42: (4)                 """Call ``renderer.get_text_width_height_descent``, caching the
results."""
 43: (4)                 return _get_text_metrics_with_cache_impl(
 44: (8)                     weakref.ref(renderer), text, fontprop.copy(), ismath, dpi)
 45: (0)             @functools.lru_cache(4096)
 46: (0)             def _get_text_metrics_with_cache_impl(
 47: (8)                     renderer_ref, text, fontprop, ismath, dpi):
 48: (4)                 return renderer_ref().get_text_width_height_descent(text, fontprop,
ismath)
 49: (0)             @_docstring.interpd
 50: (0)             @_api.define_aliases({
 51: (4)                 "color": ["c"],
 52: (4)                 "fontproperties": ["font", "font_properties"],
 53: (4)                 "fontfamily": ["family"],
 54: (4)                 "fontname": ["name"],
 55: (4)                 "fontsize": ["size"],
 56: (4)                 "fontstretch": ["stretch"],
 57: (4)                 "fontstyle": ["style"],
 58: (4)                 "fontvariant": ["variant"],
 59: (4)                 "fontweight": ["weight"],
 60: (4)                 "horizontalalignment": ["ha"],
 61: (4)                 "verticalalignment": ["va"],
 62: (4)                 "multialignment": ["ma"],
 63: (0)             })
 64: (0)             class Text(Artist):
 65: (4)                 """Handle storing and drawing of text in window or data coordinates."""
 66: (4)                 zorder = 3
 67: (4)                 _charsize_cache = dict()
 68: (4)                 def __repr__(self):
 69: (8)                     return f"Text({self._x}, {self._y}, {self._text!r})"
 70: (4)                 def __init__(self,
 71: (17)                             x=0, y=0, text='', *,
 72: (17)                             color=None,           # defaults to rc params
 73: (17)                             verticalalignment='baseline',
 74: (17)                             horizontalalignment='left',
 75: (17)                             multialignment=None,
 76: (17)                             fontproperties=None,  # defaults to FontProperties()
 77: (17)                             rotation=None,
 78: (17)                             linespacing=None,
 79: (17)                             rotation_mode=None,
 80: (17)                             usetex=None,          # defaults to rcParams['text.usetex']
 81: (17)                             wrap=False,
 82: (17)                             transform_rotates_text=False,
 83: (17)                             parse_math=None,    # defaults to rcParams['text.parse_math']
 84: (17)                             antialiased=None,  # defaults to rcParams['text.antialiased']
 85: (17)                             **kwargs
 86: (17)                             ):
```

```
87: (8)                     """
88: (8)                     Create a `.Text` instance at *x*, *y* with string *text*.
89: (8)                     The text is aligned relative to the anchor point (*x*, *y*) according
90: (8)                     to ``horizontalalignment`` (default: 'left') and ``verticalalignment``
91: (8)                     (default: 'baseline'). See also
92: (8)                     :doc:`/gallery/text_labels_and_annotations/text_alignment`.
93: (8)                     While Text accepts the 'label' keyword argument, by default it is not
94: (8)                     added to the handles of a legend.
95: (8)                     Valid keyword arguments are:
96: (8)                     %(Text:kwdoc)s
97: (8)                     """
98: (8)                     super().__init__()
99: (8)                     self._x, self._y = x, y
100: (8)                    self._text = ''
101: (8)                    self._reset_visual_defaults(
102: (12)                       text=text,
103: (12)                       color=color,
104: (12)                       fontproperties=fontproperties,
105: (12)                       usetex=usetex,
106: (12)                       parse_math=parse_math,
107: (12)                       wrap=wrap,
108: (12)                       verticalalignment=verticalalignment,
109: (12)                       horizontalalignment=horizontalalignment,
110: (12)                       multialignment=multialignment,
111: (12)                       rotation=rotation,
112: (12)                       transform_rotates_text=transform_rotates_text,
113: (12)                       linespacing=linespacing,
114: (12)                       rotation_mode=rotation_mode,
115: (12)                       antialiased=antialiased
116: (8)                    )
117: (8)                    self.update(kwargs)
118: (4)                def _reset_visual_defaults(
119: (8)                    self,
120: (8)                    text='',
121: (8)                    color=None,
122: (8)                    fontproperties=None,
123: (8)                    usetex=None,
124: (8)                    parse_math=None,
125: (8)                    wrap=False,
126: (8)                    verticalalignment='baseline',
127: (8)                    horizontalalignment='left',
128: (8)                    multialignment=None,
129: (8)                    rotation=None,
130: (8)                    transform_rotates_text=False,
131: (8)                    linespacing=None,
132: (8)                    rotation_mode=None,
133: (8)                    antialiased=None
134: (4)                ):
135: (8)                    self.set_text(text)
136: (8)                    self.set_color(mpl._val_or_rc(color, "text.color"))
137: (8)                    self.set_fontproperties(fontproperties)
138: (8)                    self.set_usetex(usetex)
139: (8)                    self.set_parse_math(mpl._val_or_rc(parse_math, 'text.parse_math'))
140: (8)                    self.set_wrap(wrap)
141: (8)                    self.set_verticalalignment(verticalalignment)
142: (8)                    self.set_horizontalalignment(horizontalalignment)
143: (8)                    self._multialignment = multialignment
144: (8)                    self.set_rotation(rotation)
145: (8)                    self._transform_rotates_text = transform_rotates_text
146: (8)                    self._bbox_patch = None  # a FancyBboxPatch instance
147: (8)                    self._renderer = None
148: (8)                    if linespacing is None:
149: (12)                       linespacing = 1.2  # Maybe use rcParam later.
150: (8)                    self.set_linespacing(linespacing)
151: (8)                    self.set_rotation_mode(rotation_mode)
152: (8)                    self.set_antialiased(antialiased if antialiased is not None else
153: (29)                                        mpl.rcParams['text.antialiased'])
154: (4)                def update(self, kwargs):
155: (8)                    ret = []
```

```
156: (8)                    kwargs = cbook.normalize_kwargs(kwargs, Text)
157: (8)                    sentinel = object()  # bbox can be None, so use another sentinel.
158: (8)                    fontproperties = kwargs.pop("fontproperties", sentinel)
159: (8)                    if fontproperties is not sentinel:
160: (12)                       ret.append(self.set_fontproperties(fontproperties))
161: (8)                    bbox = kwargs.pop("bbox", sentinel)
162: (8)                    ret.extend(super().update(kwargs))
163: (8)                    if bbox is not sentinel:
164: (12)                       ret.append(self.set_bbox(bbox))
165: (8)                    return ret
166: (4)                def __getstate__(self):
167: (8)                    d = super().__getstate__()
168: (8)                    d['_renderer'] = None
169: (8)                    return d
170: (4)                def contains(self, mouseevent):
171: (8)                    """
172: (8)                    Return whether the mouse event occurred inside the axis-aligned
173: (8)                    bounding-box of the text.
174: (8)                    """
175: (8)                    if (self._different_canvas(mouseevent) or not self.get_visible()
176: (16)                           or self._renderer is None):
177: (12)                       return False, {}
178: (8)                    bbox = Text.get_window_extent(self)
179: (8)                    inside = (bbox.x0 <= mouseevent.x <= bbox.x1
180: (18)                             and bbox.y0 <= mouseevent.y <= bbox.y1)
181: (8)                    cattr = {}
182: (8)                    if self._bbox_patch:
183: (12)                       patch_inside, patch_cattr = self._bbox_patch.contains(mouseevent)
184: (12)                       inside = inside or patch_inside
185: (12)                       cattr["bbox_patch"] = patch_cattr
186: (8)                    return inside, cattr
187: (4)                def _get_xy_display(self):
188: (8)                    """
189: (8)                    Get the (possibly unit converted) transformed x, y in display coords.
190: (8)                    """
191: (8)                    x, y = self.get_unitless_position()
192: (8)                    return self.get_transform().transform((x, y))
193: (4)                def _get_multialignment(self):
194: (8)                    if self._multialignment is not None:
195: (12)                       return self._multialignment
196: (8)                    else:
197: (12)                       return self._horizontalalignment
198: (4)                def _char_index_at(self, x):
199: (8)                    """
200: (8)                    Calculate the index closest to the coordinate x in display space.
201: (8)                    The position of text[index] is assumed to be the sum of the widths
202: (8)                    of all preceding characters text[:index].
203: (8)                    This works only on single line texts.
204: (8)                    """
205: (8)                    if not self._text:
206: (12)                       return 0
207: (8)                    text = self._text
208: (8)                    fontproperties = str(self._fontproperties)
209: (8)                    if fontproperties not in Text._charsize_cache:
210: (12)                       Text._charsize_cache[fontproperties] = dict()
211: (8)                    charsize_cache = Text._charsize_cache[fontproperties]
212: (8)                    for char in set(text):
213: (12)                       if char not in charsize_cache:
214: (16)                           self.set_text(char)
215: (16)                           bb = self.get_window_extent()
216: (16)                           charsize_cache[char] = bb.x1 - bb.x0
217: (8)                    self.set_text(text)
218: (8)                    bb = self.get_window_extent()
219: (8)                    size_accum = np.cumsum([0] + [charsize_cache[x] for x in text])
220: (8)                    std_x = x - bb.x0
221: (8)                    return (np.abs(size_accum - std_x)).argmin()
222: (4)                def get_rotation(self):
223: (8)                    """Return the text angle in degrees between 0 and 360."""
224: (8)                    if self.get_transform_rotates_text():
```

```
(12)                        return self.get_transform().transform_angles(
(16)                            [self._rotation], [self.get_unitless_position()]).item(0)
(8)                     else:
(12)                        return self._rotation
(4)                 def get_transform_rotates_text(self):
(8)                     """
(8)                     Return whether rotations of the transform affect the text direction.
(8)                     """
(8)                     return self._transform_rotates_text
(4)                 def set_rotation_mode(self, m):
(8)                     """
(8)                     Set text rotation mode.
(8)                     Parameters
(8)                     ----------
(8)                     m : {None, 'default', 'anchor'}
(12)                        If ``"default"``, the text will be first rotated, then aligned
according
(12)                        to their horizontal and vertical alignments.  If ``"anchor"``,
then
(12)                        alignment occurs before rotation. Passing ``None`` will set the
rotation
(12)                        mode to ``"default"``.
(8)                     """
(8)                     if m is None:
(12)                        m = "default"
(8)                     else:
(12)                        _api.check_in_list(("anchor", "default"), rotation_mode=m)
(8)                     self._rotation_mode = m
(8)                     self.stale = True
(4)                 def get_rotation_mode(self):
(8)                     """Return the text rotation mode."""
(8)                     return self._rotation_mode
(4)                 def set_antialiased(self, antialiased):
(8)                     """
(8)                     Set whether to use antialiased rendering.
(8)                     Parameters
(8)                     ----------
(8)                     antialiased : bool
(8)                     Notes
(8)                     -----
(8)                     Antialiasing will be determined by :rc:`text.antialiased`
(8)                     and the parameter *antialiased* will have no effect if the text
contains
(8)                     math expressions.
(8)                     """
(8)                     self._antialiased = antialiased
(8)                     self.stale = True
(4)                 def get_antialiased(self):
(8)                     """Return whether antialiased rendering is used."""
(8)                     return self._antialiased
(4)                 def update_from(self, other):
(8)                     super().update_from(other)
(8)                     self._color = other._color
(8)                     self._multialignment = other._multialignment
(8)                     self._verticalalignment = other._verticalalignment
(8)                     self._horizontalalignment = other._horizontalalignment
(8)                     self._fontproperties = other._fontproperties.copy()
(8)                     self._usetex = other._usetex
(8)                     self._rotation = other._rotation
(8)                     self._transform_rotates_text = other._transform_rotates_text
(8)                     self._picker = other._picker
(8)                     self._linespacing = other._linespacing
(8)                     self._antialiased = other._antialiased
(8)                     self.stale = True
(4)                 def _get_layout(self, renderer):
(8)                     """
(8)                     Return the extent (bbox) of the text together with
(8)                     multiple-alignment information. Note that it returns an extent
(8)                     of a rotated text when necessary.
```

```
290: (8)                    """
291: (8)                    thisx, thisy = 0.0, 0.0
292: (8)                    lines = self._get_wrapped_text().split("\n")  # Ensures lines is not
empty.
293: (8)                    ws = []
294: (8)                    hs = []
295: (8)                    xs = []
296: (8)                    ys = []
297: (8)                    _, lp_h, lp_d = _get_text_metrics_with_cache(
298: (12)                       renderer, "lp", self._fontproperties,
299: (12)                       ismath="TeX" if self.get_usetex() else False, dpi=self.figure.dpi)
300: (8)                    min_dy = (lp_h - lp_d) * self._linespacing
301: (8)                    for i, line in enumerate(lines):
302: (12)                       clean_line, ismath = self._preprocess_math(line)
303: (12)                       if clean_line:
304: (16)                           w, h, d = _get_text_metrics_with_cache(
305: (20)                               renderer, clean_line, self._fontproperties,
306: (20)                               ismath=ismath, dpi=self.figure.dpi)
307: (12)                       else:
308: (16)                           w = h = d = 0
309: (12)                       h = max(h, lp_h)
310: (12)                       d = max(d, lp_d)
311: (12)                       ws.append(w)
312: (12)                       hs.append(h)
313: (12)                       baseline = (h - d) - thisy
314: (12)                       if i == 0:
315: (16)                           thisy = -(h - d)
316: (12)                       else:
317: (16)                           thisy -= max(min_dy, (h - d) * self._linespacing)
318: (12)                       xs.append(thisx)  # == 0.
319: (12)                       ys.append(thisy)
320: (12)                       thisy -= d
321: (8)                    descent = d
322: (8)                    width = max(ws)
323: (8)                    xmin = 0
324: (8)                    xmax = width
325: (8)                    ymax = 0
326: (8)                    ymin = ys[-1] - descent  # baseline of last line minus its descent
327: (8)                    M = Affine2D().rotate_deg(self.get_rotation())
328: (8)                    malign = self._get_multialignment()
329: (8)                    if malign == 'left':
330: (12)                       offset_layout = [(x, y) for x, y in zip(xs, ys)]
331: (8)                    elif malign == 'center':
332: (12)                       offset_layout = [(x + width / 2 - w / 2, y)
333: (29)                                        for x, y, w in zip(xs, ys, ws)]
334: (8)                    elif malign == 'right':
335: (12)                       offset_layout = [(x + width - w, y)
336: (29)                                        for x, y, w in zip(xs, ys, ws)]
337: (8)                    corners_horiz = np.array(
338: (12)                       [(xmin, ymin), (xmin, ymax), (xmax, ymax), (xmax, ymin)])
339: (8)                    corners_rotated = M.transform(corners_horiz)
340: (8)                    xmin = corners_rotated[:, 0].min()
341: (8)                    xmax = corners_rotated[:, 0].max()
342: (8)                    ymin = corners_rotated[:, 1].min()
343: (8)                    ymax = corners_rotated[:, 1].max()
344: (8)                    width = xmax - xmin
345: (8)                    height = ymax - ymin
346: (8)                    halign = self._horizontalalignment
347: (8)                    valign = self._verticalalignment
348: (8)                    rotation_mode = self.get_rotation_mode()
349: (8)                    if rotation_mode != "anchor":
350: (12)                       if halign == 'center':
351: (16)                           offsetx = (xmin + xmax) / 2
352: (12)                       elif halign == 'right':
353: (16)                           offsetx = xmax
354: (12)                       else:
355: (16)                           offsetx = xmin
356: (12)                       if valign == 'center':
357: (16)                           offsety = (ymin + ymax) / 2
```

```
(12)                        elif valign == 'top':
(16)                            offsety = ymax
(12)                        elif valign == 'baseline':
(16)                            offsety = ymin + descent
(12)                        elif valign == 'center_baseline':
(16)                            offsety = ymin + height - baseline / 2.0
(12)                        else:
(16)                            offsety = ymin
(8)                     else:
(12)                        xmin1, ymin1 = corners_horiz[0]
(12)                        xmax1, ymax1 = corners_horiz[2]
(12)                        if halign == 'center':
(16)                            offsetx = (xmin1 + xmax1) / 2.0
(12)                        elif halign == 'right':
(16)                            offsetx = xmax1
(12)                        else:
(16)                            offsetx = xmin1
(12)                        if valign == 'center':
(16)                            offsety = (ymin1 + ymax1) / 2.0
(12)                        elif valign == 'top':
(16)                            offsety = ymax1
(12)                        elif valign == 'baseline':
(16)                            offsety = ymax1 - baseline
(12)                        elif valign == 'center_baseline':
(16)                            offsety = ymax1 - baseline / 2.0
(12)                        else:
(16)                            offsety = ymin1
(12)                        offsetx, offsety = M.transform((offsetx, offsety))
(8)                     xmin -= offsetx
(8)                     ymin -= offsety
(8)                     bbox = Bbox.from_bounds(xmin, ymin, width, height)
(8)                     xys = M.transform(offset_layout) - (offsetx, offsety)
(8)                     return bbox, list(zip(lines, zip(ws, hs), *xys.T)), descent
(4)                 def set_bbox(self, rectprops):
(8)                     """
(8)                     Draw a bounding box around self.
(8)                     Parameters
(8)                     ----------
(8)                     rectprops : dict with properties for `.patches.FancyBboxPatch`
(13)                         The default boxstyle is 'square'. The mutation
(13)                         scale of the `.patches.FancyBboxPatch` is set to the fontsize.
(8)                     Examples
(8)                     --------
(8)                     ::
(12)                        t.set_bbox(dict(facecolor='red', alpha=0.5))
(8)                     """
(8)                     if rectprops is not None:
(12)                        props = rectprops.copy()
(12)                        boxstyle = props.pop("boxstyle", None)
(12)                        pad = props.pop("pad", None)
(12)                        if boxstyle is None:
(16)                            boxstyle = "square"
(16)                            if pad is None:
(20)                                pad = 4  # points
(16)                            pad /= self.get_size()  # to fraction of font size
(12)                        else:
(16)                            if pad is None:
(20)                                pad = 0.3
(12)                        if isinstance(boxstyle, str) and "pad" not in boxstyle:
(16)                            boxstyle += ",pad=%0.2f" % pad
(12)                        self._bbox_patch = FancyBboxPatch(
(16)                            (0, 0), 1, 1,
(16)                            boxstyle=boxstyle, transform=IdentityTransform(), **props)
(8)                     else:
(12)                        self._bbox_patch = None
(8)                     self._update_clip_properties()
(4)                 def get_bbox_patch(self):
(8)                     """
(8)                     Return the bbox Patch, or None if the `.patches.FancyBboxPatch`
```

```
427: (8)                    is not made.
428: (8)                    """
429: (8)                    return self._bbox_patch
430: (4)                def update_bbox_position_size(self, renderer):
431: (8)                    """
432: (8)                    Update the location and the size of the bbox.
433: (8)                    This method should be used when the position and size of the bbox
needs
434: (8)                    to be updated before actually drawing the bbox.
435: (8)                    """
436: (8)                    if self._bbox_patch:
437: (12)                       posx = float(self.convert_xunits(self._x))
438: (12)                       posy = float(self.convert_yunits(self._y))
439: (12)                       posx, posy = self.get_transform().transform((posx, posy))
440: (12)                       x_box, y_box, w_box, h_box = _get_textbox(self, renderer)
441: (12)                       self._bbox_patch.set_bounds(0., 0., w_box, h_box)
442: (12)                       self._bbox_patch.set_transform(
443: (16)                           Affine2D()
444: (16)                           .rotate_deg(self.get_rotation())
445: (16)                           .translate(posx + x_box, posy + y_box))
446: (12)                       fontsize_in_pixel = renderer.points_to_pixels(self.get_size())
447: (12)                       self._bbox_patch.set_mutation_scale(fontsize_in_pixel)
448: (4)                def _update_clip_properties(self):
449: (8)                    if self._bbox_patch:
450: (12)                       clipprops = dict(clip_box=self.clipbox,
451: (29)                                        clip_path=self._clippath,
452: (29)                                        clip_on=self._clipon)
453: (12)                       self._bbox_patch.update(clipprops)
454: (4)                def set_clip_box(self, clipbox):
455: (8)                    super().set_clip_box(clipbox)
456: (8)                    self._update_clip_properties()
457: (4)                def set_clip_path(self, path, transform=None):
458: (8)                    super().set_clip_path(path, transform)
459: (8)                    self._update_clip_properties()
460: (4)                def set_clip_on(self, b):
461: (8)                    super().set_clip_on(b)
462: (8)                    self._update_clip_properties()
463: (4)                def get_wrap(self):
464: (8)                    """Return whether the text can be wrapped."""
465: (8)                    return self._wrap
466: (4)                def set_wrap(self, wrap):
467: (8)                    """
468: (8)                    Set whether the text can be wrapped.
469: (8)                    Wrapping makes sure the text is confined to the (sub)figure box. It
470: (8)                    does not take into account any other artists.
471: (8)                    Parameters
472: (8)                    ----------
473: (8)                    wrap : bool
474: (8)                    Notes
475: (8)                    -----
476: (8)                    Wrapping does not work together with
477: (8)                    ``savefig(..., bbox_inches='tight')`` (which is also used internally
478: (8)                    by ``%matplotlib inline`` in IPython/Jupyter). The 'tight' setting
479: (8)                    rescales the canvas to accommodate all content and happens before
480: (8)                    wrapping.
481: (8)                    """
482: (8)                    self._wrap = wrap
483: (4)                def _get_wrap_line_width(self):
484: (8)                    """
485: (8)                    Return the maximum line width for wrapping text based on the current
486: (8)                    orientation.
487: (8)                    """
488: (8)                    x0, y0 = self.get_transform().transform(self.get_position())
489: (8)                    figure_box = self.get_figure().get_window_extent()
490: (8)                    alignment = self.get_horizontalalignment()
491: (8)                    self.set_rotation_mode('anchor')
492: (8)                    rotation = self.get_rotation()
493: (8)                    left = self._get_dist_to_box(rotation, x0, y0, figure_box)
494: (8)                    right = self._get_dist_to_box(
```

```
(12)                        (180 + rotation) % 360, x0, y0, figure_box)
(8)                     if alignment == 'left':
(12)                        line_width = left
(8)                     elif alignment == 'right':
(12)                        line_width = right
(8)                     else:
(12)                        line_width = 2 * min(left, right)
(8)                     return line_width
(4)                 def _get_dist_to_box(self, rotation, x0, y0, figure_box):
(8)                     """
(8)                     Return the distance from the given points to the boundaries of a
(8)                     rotated box, in pixels.
(8)                     """
(8)                     if rotation > 270:
(12)                        quad = rotation - 270
(12)                        h1 = (y0 - figure_box.y0) / math.cos(math.radians(quad))
(12)                        h2 = (figure_box.x1 - x0) / math.cos(math.radians(90 - quad))
(8)                     elif rotation > 180:
(12)                        quad = rotation - 180
(12)                        h1 = (x0 - figure_box.x0) / math.cos(math.radians(quad))
(12)                        h2 = (y0 - figure_box.y0) / math.cos(math.radians(90 - quad))
(8)                     elif rotation > 90:
(12)                        quad = rotation - 90
(12)                        h1 = (figure_box.y1 - y0) / math.cos(math.radians(quad))
(12)                        h2 = (x0 - figure_box.x0) / math.cos(math.radians(90 - quad))
(8)                     else:
(12)                        h1 = (figure_box.x1 - x0) / math.cos(math.radians(rotation))
(12)                        h2 = (figure_box.y1 - y0) / math.cos(math.radians(90 - rotation))
(8)                     return min(h1, h2)
(4)                 def _get_rendered_text_width(self, text):
(8)                     """
(8)                     Return the width of a given text string, in pixels.
(8)                     """
(8)                     w, h, d = self._renderer.get_text_width_height_descent(
(12)                        text,
(12)                        self.get_fontproperties(),
(12)                        cbook.is_math_text(text))
(8)                     return math.ceil(w)
(4)                 def _get_wrapped_text(self):
(8)                     """
(8)                     Return a copy of the text string with new lines added so that the text
(8)                     is wrapped relative to the parent figure (if `get_wrap` is True).
(8)                     """
(8)                     if not self.get_wrap():
(12)                        return self.get_text()
(8)                     if self.get_usetex():
(12)                        return self.get_text()
(8)                     line_width = self._get_wrap_line_width()
(8)                     wrapped_lines = []
(8)                     unwrapped_lines = self.get_text().split('\n')
(8)                     for unwrapped_line in unwrapped_lines:
(12)                        sub_words = unwrapped_line.split(' ')
(12)                        while len(sub_words) > 0:
(16)                            if len(sub_words) == 1:
(20)                                wrapped_lines.append(sub_words.pop(0))
(20)                                continue
(16)                            for i in range(2, len(sub_words) + 1):
(20)                                line = ' '.join(sub_words[:i])
(20)                                current_width = self._get_rendered_text_width(line)
(20)                                if current_width > line_width:
(24)                                    wrapped_lines.append(' '.join(sub_words[:i - 1]))
(24)                                    sub_words = sub_words[i - 1:]
(24)                                    break
(20)                                elif i == len(sub_words):
(24)                                    wrapped_lines.append(' '.join(sub_words[:i]))
(24)                                    sub_words = []
(24)                                    break
(8)                     return '\n'.join(wrapped_lines)
(4)                 @artist.allow_rasterization
```

```
(4)     def draw(self, renderer):
(8)         if renderer is not None:
(12)            self._renderer = renderer
(8)         if not self.get_visible():
(12)            return
(8)         if self.get_text() == '':
(12)            return
(8)         renderer.open_group('text', self.get_gid())
(8)         with self._cm_set(text=self._get_wrapped_text()):
(12)            bbox, info, descent = self._get_layout(renderer)
(12)            trans = self.get_transform()
(12)            posx = float(self.convert_xunits(self._x))
(12)            posy = float(self.convert_yunits(self._y))
(12)            posx, posy = trans.transform((posx, posy))
(12)            if not np.isfinite(posx) or not np.isfinite(posy):
(16)                _log.warning("posx and posy should be finite values")
(16)                return
(12)            canvasw, canvash = renderer.get_canvas_width_height()
(12)            if self._bbox_patch:
(16)                self.update_bbox_position_size(renderer)
(16)                self._bbox_patch.draw(renderer)
(12)            gc = renderer.new_gc()
(12)            gc.set_foreground(self.get_color())
(12)            gc.set_alpha(self.get_alpha())
(12)            gc.set_url(self._url)
(12)            gc.set_antialiased(self._antialiased)
(12)            self._set_gc_clip(gc)
(12)            angle = self.get_rotation()
(12)            for line, wh, x, y in info:
(16)                mtext = self if len(info) == 1 else None
(16)                x = x + posx
(16)                y = y + posy
(16)                if renderer.flipy():
(20)                    y = canvash - y
(16)                clean_line, ismath = self._preprocess_math(line)
(16)                if self.get_path_effects():
(20)                    from matplotlib.patheffects import PathEffectRenderer
(20)                    textrenderer = PathEffectRenderer(
(24)                        self.get_path_effects(), renderer)
(16)                else:
(20)                    textrenderer = renderer
(16)                if self.get_usetex():
(20)                    textrenderer.draw_tex(gc, x, y, clean_line,
(42)                                          self._fontproperties, angle,
(42)                                          mtext=mtext)
(16)                else:
(20)                    textrenderer.draw_text(gc, x, y, clean_line,
(43)                                           self._fontproperties, angle,
(43)                                           ismath=ismath, mtext=mtext)
(8)         gc.restore()
(8)         renderer.close_group('text')
(8)         self.stale = False
(4)     def get_color(self):
(8)         """Return the color of the text."""
(8)         return self._color
(4)     def get_fontproperties(self):
(8)         """Return the `.font_manager.FontProperties`."""
(8)         return self._fontproperties
(4)     def get_fontfamily(self):
(8)         """
(8)         Return the list of font families used for font lookup.
(8)         See Also
(8)         --------
(8)         .font_manager.FontProperties.get_family
(8)         """
(8)         return self._fontproperties.get_family()
(4)     def get_fontname(self):
(8)         """
(8)         Return the font name as a string.
```

```
(8)                 See Also
(8)                 --------
(8)                 .font_manager.FontProperties.get_name
(8)                 """
(8)                 return self._fontproperties.get_name()
(4)             def get_fontstyle(self):
(8)                 """
(8)                 Return the font style as a string.
(8)                 See Also
(8)                 --------
(8)                 .font_manager.FontProperties.get_style
(8)                 """
(8)                 return self._fontproperties.get_style()
(4)             def get_fontsize(self):
(8)                 """
(8)                 Return the font size as an integer.
(8)                 See Also
(8)                 --------
(8)                 .font_manager.FontProperties.get_size_in_points
(8)                 """
(8)                 return self._fontproperties.get_size_in_points()
(4)             def get_fontvariant(self):
(8)                 """
(8)                 Return the font variant as a string.
(8)                 See Also
(8)                 --------
(8)                 .font_manager.FontProperties.get_variant
(8)                 """
(8)                 return self._fontproperties.get_variant()
(4)             def get_fontweight(self):
(8)                 """
(8)                 Return the font weight as a string or a number.
(8)                 See Also
(8)                 --------
(8)                 .font_manager.FontProperties.get_weight
(8)                 """
(8)                 return self._fontproperties.get_weight()
(4)             def get_stretch(self):
(8)                 """
(8)                 Return the font stretch as a string or a number.
(8)                 See Also
(8)                 --------
(8)                 .font_manager.FontProperties.get_stretch
(8)                 """
(8)                 return self._fontproperties.get_stretch()
(4)             def get_horizontalalignment(self):
(8)                 """
(8)                 Return the horizontal alignment as a string.  Will be one of
(8)                 'left', 'center' or 'right'.
(8)                 """
(8)                 return self._horizontalalignment
(4)             def get_unitless_position(self):
(8)                 """Return the (x, y) unitless position of the text."""
(8)                 x = float(self.convert_xunits(self._x))
(8)                 y = float(self.convert_yunits(self._y))
(8)                 return x, y
(4)             def get_position(self):
(8)                 """Return the (x, y) position of the text."""
(8)                 return self._x, self._y
(4)             def get_text(self):
(8)                 """Return the text string."""
(8)                 return self._text
(4)             def get_verticalalignment(self):
(8)                 """
(8)                 Return the vertical alignment as a string.  Will be one of
(8)                 'top', 'center', 'bottom', 'baseline' or 'center_baseline'.
(8)                 """
(8)                 return self._verticalalignment
(4)             def get_window_extent(self, renderer=None, dpi=None):
```

```
(8)                 """
(8)                 Return the `.Bbox` bounding the text, in display units.
(8)                 In addition to being used internally, this is useful for specifying
(8)                 clickable regions in a png file on a web page.
(8)                 Parameters
(8)                 ----------
(8)                 renderer : Renderer, optional
(12)                    A renderer is needed to compute the bounding box.  If the artist
(12)                    has already been drawn, the renderer is cached; thus, it is only
(12)                    necessary to pass this argument when calling `get_window_extent`
(12)                    before the first draw.  In practice, it is usually easier to
(12)                    trigger a draw first, e.g. by calling
(12)                    `~.Figure.draw_without_rendering` or ``plt.show()``.
(8)                 dpi : float, optional
(12)                    The dpi value for computing the bbox, defaults to
(12)                    ``self.figure.dpi`` (*not* the renderer dpi); should be set e.g.
if
(12)                    to match regions with a figure saved with a custom dpi value.
(8)                 """
(8)                 if not self.get_visible():
(12)                    return Bbox.unit()
(8)                 if dpi is None:
(12)                    dpi = self.figure.dpi
(8)                 if self.get_text() == '':
(12)                    with cbook._setattr_cm(self.figure, dpi=dpi):
(16)                        tx, ty = self._get_xy_display()
(16)                        return Bbox.from_bounds(tx, ty, 0, 0)
(8)                 if renderer is not None:
(12)                    self._renderer = renderer
(8)                 if self._renderer is None:
(12)                    self._renderer = self.figure._get_renderer()
(8)                 if self._renderer is None:
(12)                    raise RuntimeError(
(16)                        "Cannot get window extent of text w/o renderer. You likely "
(16)                        "want to call 'figure.draw_without_rendering()' first.")
(8)                 with cbook._setattr_cm(self.figure, dpi=dpi):
(12)                    bbox, info, descent = self._get_layout(self._renderer)
(12)                    x, y = self.get_unitless_position()
(12)                    x, y = self.get_transform().transform((x, y))
(12)                    bbox = bbox.translated(x, y)
(12)                    return bbox
(4)             def set_backgroundcolor(self, color):
(8)                 """
(8)                 Set the background color of the text by updating the bbox.
(8)                 Parameters
(8)                 ----------
(8)                 color : :mpltype:`color`
(8)                 See Also
(8)                 --------
(8)                 .set_bbox : To change the position of the bounding box
(8)                 """
(8)                 if self._bbox_patch is None:
(12)                    self.set_bbox(dict(facecolor=color, edgecolor=color))
(8)                 else:
(12)                    self._bbox_patch.update(dict(facecolor=color))
(8)                 self._update_clip_properties()
(8)                 self.stale = True
(4)             def set_color(self, color):
(8)                 """
(8)                 Set the foreground color of the text
(8)                 Parameters
(8)                 ----------
(8)                 color : :mpltype:`color`
(8)                 """
(8)                 if not cbook._str_equal(color, "auto"):
(12)                    mpl.colors._check_color_like(color=color)
(8)                 self._color = color
(8)                 self.stale = True
(4)             def set_horizontalalignment(self, align):
```

```
(8)                 """
(8)                 Set the horizontal alignment relative to the anchor point.
(8)                 See also :doc:`/gallery/text_labels_and_annotations/text_alignment`.
(8)                 Parameters
(8)                 ----------
(8)                 align : {'left', 'center', 'right'}
(8)                 """
(8)                 _api.check_in_list(['center', 'right', 'left'], align=align)
(8)                 self._horizontalalignment = align
(8)                 self.stale = True
(4)             def set_multialignment(self, align):
(8)                 """
(8)                 Set the text alignment for multiline texts.
(8)                 The layout of the bounding box of all the lines is determined by the
(8)                 horizontalalignment and verticalalignment properties. This property
(8)                 controls the alignment of the text lines within that box.
(8)                 Parameters
(8)                 ----------
(8)                 align : {'left', 'right', 'center'}
(8)                 """
(8)                 _api.check_in_list(['center', 'right', 'left'], align=align)
(8)                 self._multialignment = align
(8)                 self.stale = True
(4)             def set_linespacing(self, spacing):
(8)                 """
(8)                 Set the line spacing as a multiple of the font size.
(8)                 The default line spacing is 1.2.
(8)                 Parameters
(8)                 ----------
(8)                 spacing : float (multiple of font size)
(8)                 """
(8)                 _api.check_isinstance(Real, spacing=spacing)
(8)                 self._linespacing = spacing
(8)                 self.stale = True
(4)             def set_fontfamily(self, fontname):
(8)                 """
(8)                 Set the font family.  Can be either a single string, or a list of
(8)                 strings in decreasing priority.  Each string may be either a real font
(8)                 name or a generic font class name.  If the latter, the specific font
(8)                 names will be looked up in the corresponding rcParams.
(8)                 If a `Text` instance is constructed with ``fontfamily=None``, then the
(8)                 font is set to :rc:`font.family`, and the
(8)                 same is done when `set_fontfamily()` is called on an existing
(8)                 `Text` instance.
(8)                 Parameters
(8)                 ----------
(8)                 fontname : {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', \
(0)         'monospace'}
(8)                 See Also
(8)                 --------
(8)                 .font_manager.FontProperties.set_family
(8)                 """
(8)                 self._fontproperties.set_family(fontname)
(8)                 self.stale = True
(4)             def set_fontvariant(self, variant):
(8)                 """
(8)                 Set the font variant.
(8)                 Parameters
(8)                 ----------
(8)                 variant : {'normal', 'small-caps'}
(8)                 See Also
(8)                 --------
(8)                 .font_manager.FontProperties.set_variant
(8)                 """
(8)                 self._fontproperties.set_variant(variant)
(8)                 self.stale = True
(4)             def set_fontstyle(self, fontstyle):
(8)                 """
(8)                 Set the font style.
```

```
(8)                     Parameters
(8)                     ----------
(8)                     fontstyle : {'normal', 'italic', 'oblique'}
(8)                     See Also
(8)                     --------
(8)                     .font_manager.FontProperties.set_style
(8)                     """
(8)                     self._fontproperties.set_style(fontstyle)
(8)                     self.stale = True
(4)                 def set_fontsize(self, fontsize):
(8)                     """
(8)                     Set the font size.
(8)                     Parameters
(8)                     ----------
(8)                     fontsize : float or {'xx-small', 'x-small', 'small', 'medium', \
(0)             'large', 'x-large', 'xx-large'}
(12)                        If a float, the fontsize in points. The string values denote sizes
(12)                        relative to the default font size.
(8)                     See Also
(8)                     --------
(8)                     .font_manager.FontProperties.set_size
(8)                     """
(8)                     self._fontproperties.set_size(fontsize)
(8)                     self.stale = True
(4)                 def get_math_fontfamily(self):
(8)                     """
(8)                     Return the font family name for math text rendered by Matplotlib.
(8)                     The default value is :rc:`mathtext.fontset`.
(8)                     See Also
(8)                     --------
(8)                     set_math_fontfamily
(8)                     """
(8)                     return self._fontproperties.get_math_fontfamily()
(4)                 def set_math_fontfamily(self, fontfamily):
(8)                     """
(8)                     Set the font family for math text rendered by Matplotlib.
(8)                     This does only affect Matplotlib's own math renderer. It has no effect
(8)                     when rendering with TeX (``usetex=True``).
(8)                     Parameters
(8)                     ----------
(8)                     fontfamily : str
(12)                        The name of the font family.
(12)                        Available font families are defined in the
(12)                        :ref:`default matplotlibrc file
(12)                        <customizing-with-matplotlibrc-files>`.
(8)                     See Also
(8)                     --------
(8)                     get_math_fontfamily
(8)                     """
(8)                     self._fontproperties.set_math_fontfamily(fontfamily)
(4)                 def set_fontweight(self, weight):
(8)                     """
(8)                     Set the font weight.
(8)                     Parameters
(8)                     ----------
(8)                     weight : {a numeric value in range 0-1000, 'ultralight', 'light', \
(0)             'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', \
(0)             'demi', 'bold', 'heavy', 'extra bold', 'black'}
(8)                     See Also
(8)                     --------
(8)                     .font_manager.FontProperties.set_weight
(8)                     """
(8)                     self._fontproperties.set_weight(weight)
(8)                     self.stale = True
(4)                 def set_fontstretch(self, stretch):
(8)                     """
(8)                     Set the font stretch (horizontal condensation or expansion).
(8)                     Parameters
(8)                     ----------
```

```
(8)                     stretch : {a numeric value in range 0-1000, 'ultra-condensed', \
(0)             'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', \
(0)             'expanded', 'extra-expanded', 'ultra-expanded'}
(8)                     See Also
(8)                     --------
(8)                     .font_manager.FontProperties.set_stretch
(8)                     """
(8)                     self._fontproperties.set_stretch(stretch)
(8)                     self.stale = True
(4)                 def set_position(self, xy):
(8)                     """
(8)                     Set the (*x*, *y*) position of the text.
(8)                     Parameters
(8)                     ----------
(8)                     xy : (float, float)
(8)                     """
(8)                     self.set_x(xy[0])
(8)                     self.set_y(xy[1])
(4)                 def set_x(self, x):
(8)                     """
(8)                     Set the *x* position of the text.
(8)                     Parameters
(8)                     ----------
(8)                     x : float
(8)                     """
(8)                     self._x = x
(8)                     self.stale = True
(4)                 def set_y(self, y):
(8)                     """
(8)                     Set the *y* position of the text.
(8)                     Parameters
(8)                     ----------
(8)                     y : float
(8)                     """
(8)                     self._y = y
(8)                     self.stale = True
(4)                 def set_rotation(self, s):
(8)                     """
(8)                     Set the rotation of the text.
(8)                     Parameters
(8)                     ----------
(8)                     s : float or {'vertical', 'horizontal'}
(12)                        The rotation angle in degrees in mathematically positive direction
(12)                        (counterclockwise). 'horizontal' equals 0, 'vertical' equals 90.
(8)                     """
(8)                     if isinstance(s, Real):
(12)                        self._rotation = float(s) % 360
(8)                     elif cbook._str_equal(s, 'horizontal') or s is None:
(12)                        self._rotation = 0.
(8)                     elif cbook._str_equal(s, 'vertical'):
(12)                        self._rotation = 90.
(8)                     else:
(12)                        raise ValueError("rotation must be 'vertical', 'horizontal' or "
(29)                                         f"a number, not {s}")
(8)                     self.stale = True
(4)                 def set_transform_rotates_text(self, t):
(8)                     """
(8)                     Whether rotations of the transform affect the text direction.
(8)                     Parameters
(8)                     ----------
(8)                     t : bool
(8)                     """
(8)                     self._transform_rotates_text = t
(8)                     self.stale = True
(4)                 def set_verticalalignment(self, align):
(8)                     """
(8)                     Set the vertical alignment relative to the anchor point.
(8)                     See also :doc:`/gallery/text_labels_and_annotations/text_alignment`.
(8)                     Parameters
```

```
(8)                     ----------
(8)                     align : {'baseline', 'bottom', 'center', 'center_baseline', 'top'}
(8)                     """
(8)                     _api.check_in_list(
(12)                        ['top', 'bottom', 'center', 'baseline', 'center_baseline'],
(12)                        align=align)
(8)                     self._verticalalignment = align
(8)                     self.stale = True
(4)                 def set_text(self, s):
(8)                     r"""
(8)                     Set the text string *s*.
(8)                     It may contain newlines (``\n``) or math in LaTeX syntax.
(8)                     Parameters
(8)                     ----------
(8)                     s : object
(12)                        Any object gets converted to its `str` representation, except for
(12)                        ``None`` which is converted to an empty string.
(8)                     """
(8)                     s = '' if s is None else str(s)
(8)                     if s != self._text:
(12)                        self._text = s
(12)                        self.stale = True
(4)                 def _preprocess_math(self, s):
(8)                     """
(8)                     Return the string *s* after mathtext preprocessing, and the kind of
(8)                     mathtext support needed.
(8)                     - If *self* is configured to use TeX, return *s* unchanged except that
(10)                      a single space gets escaped, and the flag "TeX".
(8)                     - Otherwise, if *s* is mathtext (has an even number of unescaped
dollar
(10)                      signs) and ``parse_math`` is not set to False, return *s* and the
(10)                      flag True.
(8)                     - Otherwise, return *s* with dollar signs unescaped, and the flag
(10)                      False.
(8)                     """
(8)                     if self.get_usetex():
(12)                        if s == " ":
(16)                            s = r"\ "
(12)                        return s, "TeX"
(8)                     elif not self.get_parse_math():
(12)                        return s, False
(8)                     elif cbook.is_math_text(s):
(12)                        return s, True
(8)                     else:
(12)                        return s.replace(r"\$", "$"), False
(4)                 def set_fontproperties(self, fp):
(8)                     """
(8)                     Set the font properties that control the text.
(8)                     Parameters
(8)                     ----------
(8)                     fp : `.font_manager.FontProperties` or `str` or `pathlib.Path`
(12)                        If a `str`, it is interpreted as a fontconfig pattern parsed by
(12)                        `.FontProperties`.  If a `pathlib.Path`, it is interpreted as the
(12)                        absolute path to a font file.
(8)                     """
(8)                     self._fontproperties = FontProperties._from_any(fp).copy()
(8)                     self.stale = True
(4)                 @_docstring.kwarg_doc("bool, default: :rc:`text.usetex`")
(4)                 def set_usetex(self, usetex):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     usetex : bool or None
(12)                        Whether to render using TeX, ``None`` means to use
(12)                        :rc:`text.usetex`.
(8)                     """
(8)                     if usetex is None:
(12)                        self._usetex = mpl.rcParams['text.usetex']
(8)                     else:
```

```
(12)                        self._usetex = bool(usetex)
(8)                     self.stale = True
(4)                 def get_usetex(self):
(8)                     """Return whether this `Text` object uses TeX for rendering."""
(8)                     return self._usetex
(4)                 def set_parse_math(self, parse_math):
(8)                     """
(8)                     Override switch to disable any mathtext parsing for this `Text`.
(8)                     Parameters
(8)                     ----------
(8)                     parse_math : bool
(12)                        If False, this `Text` will never use mathtext.  If True, mathtext
(12)                        will be used if there is an even number of unescaped dollar signs.
(8)                     """
(8)                     self._parse_math = bool(parse_math)
(4)                 def get_parse_math(self):
(8)                     """Return whether mathtext parsing is considered for this `Text`."""
(8)                     return self._parse_math
(4)                 def set_fontname(self, fontname):
(8)                     """
(8)                     Alias for `set_fontfamily`.
(8)                     One-way alias only: the getter differs.
(8)                     Parameters
(8)                     ----------
(8)                     fontname : {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', \
(0)             'monospace'}
(8)                     See Also
(8)                     --------
(8)                     .font_manager.FontProperties.set_family
(8)                     """
(8)                     self.set_fontfamily(fontname)
(0)             class OffsetFrom:
(4)                 """Callable helper class for working with `Annotation`."""
(4)                 def __init__(self, artist, ref_coord, unit="points"):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     artist : `~matplotlib.artist.Artist` or `.BboxBase` or `.Transform`
(12)                        The object to compute the offset from.
(8)                     ref_coord : (float, float)
(12)                        If *artist* is an `.Artist` or `.BboxBase`, this values is
(12)                        the location to of the offset origin in fractions of the
(12)                        *artist* bounding box.
(12)                        If *artist* is a transform, the offset origin is the
(12)                        transform applied to this value.
(8)                     unit : {'points, 'pixels'}, default: 'points'
(12)                        The screen units to use (pixels or points) for the offset input.
(8)                     """
(8)                     self._artist = artist
(8)                     x, y = ref_coord  # Make copy when ref_coord is an array (and check
the shape).
(8)                     self._ref_coord = x, y
(8)                     self.set_unit(unit)
(4)                 def set_unit(self, unit):
(8)                     """
(8)                     Set the unit for input to the transform used by ``__call__``.
(8)                     Parameters
(8)                     ----------
(8)                     unit : {'points', 'pixels'}
(8)                     """
(8)                     _api.check_in_list(["points", "pixels"], unit=unit)
(8)                     self._unit = unit
(4)                 def get_unit(self):
(8)                     """Return the unit for input to the transform used by ``__call__``."""
(8)                     return self._unit
(4)                 def __call__(self, renderer):
(8)                     """
(8)                     Return the offset transform.
(8)                     Parameters
```

```
1113: (8)                   ----------
1114: (8)                   renderer : `RendererBase`
1115: (12)                      The renderer to use to compute the offset
1116: (8)                   Returns
1117: (8)                   -------
1118: (8)                   `Transform`
1119: (12)                      Maps (x, y) in pixel or point units to screen units
1120: (12)                      relative to the given artist.
1121: (8)                   """
1122: (8)                   if isinstance(self._artist, Artist):
1123: (12)                      bbox = self._artist.get_window_extent(renderer)
1124: (12)                      xf, yf = self._ref_coord
1125: (12)                      x = bbox.x0 + bbox.width * xf
1126: (12)                      y = bbox.y0 + bbox.height * yf
1127: (8)                   elif isinstance(self._artist, BboxBase):
1128: (12)                      bbox = self._artist
1129: (12)                      xf, yf = self._ref_coord
1130: (12)                      x = bbox.x0 + bbox.width * xf
1131: (12)                      y = bbox.y0 + bbox.height * yf
1132: (8)                   elif isinstance(self._artist, Transform):
1133: (12)                      x, y = self._artist.transform(self._ref_coord)
1134: (8)                   else:
1135: (12)                      _api.check_isinstance((Artist, BboxBase, Transform),
artist=self._artist)
1136: (8)                   scale = 1 if self._unit == "pixels" else renderer.points_to_pixels(1)
1137: (8)                   return Affine2D().scale(scale).translate(x, y)
1138: (0)           class _AnnotationBase:
1139: (4)               def __init__(self,
1140: (17)                           xy,
1141: (17)                           xycoords='data',
1142: (17)                           annotation_clip=None):
1143: (8)                   x, y = xy  # Make copy when xy is an array (and check the shape).
1144: (8)                   self.xy = x, y
1145: (8)                   self.xycoords = xycoords
1146: (8)                   self.set_annotation_clip(annotation_clip)
1147: (8)                   self._draggable = None
1148: (4)               def _get_xy(self, renderer, xy, coords):
1149: (8)                   x, y = xy
1150: (8)                   xcoord, ycoord = coords if isinstance(coords, tuple) else (coords,
coords)
1151: (8)                   if xcoord == 'data':
1152: (12)                      x = float(self.convert_xunits(x))
1153: (8)                   if ycoord == 'data':
1154: (12)                      y = float(self.convert_yunits(y))
1155: (8)                   return self._get_xy_transform(renderer, coords).transform((x, y))
1156: (4)               def _get_xy_transform(self, renderer, coords):
1157: (8)                   if isinstance(coords, tuple):
1158: (12)                      xcoord, ycoord = coords
1159: (12)                      from matplotlib.transforms import blended_transform_factory
1160: (12)                      tr1 = self._get_xy_transform(renderer, xcoord)
1161: (12)                      tr2 = self._get_xy_transform(renderer, ycoord)
1162: (12)                      return blended_transform_factory(tr1, tr2)
1163: (8)                   elif callable(coords):
1164: (12)                      tr = coords(renderer)
1165: (12)                      if isinstance(tr, BboxBase):
1166: (16)                          return BboxTransformTo(tr)
1167: (12)                      elif isinstance(tr, Transform):
1168: (16)                          return tr
1169: (12)                      else:
1170: (16)                          raise TypeError(
1171: (20)                              f"xycoords callable must return a BboxBase or Transform,
not a "
1172: (20)                              f"{type(tr).__name__}")
1173: (8)                   elif isinstance(coords, Artist):
1174: (12)                      bbox = coords.get_window_extent(renderer)
1175: (12)                      return BboxTransformTo(bbox)
1176: (8)                   elif isinstance(coords, BboxBase):
1177: (12)                      return BboxTransformTo(coords)
1178: (8)                   elif isinstance(coords, Transform):
```

```
1179: (12)                        return coords
1180: (8)                     elif not isinstance(coords, str):
1181: (12)                        raise TypeError(
1182: (16)                            f"'xycoords' must be an instance of str, tuple[str, str],
Artist, "
1183: (16)                            f"Transform, or Callable, not a {type(coords).__name__}")
1184: (8)                     if coords == 'data':
1185: (12)                        return self.axes.transData
1186: (8)                     elif coords == 'polar':
1187: (12)                        from matplotlib.projections import PolarAxes
1188: (12)                        tr = PolarAxes.PolarTransform(apply_theta_transforms=False)
1189: (12)                        trans = tr + self.axes.transData
1190: (12)                        return trans
1191: (8)                     try:
1192: (12)                        bbox_name, unit = coords.split()
1193: (8)                     except ValueError:  # i.e. len(coords.split()) != 2.
1194: (12)                        raise ValueError(f"{coords!r} is not a valid coordinate") from
None
1195: (8)                     bbox0, xy0 = None, None
1196: (8)                     if bbox_name == "figure":
1197: (12)                        bbox0 = self.figure.figbbox
1198: (8)                     elif bbox_name == "subfigure":
1199: (12)                        bbox0 = self.figure.bbox
1200: (8)                     elif bbox_name == "axes":
1201: (12)                        bbox0 = self.axes.bbox
1202: (8)                     if bbox0 is not None:
1203: (12)                        xy0 = bbox0.p0
1204: (8)                     elif bbox_name == "offset":
1205: (12)                        xy0 = self._get_position_xy(renderer)
1206: (8)                     else:
1207: (12)                        raise ValueError(f"{coords!r} is not a valid coordinate")
1208: (8)                     if unit == "points":
1209: (12)                        tr = Affine2D().scale(self.figure.dpi / 72)  # dpi/72 dots per
point
1210: (8)                     elif unit == "pixels":
1211: (12)                        tr = Affine2D()
1212: (8)                     elif unit == "fontsize":
1213: (12)                        tr = Affine2D().scale(self.get_size() * self.figure.dpi / 72)
1214: (8)                     elif unit == "fraction":
1215: (12)                        tr = Affine2D().scale(*bbox0.size)
1216: (8)                     else:
1217: (12)                        raise ValueError(f"{unit!r} is not a recognized unit")
1218: (8)                     return tr.translate(*xy0)
1219: (4)                 def set_annotation_clip(self, b):
1220: (8)                     """
1221: (8)                     Set the annotation's clipping behavior.
1222: (8)                     Parameters
1223: (8)                     ----------
1224: (8)                     b : bool or None
1225: (12)                        - True: The annotation will be clipped when ``self.xy`` is
1226: (14)                          outside the Axes.
1227: (12)                        - False: The annotation will always be drawn.
1228: (12)                        - None: The annotation will be clipped when ``self.xy`` is
1229: (14)                          outside the Axes and ``self.xycoords == "data"``.
1230: (8)                     """
1231: (8)                     self._annotation_clip = b
1232: (4)                 def get_annotation_clip(self):
1233: (8)                     """
1234: (8)                     Return the annotation's clipping behavior.
1235: (8)                     See `set_annotation_clip` for the meaning of return values.
1236: (8)                     """
1237: (8)                     return self._annotation_clip
1238: (4)                 def _get_position_xy(self, renderer):
1239: (8)                     """Return the pixel position of the annotated point."""
1240: (8)                     return self._get_xy(renderer, self.xy, self.xycoords)
1241: (4)                 def _check_xy(self, renderer=None):
1242: (8)                     """Check whether the annotation at *xy_pixel* should be drawn."""
1243: (8)                     if renderer is None:
1244: (12)                        renderer = self.figure._get_renderer()
```

```
1245: (8)                   b = self.get_annotation_clip()
1246: (8)                   if b or (b is None and self.xycoords == "data"):
1247: (12)                      xy_pixel = self._get_position_xy(renderer)
1248: (12)                      return self.axes.contains_point(xy_pixel)
1249: (8)                   return True
1250: (4)               def draggable(self, state=None, use_blit=False):
1251: (8)                   """
1252: (8)                   Set whether the annotation is draggable with the mouse.
1253: (8)                   Parameters
1254: (8)                   ----------
1255: (8)                   state : bool or None
1256: (12)                      - True or False: set the draggability.
1257: (12)                      - None: toggle the draggability.
1258: (8)                   use_blit : bool, default: False
1259: (12)                      Use blitting for faster image composition. For details see
1260: (12)                      :ref:`func-animation`.
1261: (8)                   Returns
1262: (8)                   -------
1263: (8)                   DraggableAnnotation or None
1264: (12)                      If the annotation is draggable, the corresponding
1265: (12)                      `.DraggableAnnotation` helper is returned.
1266: (8)                   """
1267: (8)                   from matplotlib.offsetbox import DraggableAnnotation
1268: (8)                   is_draggable = self._draggable is not None
1269: (8)                   if state is None:
1270: (12)                      state = not is_draggable
1271: (8)                   if state:
1272: (12)                      if self._draggable is None:
1273: (16)                          self._draggable = DraggableAnnotation(self, use_blit)
1274: (8)                   else:
1275: (12)                      if self._draggable is not None:
1276: (16)                          self._draggable.disconnect()
1277: (12)                      self._draggable = None
1278: (8)                   return self._draggable
1279: (0)           class Annotation(Text, _AnnotationBase):
1280: (4)               """
1281: (4)               An `.Annotation` is a `.Text` that can refer to a specific position *xy*.
1282: (4)               Optionally an arrow pointing from the text to *xy* can be drawn.
1283: (4)               Attributes
1284: (4)               ----------
1285: (4)               xy
1286: (8)                   The annotated position.
1287: (4)               xycoords
1288: (8)                   The coordinate system for *xy*.
1289: (4)               arrow_patch
1290: (8)                   A `.FancyArrowPatch` to point from *xytext* to *xy*.
1291: (4)               """
1292: (4)               def __str__(self):
1293: (8)                   return f"Annotation({self.xy[0]:g}, {self.xy[1]:g}, {self._text!r})"
1294: (4)               def __init__(self, text, xy,
1295: (17)                           xytext=None,
1296: (17)                           xycoords='data',
1297: (17)                           textcoords=None,
1298: (17)                           arrowprops=None,
1299: (17)                           annotation_clip=None,
1300: (17)                           **kwargs):
1301: (8)                   """
1302: (8)                   Annotate the point *xy* with text *text*.
1303: (8)                   In the simplest form, the text is placed at *xy*.
1304: (8)                   Optionally, the text can be displayed in another position *xytext*.
1305: (8)                   An arrow pointing from the text to the annotated point *xy* can then
1306: (8)                   be added by defining *arrowprops*.
1307: (8)                   Parameters
1308: (8)                   ----------
1309: (8)                   text : str
1310: (12)                      The text of the annotation.
1311: (8)                   xy : (float, float)
1312: (12)                      The point *(x, y)* to annotate. The coordinate system is
determined
```

```
(12)                        by *xycoords*.
(8)                     xytext : (float, float), default: *xy*
(12)                        The position *(x, y)* to place the text at. The coordinate system
(12)                        is determined by *textcoords*.
(8)                     xycoords : single or two-tuple of str or `.Artist` or `.Transform` or
\
(0)             callable, default: 'data'
(12)                        The coordinate system that *xy* is given in. The following types
(12)                        of values are supported:
(12)                        - One of the following strings:
(14)                          ====================
==========================================
(14)                          Value                Description
(14)                          ====================
==========================================
(14)                          'figure points'      Points from the lower left of the figure
(14)                          'figure pixels'      Pixels from the lower left of the figure
(14)                          'figure fraction'    Fraction of figure from lower left
(14)                          'subfigure points'   Points from the lower left of the subfigure
(14)                          'subfigure pixels'   Pixels from the lower left of the subfigure
(14)                          'subfigure fraction' Fraction of subfigure from lower left
(14)                          'axes points'        Points from lower left corner of the Axes
(14)                          'axes pixels'        Pixels from lower left corner of the Axes
(14)                          'axes fraction'      Fraction of Axes from lower left
(14)                          'data'               Use the coordinate system of the object
(35)                                               being annotated (default)
(14)                          'polar'              *(theta, r)* if not native 'data'
(35)                                               coordinates
(14)                          ====================
==========================================
(14)                          Note that 'subfigure pixels' and 'figure pixels' are the same
(14)                          for the parent figure, so users who want code that is usable in
(14)                          a subfigure can use 'subfigure pixels'.
(12)                        - An `.Artist`: *xy* is interpreted as a fraction of the artist's
(14)                          `~matplotlib.transforms.Bbox`. E.g. *(0, 0)* would be the lower
(14)                          left corner of the bounding box and *(0.5, 1)* would be the
(14)                          center top of the bounding box.
(12)                        - A `.Transform` to transform *xy* to screen coordinates.
(12)                        - A function with one of the following signatures::
(16)                            def transform(renderer) -> Bbox
(16)                            def transform(renderer) -> Transform
(14)                          where *renderer* is a `.RendererBase` subclass.
(14)                          The result of the function is interpreted like the `.Artist` and
(14)                          `.Transform` cases above.
(12)                        - A tuple *(xcoords, ycoords)* specifying separate coordinate
(14)                          systems for *x* and *y*. *xcoords* and *ycoords* must each be
(14)                          of one of the above described types.
(12)                        See :ref:`plotting-guide-annotation` for more details.
(8)                     textcoords : single or two-tuple of str or `.Artist` or `.Transform` \
(0)             or callable, default: value of *xycoords*
(12)                        The coordinate system that *xytext* is given in.
(12)                        All *xycoords* values are valid as well as the following strings:
(12)                        =================
=================================================
(12)                        Value               Description
(12)                        =================
=================================================
(12)                        'offset points'     Offset, in points, from the *xy* value
(12)                        'offset pixels'     Offset, in pixels, from the *xy* value
(12)                        'offset fontsize'   Offset, relative to fontsize, from the *xy*
value
(12)                        =================
=================================================
(8)                     arrowprops : dict, optional
(12)                        The properties used to draw a `.FancyArrowPatch` arrow between the
(12)                        positions *xy* and *xytext*.  Defaults to None, i.e. no arrow is
(12)                        drawn.
(12)                        For historical reasons there are two different ways to specify
(12)                        arrows, "simple" and "fancy":
```

```
            **Simple arrow:**
            If *arrowprops* does not contain the key 'arrowstyle' the
            allowed keys are:
            ==========  ==================================================
            Key         Description
            ==========  ==================================================
            width       The width of the arrow in points
            headwidth   The width of the base of the arrow head in points
            headlength  The length of the arrow head in points
            shrink      Fraction of total length to shrink from both ends
            ?           Any `.FancyArrowPatch` property
            ==========  ==================================================
            The arrow is attached to the edge of the text box, the exact
            position (corners or centers) depending on where it's pointing to.
            **Fancy arrow:**
            This is used if 'arrowstyle' is provided in the *arrowprops*.
            Valid keys are the following `.FancyArrowPatch` parameters:
            ===============  ===================================
            Key              Description
            ===============  ===================================
            arrowstyle       The arrow style
            connectionstyle  The connection style
            relpos           See below; default is (0.5, 0.5)
            patchA           Default is bounding box of the text
            patchB           Default is None
            shrinkA          In points. Default is 2 points
            shrinkB          In points. Default is 2 points
            mutation_scale   Default is text size (in points)
            mutation_aspect  Default is 1
            ?                Any `.FancyArrowPatch` property
            ===============  ===================================
            The exact starting point position of the arrow is defined by
            *relpos*. It's a tuple of relative coordinates of the text box,
            where (0, 0) is the lower left corner and (1, 1) is the upper
            right corner. Values <0 and >1 are supported and specify points
            outside the text box. By default (0.5, 0.5), so the starting point
            is centered in the text box.
        annotation_clip : bool or None, default: None
            Whether to clip (i.e. not draw) the annotation when the annotation
            point *xy* is outside the Axes area.
            - If *True*, the annotation will be clipped when *xy* is outside
              the Axes.
            - If *False*, the annotation will always be drawn.
            - If *None*, the annotation will be clipped when *xy* is outside
              the Axes and *xycoords* is 'data'.
        **kwargs
            Additional kwargs are passed to `.Text`.
        Returns
        -------
        `.Annotation`
        See Also
        --------
        :ref:`annotations`
        """
        _AnnotationBase.__init__(self,
                                 xy,
                                 xycoords=xycoords,
                                 annotation_clip=annotation_clip)
        if (xytext is None and
                textcoords is not None and
                textcoords != xycoords):
            _api.warn_external("You have used the `textcoords` kwarg, but "
                               "not the `xytext` kwarg.  This can lead to "
                               "surprising results.")
        if textcoords is None:
            textcoords = self.xycoords
        self._textcoords = textcoords
        if xytext is None:
            xytext = self.xy
```

```
(8)                 x, y = xytext
(8)                 self.arrowprops = arrowprops
(8)                 if arrowprops is not None:
(12)                    arrowprops = arrowprops.copy()
(12)                    if "arrowstyle" in arrowprops:
(16)                        self._arrow_relpos = arrowprops.pop("relpos", (0.5, 0.5))
(12)                    else:
(16)                        for key in ['width', 'headwidth', 'headlength', 'shrink']:
(20)                            arrowprops.pop(key, None)
(16)                        if 'frac' in arrowprops:
(20)                            _api.warn_deprecated(
(24)                                "3.8", name="the (unused) 'frac' key in 'arrowprops'")
(20)                            arrowprops.pop("frac")
(12)                    self.arrow_patch = FancyArrowPatch((0, 0), (1, 1), **arrowprops)
(8)                 else:
(12)                    self.arrow_patch = None
(8)                 Text.__init__(self, x, y, text, **kwargs)
(4)             @_api.rename_parameter("3.8", "event", "mouseevent")
(4)             def contains(self, mouseevent):
(8)                 if self._different_canvas(mouseevent):
(12)                    return False, {}
(8)                 contains, tinfo = Text.contains(self, mouseevent)
(8)                 if self.arrow_patch is not None:
(12)                    in_patch, _ = self.arrow_patch.contains(mouseevent)
(12)                    contains = contains or in_patch
(8)                 return contains, tinfo
(4)             @property
(4)             def xycoords(self):
(8)                 return self._xycoords
(4)             @xycoords.setter
(4)             def xycoords(self, xycoords):
(8)                 def is_offset(s):
(12)                    return isinstance(s, str) and s.startswith("offset")
(8)                 if (isinstance(xycoords, tuple) and any(map(is_offset, xycoords))
(16)                        or is_offset(xycoords)):
(12)                    raise ValueError("xycoords cannot be an offset coordinate")
(8)                 self._xycoords = xycoords
(4)             @property
(4)             def xyann(self):
(8)                 """
(8)                 The text position.
(8)                 See also *xytext* in `.Annotation`.
(8)                 """
(8)                 return self.get_position()
(4)             @xyann.setter
(4)             def xyann(self, xytext):
(8)                 self.set_position(xytext)
(4)             def get_anncoords(self):
(8)                 """
(8)                 Return the coordinate system to use for `.Annotation.xyann`.
(8)                 See also *xycoords* in `.Annotation`.
(8)                 """
(8)                 return self._textcoords
(4)             def set_anncoords(self, coords):
(8)                 """
(8)                 Set the coordinate system to use for `.Annotation.xyann`.
(8)                 See also *xycoords* in `.Annotation`.
(8)                 """
(8)                 self._textcoords = coords
(4)             anncoords = property(get_anncoords, set_anncoords, doc="""
(8)                 The coordinate system to use for `.Annotation.xyann`.""")
(4)             def set_figure(self, fig):
(8)                 if self.arrow_patch is not None:
(12)                    self.arrow_patch.set_figure(fig)
(8)                 Artist.set_figure(self, fig)
(4)             def update_positions(self, renderer):
(8)                 """
(8)                 Update the pixel positions of the annotation text and the arrow patch.
(8)                 """
```

```
1512: (8)                    self.set_transform(self._get_xy_transform(renderer, self.anncoords))
1513: (8)                    arrowprops = self.arrowprops
1514: (8)                    if arrowprops is None:
1515: (12)                       return
1516: (8)                    bbox = Text.get_window_extent(self, renderer)
1517: (8)                    arrow_end = x1, y1 = self._get_position_xy(renderer)  # Annotated pos.
1518: (8)                    ms = arrowprops.get("mutation_scale", self.get_size())
1519: (8)                    self.arrow_patch.set_mutation_scale(ms)
1520: (8)                    if "arrowstyle" not in arrowprops:
1521: (12)                       shrink = arrowprops.get('shrink', 0.0)
1522: (12)                       width = arrowprops.get('width', 4)
1523: (12)                       headwidth = arrowprops.get('headwidth', 12)
1524: (12)                       headlength = arrowprops.get('headlength', 12)
1525: (12)                       stylekw = dict(head_length=headlength / ms,
1526: (27)                                      head_width=headwidth / ms,
1527: (27)                                      tail_width=width / ms)
1528: (12)                       self.arrow_patch.set_arrowstyle('simple', **stylekw)
1529: (12)                       xpos = [(bbox.x0, 0), ((bbox.x0 + bbox.x1) / 2, 0.5), (bbox.x1,
1)]
1530: (12)                       ypos = [(bbox.y0, 0), ((bbox.y0 + bbox.y1) / 2, 0.5), (bbox.y1,
1)]
1531: (12)                       x, relposx = min(xpos, key=lambda v: abs(v[0] - x1))
1532: (12)                       y, relposy = min(ypos, key=lambda v: abs(v[0] - y1))
1533: (12)                       self._arrow_relpos = (relposx, relposy)
1534: (12)                       r = np.hypot(y - y1, x - x1)
1535: (12)                       shrink_pts = shrink * r / renderer.points_to_pixels(1)
1536: (12)                       self.arrow_patch.shrinkA = self.arrow_patch.shrinkB = shrink_pts
1537: (8)                    arrow_begin = bbox.p0 + bbox.size * self._arrow_relpos
1538: (8)                    self.arrow_patch.set_positions(arrow_begin, arrow_end)
1539: (8)                    if "patchA" in arrowprops:
1540: (12)                       patchA = arrowprops["patchA"]
1541: (8)                    elif self._bbox_patch:
1542: (12)                       patchA = self._bbox_patch
1543: (8)                    elif self.get_text() == "":
1544: (12)                       patchA = None
1545: (8)                    else:
1546: (12)                       pad = renderer.points_to_pixels(4)
1547: (12)                       patchA = Rectangle(
1548: (16)                           xy=(bbox.x0 - pad / 2, bbox.y0 - pad / 2),
1549: (16)                           width=bbox.width + pad, height=bbox.height + pad,
1550: (16)                           transform=IdentityTransform(), clip_on=False)
1551: (8)                    self.arrow_patch.set_patchA(patchA)
1552: (4)                @artist.allow_rasterization
1553: (4)                def draw(self, renderer):
1554: (8)                    if renderer is not None:
1555: (12)                       self._renderer = renderer
1556: (8)                    if not self.get_visible() or not self._check_xy(renderer):
1557: (12)                       return
1558: (8)                    self.update_positions(renderer)
1559: (8)                    self.update_bbox_position_size(renderer)
1560: (8)                    if self.arrow_patch is not None:  # FancyArrowPatch
1561: (12)                       if self.arrow_patch.figure is None and self.figure is not None:
1562: (16)                           self.arrow_patch.figure = self.figure
1563: (12)                       self.arrow_patch.draw(renderer)
1564: (8)                    Text.draw(self, renderer)
1565: (4)                def get_window_extent(self, renderer=None):
1566: (8)                    if not self.get_visible() or not self._check_xy(renderer):
1567: (12)                       return Bbox.unit()
1568: (8)                    if renderer is not None:
1569: (12)                       self._renderer = renderer
1570: (8)                    if self._renderer is None:
1571: (12)                       self._renderer = self.figure._get_renderer()
1572: (8)                    if self._renderer is None:
1573: (12)                       raise RuntimeError('Cannot get window extent without renderer')
1574: (8)                    self.update_positions(self._renderer)
1575: (8)                    text_bbox = Text.get_window_extent(self)
1576: (8)                    bboxes = [text_bbox]
1577: (8)                    if self.arrow_patch is not None:
1578: (12)                       bboxes.append(self.arrow_patch.get_window_extent())
```

```
1579: (8)                   return Bbox.union(bboxes)
1580: (4)               def get_tightbbox(self, renderer=None):
1581: (8)                   if not self._check_xy(renderer):
1582: (12)                      return Bbox.null()
1583: (8)                   return super().get_tightbbox(renderer)
1584: (0)           _docstring.interpd.update(Annotation=Annotation.__init__.__doc__)
```

File 67 - ticker.py:

```
1: (0)              """
2: (0)              Tick locating and formatting
3: (0)              ============================
4: (0)              This module contains classes for configuring tick locating and formatting.
5: (0)              Generic tick locators and formatters are provided, as well as domain specific
6: (0)              custom ones.
7: (0)              Although the locators know nothing about major or minor ticks, they are used
8: (0)              by the Axis class to support major and minor tick locating and formatting.
9: (0)              .. _tick_locating:
10: (0)             .. _locators:
11: (0)             Tick locating
12: (0)             -------------
13: (0)             The Locator class is the base class for all tick locators. The locators
14: (0)             handle autoscaling of the view limits based on the data limits, and the
15: (0)             choosing of tick locations. A useful semi-automatic tick locator is
16: (0)             `MultipleLocator`. It is initialized with a base, e.g., 10, and it picks
17: (0)             axis limits and ticks that are multiples of that base.
18: (0)             The Locator subclasses defined here are:
19: (0)             ========================
=======================================================
20: (0)             `AutoLocator`           `MaxNLocator` with simple defaults. This is the
default
21: (24)                                    tick locator for most plotting.
22: (0)             `MaxNLocator`           Finds up to a max number of intervals with ticks at
23: (24)                                    nice locations.
24: (0)             `LinearLocator`         Space ticks evenly from min to max.
25: (0)             `LogLocator`            Space ticks logarithmically from min to max.
26: (0)             `MultipleLocator`       Ticks and range are a multiple of base; either integer
27: (24)                                    or float.
28: (0)             `FixedLocator`          Tick locations are fixed.
29: (0)             `IndexLocator`          Locator for index plots (e.g., where
30: (24)                                    ``x = range(len(y))``).
31: (0)             `NullLocator`           No ticks.
32: (0)             `SymmetricalLogLocator` Locator for use with the symlog norm; works like
33: (24)                                    `LogLocator` for the part outside of the threshold and
34: (24)                                    adds 0 if inside the limits.
35: (0)             `AsinhLocator`          Locator for use with the asinh norm, attempting to
36: (24)                                    space ticks approximately uniformly.
37: (0)             `LogitLocator`          Locator for logit scaling.
38: (0)             `AutoMinorLocator`      Locator for minor ticks when the axis is linear and
the
39: (24)                                    major ticks are uniformly spaced. Subdivides the major
40: (24)                                    tick interval into a specified number of minor
41: (24)                                    intervals, defaulting to 4 or 5 depending on the major
42: (24)                                    interval.
43: (0)             ========================
=======================================================
44: (0)             There are a number of locators specialized for date locations - see
45: (0)             the :mod:`.dates` module.
46: (0)             You can define your own locator by deriving from Locator. You must
47: (0)             override the ``__call__`` method, which returns a sequence of locations,
48: (0)             and you will probably want to override the autoscale method to set the
49: (0)             view limits from the data limits.
50: (0)             If you want to override the default locator, use one of the above or a custom
51: (0)             locator and pass it to the x- or y-axis instance. The relevant methods are::
52: (2)               ax.xaxis.set_major_locator(xmajor_locator)
53: (2)               ax.xaxis.set_minor_locator(xminor_locator)
54: (2)               ax.yaxis.set_major_locator(ymajor_locator)
```

```
 55: (2)              ax.yaxis.set_minor_locator(yminor_locator)
 56: (0)            The default minor locator is `NullLocator`, i.e., no minor ticks on by
default.
 57: (0)            .. note::
 58: (4)                `Locator` instances should not be used with more than one
 59: (4)                `~matplotlib.axis.Axis` or `~matplotlib.axes.Axes`. So instead of::
 60: (8)                    locator = MultipleLocator(5)
 61: (8)                    ax.xaxis.set_major_locator(locator)
 62: (8)                    ax2.xaxis.set_major_locator(locator)
 63: (4)                do the following instead::
 64: (8)                    ax.xaxis.set_major_locator(MultipleLocator(5))
 65: (8)                    ax2.xaxis.set_major_locator(MultipleLocator(5))
 66: (0)            .. _formatters:
 67: (0)            Tick formatting
 68: (0)            ---------------
 69: (0)            Tick formatting is controlled by classes derived from Formatter. The formatter
 70: (0)            operates on a single tick value and returns a string to the axis.
 71: (0)            =========================
=====================================================
 72: (0)            `NullFormatter`           No labels on the ticks.
 73: (0)            `FixedFormatter`          Set the strings manually for the labels.
 74: (0)            `FuncFormatter`           User defined function sets the labels.
 75: (0)            `StrMethodFormatter`      Use string `format` method.
 76: (0)            `FormatStrFormatter`      Use an old-style sprintf format string.
 77: (0)            `ScalarFormatter`         Default formatter for scalars: autopick the format
 78: (26)                                     string.
 79: (0)            `LogFormatter`            Formatter for log axes.
 80: (0)            `LogFormatterExponent`    Format values for log axis using
 81: (26)                                     ``exponent = log_base(value)``.
 82: (0)            `LogFormatterMathtext`    Format values for log axis using
 83: (26)                                     ``exponent = log_base(value)`` using Math text.
 84: (0)            `LogFormatterSciNotation` Format values for log axis using scientific
notation.
 85: (0)            `LogitFormatter`          Probability formatter.
 86: (0)            `EngFormatter`            Format labels in engineering notation.
 87: (0)            `PercentFormatter`        Format labels as a percentage.
 88: (0)            =========================
=====================================================
 89: (0)            You can derive your own formatter from the Formatter base class by
 90: (0)            simply overriding the ``__call__`` method. The formatter class has
 91: (0)            access to the axis view and data limits.
 92: (0)            To control the major and minor tick label formats, use one of the
 93: (0)            following methods::
 94: (2)              ax.xaxis.set_major_formatter(xmajor_formatter)
 95: (2)              ax.xaxis.set_minor_formatter(xminor_formatter)
 96: (2)              ax.yaxis.set_major_formatter(ymajor_formatter)
 97: (2)              ax.yaxis.set_minor_formatter(yminor_formatter)
 98: (0)            In addition to a `.Formatter` instance, `~.Axis.set_major_formatter` and
 99: (0)            `~.Axis.set_minor_formatter` also accept a ``str`` or function.  ``str`` input
100: (0)            will be internally replaced with an autogenerated `.StrMethodFormatter` with
101: (0)            the input ``str``. For function input, a `.FuncFormatter` with the input
102: (0)            function will be generated and used.
103: (0)            See :doc:`/gallery/ticks/major_minor_demo` for an example of setting major
104: (0)            and minor ticks. See the :mod:`matplotlib.dates` module for more information
105: (0)            and examples of using date locators and formatters.
106: (0)            """
107: (0)            import itertools
108: (0)            import logging
109: (0)            import locale
110: (0)            import math
111: (0)            from numbers import Integral
112: (0)            import string
113: (0)            import numpy as np
114: (0)            import matplotlib as mpl
115: (0)            from matplotlib import _api, cbook
116: (0)            from matplotlib import transforms as mtransforms
117: (0)            _log = logging.getLogger(__name__)
118: (0)            __all__ = ('TickHelper', 'Formatter', 'FixedFormatter',
119: (11)                       'NullFormatter', 'FuncFormatter', 'FormatStrFormatter',
```

```
(11)                    'StrMethodFormatter', 'ScalarFormatter', 'LogFormatter',
(11)                    'LogFormatterExponent', 'LogFormatterMathtext',
(11)                    'LogFormatterSciNotation',
(11)                    'LogitFormatter', 'EngFormatter', 'PercentFormatter',
(11)                    'Locator', 'IndexLocator', 'FixedLocator', 'NullLocator',
(11)                    'LinearLocator', 'LogLocator', 'AutoLocator',
(11)                    'MultipleLocator', 'MaxNLocator', 'AutoMinorLocator',
(11)                    'SymmetricalLogLocator', 'AsinhLocator', 'LogitLocator')
(0)             class _DummyAxis:
(4)                 __name__ = "dummy"
(4)                 def __init__(self, minpos=0):
(8)                     self._data_interval = (0, 1)
(8)                     self._view_interval = (0, 1)
(8)                     self._minpos = minpos
(4)                 def get_view_interval(self):
(8)                     return self._view_interval
(4)                 def set_view_interval(self, vmin, vmax):
(8)                     self._view_interval = (vmin, vmax)
(4)                 def get_minpos(self):
(8)                     return self._minpos
(4)                 def get_data_interval(self):
(8)                     return self._data_interval
(4)                 def set_data_interval(self, vmin, vmax):
(8)                     self._data_interval = (vmin, vmax)
(4)                 def get_tick_space(self):
(8)                     return 9
(0)             class TickHelper:
(4)                 axis = None
(4)                 def set_axis(self, axis):
(8)                     self.axis = axis
(4)                 def create_dummy_axis(self, **kwargs):
(8)                     if self.axis is None:
(12)                        self.axis = _DummyAxis(**kwargs)
(0)             class Formatter(TickHelper):
(4)                 """
(4)                 Create a string based on a tick value and location.
(4)                 """
(4)                 locs = []
(4)                 def __call__(self, x, pos=None):
(8)                     """
(8)                     Return the format for tick value *x* at position pos.
(8)                     ``pos=None`` indicates an unspecified location.
(8)                     """
(8)                     raise NotImplementedError('Derived must override')
(4)                 def format_ticks(self, values):
(8)                     """Return the tick labels for all the ticks at once."""
(8)                     self.set_locs(values)
(8)                     return [self(value, i) for i, value in enumerate(values)]
(4)                 def format_data(self, value):
(8)                     """
(8)                     Return the full string representation of the value with the
(8)                     position unspecified.
(8)                     """
(8)                     return self.__call__(value)
(4)                 def format_data_short(self, value):
(8)                     """
(8)                     Return a short string version of the tick value.
(8)                     Defaults to the position-independent long value.
(8)                     """
(8)                     return self.format_data(value)
(4)                 def get_offset(self):
(8)                     return ''
(4)                 def set_locs(self, locs):
(8)                     """
(8)                     Set the locations of the ticks.
(8)                     This method is called before computing the tick labels because some
(8)                     formatters need to know all tick locations to do so.
(8)                     """
(8)                     self.locs = locs
```

```
(4)                 @staticmethod
(4)                 def fix_minus(s):
(8)                     """
(8)                     Some classes may want to replace a hyphen for minus with the proper
(8)                     Unicode symbol (U+2212) for typographical correctness.  This is a
(8)                     helper method to perform such a replacement when it is enabled via
(8)                     :rc:`axes.unicode_minus`.
(8)                     """
(8)                     return (s.replace('-', '\N{MINUS SIGN}')
(16)                            if mpl.rcParams['axes.unicode_minus']
(16)                            else s)
(4)                 def _set_locator(self, locator):
(8)                     """Subclasses may want to override this to set a locator."""
(8)                     pass
(0)             class NullFormatter(Formatter):
(4)                 """Always return the empty string."""
(4)                 def __call__(self, x, pos=None):
(8)                     return ''
(0)             class FixedFormatter(Formatter):
(4)                 """
(4)                 Return fixed strings for tick labels based only on position, not value.
(4)                 .. note::
(8)                     `.FixedFormatter` should only be used together with `.FixedLocator`.
(8)                     Otherwise, the labels may end up in unexpected positions.
(4)                 """
(4)                 def __init__(self, seq):
(8)                     """Set the sequence *seq* of strings that will be used for labels."""
(8)                     self.seq = seq
(8)                     self.offset_string = ''
(4)                 def __call__(self, x, pos=None):
(8)                     """
(8)                     Return the label that matches the position, regardless of the value.
(8)                     For positions ``pos < len(seq)``, return ``seq[i]`` regardless of
(8)                     *x*. Otherwise return empty string. ``seq`` is the sequence of
(8)                     strings that this object was initialized with.
(8)                     """
(8)                     if pos is None or pos >= len(self.seq):
(12)                        return ''
(8)                     else:
(12)                        return self.seq[pos]
(4)                 def get_offset(self):
(8)                     return self.offset_string
(4)                 def set_offset_string(self, ofs):
(8)                     self.offset_string = ofs
(0)             class FuncFormatter(Formatter):
(4)                 """
(4)                 Use a user-defined function for formatting.
(4)                 The function should take in two inputs (a tick value ``x`` and a
(4)                 position ``pos``), and return a string containing the corresponding
(4)                 tick label.
(4)                 """
(4)                 def __init__(self, func):
(8)                     self.func = func
(8)                     self.offset_string = ""
(4)                 def __call__(self, x, pos=None):
(8)                     """
(8)                     Return the value of the user defined function.
(8)                     *x* and *pos* are passed through as-is.
(8)                     """
(8)                     return self.func(x, pos)
(4)                 def get_offset(self):
(8)                     return self.offset_string
(4)                 def set_offset_string(self, ofs):
(8)                     self.offset_string = ofs
(0)             class FormatStrFormatter(Formatter):
(4)                 """
(4)                 Use an old-style ('%' operator) format string to format the tick.
(4)                 The format string should have a single variable format (%) in it.
(4)                 It will be applied to the value (not the position) of the tick.
```

```
258: (4)                Negative numeric values (e.g., -1) will use a dash, not a Unicode minus;
259: (4)                use mathtext to get a Unicode minus by wrapping the format specifier with
$
260: (4)                (e.g. "$%g$").
261: (4)                """
262: (4)                def __init__(self, fmt):
263: (8)                    self.fmt = fmt
264: (4)                def __call__(self, x, pos=None):
265: (8)                    """
266: (8)                    Return the formatted label string.
267: (8)                    Only the value *x* is formatted. The position is ignored.
268: (8)                    """
269: (8)                    return self.fmt % x
270: (0)            class _UnicodeMinusFormat(string.Formatter):
271: (4)                """
272: (4)                A specialized string formatter so that `.StrMethodFormatter` respects
273: (4)                :rc:`axes.unicode_minus`.  This implementation relies on the fact that the
274: (4)                format string is only ever called with kwargs *x* and *pos*, so it blindly
275: (4)                replaces dashes by unicode minuses without further checking.
276: (4)                """
277: (4)                def format_field(self, value, format_spec):
278: (8)                    return Formatter.fix_minus(super().format_field(value, format_spec))
279: (0)            class StrMethodFormatter(Formatter):
280: (4)                """
281: (4)                Use a new-style format string (as used by `str.format`) to format the
tick.
282: (4)                The field used for the tick value must be labeled *x* and the field used
283: (4)                for the tick position must be labeled *pos*.
284: (4)                The formatter will respect :rc:`axes.unicode_minus` when formatting
285: (4)                negative numeric values.
286: (4)                It is typically unnecessary to explicitly construct `.StrMethodFormatter`
287: (4)                objects, as `~.Axis.set_major_formatter` directly accepts the format
string
288: (4)                itself.
289: (4)                """
290: (4)                def __init__(self, fmt):
291: (8)                    self.fmt = fmt
292: (4)                def __call__(self, x, pos=None):
293: (8)                    """
294: (8)                    Return the formatted label string.
295: (8)                    *x* and *pos* are passed to `str.format` as keyword arguments
296: (8)                    with those exact names.
297: (8)                    """
298: (8)                    return _UnicodeMinusFormat().format(self.fmt, x=x, pos=pos)
299: (0)            class ScalarFormatter(Formatter):
300: (4)                """
301: (4)                Format tick values as a number.
302: (4)                Parameters
303: (4)                ----------
304: (4)                useOffset : bool or float, default: :rc:`axes.formatter.useoffset`
305: (8)                    Whether to use offset notation. See `.set_useOffset`.
306: (4)                useMathText : bool, default: :rc:`axes.formatter.use_mathtext`
307: (8)                    Whether to use fancy math formatting. See `.set_useMathText`.
308: (4)                useLocale : bool, default: :rc:`axes.formatter.use_locale`.
309: (8)                    Whether to use locale settings for decimal sign and positive sign.
310: (8)                    See `.set_useLocale`.
311: (4)                Notes
312: (4)                -----
313: (4)                In addition to the parameters above, the formatting of scientific vs.
314: (4)                floating point representation can be configured via `.set_scientific`
315: (4)                and `.set_powerlimits`).
316: (4)                **Offset notation and scientific notation**
317: (4)                Offset notation and scientific notation look quite similar at first sight.
318: (4)                Both split some information from the formatted tick values and display it
319: (4)                at the end of the axis.
320: (4)                - The scientific notation splits up the order of magnitude, i.e. a
321: (6)                  multiplicative scaling factor, e.g. ``1e6``.
322: (4)                - The offset notation separates an additive constant, e.g. ``+1e6``. The
323: (6)                  offset notation label is always prefixed with a ``+`` or ``-`` sign
```

```
  and is thus distinguishable from the order of magnitude label.
The following plot with x limits ``1_000_000`` to ``1_000_010`` illustrates
the different formatting. Note the labels at the right edge of the x axis.
.. plot::
    lim = (1_000_000, 1_000_010)
    fig, (ax1, ax2, ax3) = plt.subplots(3, 1, gridspec_kw={'hspace': 2})
    ax1.set(title='offset_notation', xlim=lim)
    ax2.set(title='scientific notation', xlim=lim)
    ax2.xaxis.get_major_formatter().set_useOffset(False)
    ax3.set(title='floating point notation', xlim=lim)
    ax3.xaxis.get_major_formatter().set_useOffset(False)
    ax3.xaxis.get_major_formatter().set_scientific(False)
"""
def __init__(self, useOffset=None, useMathText=None, useLocale=None):
    if useOffset is None:
        useOffset = mpl.rcParams['axes.formatter.useoffset']
    self._offset_threshold = \
        mpl.rcParams['axes.formatter.offset_threshold']
    self.set_useOffset(useOffset)
    self._usetex = mpl.rcParams['text.usetex']
    self.set_useMathText(useMathText)
    self.orderOfMagnitude = 0
    self.format = ''
    self._scientific = True
    self._powerlimits = mpl.rcParams['axes.formatter.limits']
    self.set_useLocale(useLocale)
def get_useOffset(self):
    """
    Return whether automatic mode for offset notation is active.
    This returns True if ``set_useOffset(True)``; it returns False if an
    explicit offset was set, e.g. ``set_useOffset(1000)``.
    See Also
    --------
    ScalarFormatter.set_useOffset
    """
    return self._useOffset
def set_useOffset(self, val):
    """
    Set whether to use offset notation.
    When formatting a set numbers whose value is large compared to their
    range, the formatter can separate an additive constant. This can
    shorten the formatted numbers so that they are less likely to overlap
    when drawn on an axis.
    Parameters
    ----------
    val : bool or float
        - If False, do not use offset notation.
        - If True (=automatic mode), use offset notation if it can make
          the residual numbers significantly shorter. The exact behavior
          is controlled by :rc:`axes.formatter.offset_threshold`.
        - If a number, force an offset of the given value.
    Examples
    --------
    With active offset notation, the values
    ``100_000, 100_002, 100_004, 100_006, 100_008``
    will be formatted as ``0, 2, 4, 6, 8`` plus an offset ``+1e5``, which
    is written to the edge of the axis.
    """
    if val in [True, False]:
        self.offset = 0
        self._useOffset = val
    else:
        self._useOffset = False
        self.offset = val
useOffset = property(fget=get_useOffset, fset=set_useOffset)
def get_useLocale(self):
    """
    Return whether locale settings are used for formatting.
```

```
(8)                     See Also
(8)                     --------
(8)                     ScalarFormatter.set_useLocale
(8)                     """
(8)                     return self._useLocale
(4)                 def set_useLocale(self, val):
(8)                     """
(8)                     Set whether to use locale settings for decimal sign and positive sign.
(8)                     Parameters
(8)                     ----------
(8)                     val : bool or None
(12)                        *None* resets to :rc:`axes.formatter.use_locale`.
(8)                     """
(8)                     if val is None:
(12)                        self._useLocale = mpl.rcParams['axes.formatter.use_locale']
(8)                     else:
(12)                        self._useLocale = val
(4)                 useLocale = property(fget=get_useLocale, fset=set_useLocale)
(4)                 def _format_maybe_minus_and_locale(self, fmt, arg):
(8)                     """
(8)                     Format *arg* with *fmt*, applying Unicode minus and locale if desired.
(8)                     """
(8)                     return self.fix_minus(
(16)                            (",".join(locale.format_string(part, (arg,),
True).replace(",", "{,}")
(26)                                      for part in fmt.split(",")) if self._useMathText
(17)                             else locale.format_string(fmt, (arg,), True))
(16)                            if self._useLocale
(16)                            else fmt % arg)
(4)                 def get_useMathText(self):
(8)                     """
(8)                     Return whether to use fancy math formatting.
(8)                     See Also
(8)                     --------
(8)                     ScalarFormatter.set_useMathText
(8)                     """
(8)                     return self._useMathText
(4)                 def set_useMathText(self, val):
(8)                     r"""
(8)                     Set whether to use fancy math formatting.
(8)                     If active, scientific notation is formatted as :math:`1.2 \times
10^3`.
(8)                     Parameters
(8)                     ----------
(8)                     val : bool or None
(12)                        *None* resets to :rc:`axes.formatter.use_mathtext`.
(8)                     """
(8)                     if val is None:
(12)                        self._useMathText = mpl.rcParams['axes.formatter.use_mathtext']
(12)                        if self._useMathText is False:
(16)                            try:
(20)                                from matplotlib import font_manager
(20)                                ufont = font_manager.findfont(
(24)                                    font_manager.FontProperties(
(28)                                        mpl.rcParams["font.family"]
(24)                                    ),
(24)                                    fallback_to_default=False,
(20)                                )
(16)                            except ValueError:
(20)                                ufont = None
(16)                            if ufont == str(cbook._get_data_path("fonts/ttf/cmr10.ttf")):
(20)                                _api.warn_external(
(24)                                    "cmr10 font should ideally be used with "
(24)                                    "mathtext, set axes.formatter.use_mathtext to True"
(20)                                )
(8)                     else:
(12)                        self._useMathText = val
(4)                 useMathText = property(fget=get_useMathText, fset=set_useMathText)
(4)                 def __call__(self, x, pos=None):
```

```
(8)                 """
(8)                 Return the format for tick value *x* at position *pos*.
(8)                 """
(8)                 if len(self.locs) == 0:
(12)                    return ''
(8)                 else:
(12)                    xp = (x - self.offset) / (10. ** self.orderOfMagnitude)
(12)                    if abs(xp) < 1e-8:
(16)                        xp = 0
(12)                    return self._format_maybe_minus_and_locale(self.format, xp)
(4)             def set_scientific(self, b):
(8)                 """
(8)                 Turn scientific notation on or off.
(8)                 See Also
(8)                 --------
(8)                 ScalarFormatter.set_powerlimits
(8)                 """
(8)                 self._scientific = bool(b)
(4)             def set_powerlimits(self, lims):
(8)                 r"""
(8)                 Set size thresholds for scientific notation.
(8)                 Parameters
(8)                 ----------
(8)                 lims : (int, int)
(12)                    A tuple *(min_exp, max_exp)* containing the powers of 10 that
(12)                    determine the switchover threshold. For a number representable as
(12)                    :math:`a \times 10^\mathrm{exp}` with :math:`1 <= |a| < 10`,
(12)                    scientific notation will be used if ``exp <= min_exp`` or
(12)                    ``exp >= max_exp``.
(12)                    The default limits are controlled by :rc:`axes.formatter.limits`.
(12)                    In particular numbers with *exp* equal to the thresholds are
(12)                    written in scientific notation.
(12)                    Typically, *min_exp* will be negative and *max_exp* will be
(12)                    positive.
(12)                    For example, ``formatter.set_powerlimits((-3, 4))`` will provide
(12)                    the following formatting:
(12)                    :math:`1 \times 10^{-3}, 9.9 \times 10^{-3}, 0.01,`
(12)                    :math:`9999, 1 \times 10^4`.
(8)                 See Also
(8)                 --------
(8)                 ScalarFormatter.set_scientific
(8)                 """
(8)                 if len(lims) != 2:
(12)                    raise ValueError("'lims' must be a sequence of length 2")
(8)                 self._powerlimits = lims
(4)             def format_data_short(self, value):
(8)                 if value is np.ma.masked:
(12)                    return ""
(8)                 if isinstance(value, Integral):
(12)                    fmt = "%d"
(8)                 else:
(12)                    if getattr(self.axis, "__name__", "") in ["xaxis", "yaxis"]:
(16)                        if self.axis.__name__ == "xaxis":
(20)                            axis_trf = self.axis.axes.get_xaxis_transform()
(20)                            axis_inv_trf = axis_trf.inverted()
(20)                            screen_xy = axis_trf.transform((value, 0))
(20)                            neighbor_values = axis_inv_trf.transform(
(24)                                screen_xy + [[-1, 0], [+1, 0]])[:, 0]
(16)                        else:  # yaxis:
(20)                            axis_trf = self.axis.axes.get_yaxis_transform()
(20)                            axis_inv_trf = axis_trf.inverted()
(20)                            screen_xy = axis_trf.transform((0, value))
(20)                            neighbor_values = axis_inv_trf.transform(
(24)                                screen_xy + [[0, -1], [0, +1]])[:, 1]
(16)                        delta = abs(neighbor_values - value).max()
(12)                    else:
(16)                        a, b = self.axis.get_view_interval()
(16)                        delta = (b - a) / 1e4
(12)                    fmt = f"%-#.{cbook._g_sig_digits(value, delta)}g"
```

```
(8)                 return self._format_maybe_minus_and_locale(fmt, value)
(4)             def format_data(self, value):
(8)                 e = math.floor(math.log10(abs(value)))
(8)                 s = round(value / 10**e, 10)
(8)                 significand = self._format_maybe_minus_and_locale(
(12)                    "%d" if s % 1 == 0 else "%1.10g", s)
(8)                 if e == 0:
(12)                    return significand
(8)                 exponent = self._format_maybe_minus_and_locale("%d", e)
(8)                 if self._useMathText or self._usetex:
(12)                    exponent = "10^{%s}" % exponent
(12)                    return (exponent if s == 1  # reformat 1x10^y as 10^y
(20)                            else rf"{significand} \times {exponent}")
(8)                 else:
(12)                    return f"{significand}e{exponent}"
(4)             def get_offset(self):
(8)                 """
(8)                 Return scientific notation, plus offset.
(8)                 """
(8)                 if len(self.locs) == 0:
(12)                    return ''
(8)                 if self.orderOfMagnitude or self.offset:
(12)                    offsetStr = ''
(12)                    sciNotStr = ''
(12)                    if self.offset:
(16)                        offsetStr = self.format_data(self.offset)
(16)                        if self.offset > 0:
(20)                            offsetStr = '+' + offsetStr
(12)                    if self.orderOfMagnitude:
(16)                        if self._usetex or self._useMathText:
(20)                            sciNotStr = self.format_data(10 ** self.orderOfMagnitude)
(16)                        else:
(20)                            sciNotStr = '1e%d' % self.orderOfMagnitude
(12)                    if self._useMathText or self._usetex:
(16)                        if sciNotStr != '':
(20)                            sciNotStr = r'\times\mathdefault{%s}' % sciNotStr
(16)                        s = fr'${sciNotStr}\mathdefault{{{offsetStr}}}$'
(12)                    else:
(16)                        s = ''.join((sciNotStr, offsetStr))
(12)                    return self.fix_minus(s)
(8)                 return ''
(4)             def set_locs(self, locs):
(8)                 self.locs = locs
(8)                 if len(self.locs) > 0:
(12)                    if self._useOffset:
(16)                        self._compute_offset()
(12)                    self._set_order_of_magnitude()
(12)                    self._set_format()
(4)             def _compute_offset(self):
(8)                 locs = self.locs
(8)                 vmin, vmax = sorted(self.axis.get_view_interval())
(8)                 locs = np.asarray(locs)
(8)                 locs = locs[(vmin <= locs) & (locs <= vmax)]
(8)                 if not len(locs):
(12)                    self.offset = 0
(12)                    return
(8)                 lmin, lmax = locs.min(), locs.max()
(8)                 if lmin == lmax or lmin <= 0 <= lmax:
(12)                    self.offset = 0
(12)                    return
(8)                 abs_min, abs_max = sorted([abs(float(lmin)), abs(float(lmax))])
(8)                 sign = math.copysign(1, lmin)
(8)                 oom_max = np.ceil(math.log10(abs_max))
(8)                 oom = 1 + next(oom for oom in itertools.count(oom_max, -1)
(23)                               if abs_min // 10 ** oom != abs_max // 10 ** oom)
(8)                 if (abs_max - abs_min) / 10 ** oom <= 1e-2:
(12)                    oom = 1 + next(oom for oom in itertools.count(oom_max, -1)
(27)                                   if abs_max // 10 ** oom - abs_min // 10 ** oom > 1)
(8)                 n = self._offset_threshold - 1
```

```
(8)                     self.offset = (sign * (abs_max // 10 ** oom) * 10 ** oom
(23)                                   if abs_max // 10 ** oom >= 10**n
(23)                                   else 0)
(4)                 def _set_order_of_magnitude(self):
(8)                     if not self._scientific:
(12)                        self.orderOfMagnitude = 0
(12)                        return
(8)                     if self._powerlimits[0] == self._powerlimits[1] != 0:
(12)                        self.orderOfMagnitude = self._powerlimits[0]
(12)                        return
(8)                     vmin, vmax = sorted(self.axis.get_view_interval())
(8)                     locs = np.asarray(self.locs)
(8)                     locs = locs[(vmin <= locs) & (locs <= vmax)]
(8)                     locs = np.abs(locs)
(8)                     if not len(locs):
(12)                        self.orderOfMagnitude = 0
(12)                        return
(8)                     if self.offset:
(12)                        oom = math.floor(math.log10(vmax - vmin))
(8)                     else:
(12)                        val = locs.max()
(12)                        if val == 0:
(16)                            oom = 0
(12)                        else:
(16)                            oom = math.floor(math.log10(val))
(8)                     if oom <= self._powerlimits[0]:
(12)                        self.orderOfMagnitude = oom
(8)                     elif oom >= self._powerlimits[1]:
(12)                        self.orderOfMagnitude = oom
(8)                     else:
(12)                        self.orderOfMagnitude = 0
(4)                 def _set_format(self):
(8)                     if len(self.locs) < 2:
(12)                        _locs = [*self.locs, *self.axis.get_view_interval()]
(8)                     else:
(12)                        _locs = self.locs
(8)                     locs = (np.asarray(_locs) - self.offset) / 10. **
self.orderOfMagnitude
(8)                     loc_range = np.ptp(locs)
(8)                     if loc_range == 0:
(12)                        loc_range = np.max(np.abs(locs))
(8)                     if loc_range == 0:
(12)                        loc_range = 1
(8)                     if len(self.locs) < 2:
(12)                        locs = locs[:-2]
(8)                     loc_range_oom = int(math.floor(math.log10(loc_range)))
(8)                     sigfigs = max(0, 3 - loc_range_oom)
(8)                     thresh = 1e-3 * 10 ** loc_range_oom
(8)                     while sigfigs >= 0:
(12)                        if np.abs(locs - np.round(locs, decimals=sigfigs)).max() < thresh:
(16)                            sigfigs -= 1
(12)                        else:
(16)                            break
(8)                     sigfigs += 1
(8)                     self.format = f'%1.{sigfigs}f'
(8)                     if self._usetex or self._useMathText:
(12)                        self.format = r'$\mathdefault{%s}$' % self.format
(0)             class LogFormatter(Formatter):
(4)                 """
(4)                 Base class for formatting ticks on a log or symlog scale.
(4)                 It may be instantiated directly, or subclassed.
(4)                 Parameters
(4)                 ----------
(4)                 base : float, default: 10.
(8)                     Base of the logarithm used in all calculations.
(4)                 labelOnlyBase : bool, default: False
(8)                     If True, label ticks only at integer powers of base.
(8)                     This is normally True for major ticks and False for
(8)                     minor ticks.
```

```
(4)                 minor_thresholds : (subset, all), default: (1, 0.4)
(8)                     If labelOnlyBase is False, these two numbers control
(8)                     the labeling of ticks that are not at integer powers of
(8)                     base; normally these are the minor ticks. The controlling
(8)                     parameter is the log of the axis data range.  In the typical
(8)                     case where base is 10 it is the number of decades spanned
(8)                     by the axis, so we can call it 'numdec'. If ``numdec <= all``,
(8)                     all minor ticks will be labeled.  If ``all < numdec <= subset``,
(8)                     then only a subset of minor ticks will be labeled, so as to
(8)                     avoid crowding. If ``numdec > subset`` then no minor ticks will
(8)                     be labeled.
(4)                 linthresh : None or float, default: None
(8)                     If a symmetric log scale is in use, its ``linthresh``
(8)                     parameter must be supplied here.
(4)                 Notes
(4)                 -----
(4)                 The `set_locs` method must be called to enable the subsetting
(4)                 logic controlled by the ``minor_thresholds`` parameter.
(4)                 In some cases such as the colorbar, there is no distinction between
(4)                 major and minor ticks; the tick locations might be set manually,
(4)                 or by a locator that puts ticks at integer powers of base and
(4)                 at intermediate locations.  For this situation, disable the
(4)                 minor_thresholds logic by using ``minor_thresholds=(np.inf, np.inf)``,
(4)                 so that all ticks will be labeled.
(4)                 To disable labeling of minor ticks when 'labelOnlyBase' is False,
(4)                 use ``minor_thresholds=(0, 0)``.  This is the default for the
(4)                 "classic" style.
(4)                 Examples
(4)                 --------
(4)                 To label a subset of minor ticks when the view limits span up
(4)                 to 2 decades, and all of the ticks when zoomed in to 0.5 decades
(4)                 or less, use ``minor_thresholds=(2, 0.5)``.
(4)                 To label all minor ticks when the view limits span up to 1.5
(4)                 decades, use ``minor_thresholds=(1.5, 1.5)``.
(4)                 """
(4)                 def __init__(self, base=10.0, labelOnlyBase=False,
(17)                             minor_thresholds=None,
(17)                             linthresh=None):
(8)                     self.set_base(base)
(8)                     self.set_label_minor(labelOnlyBase)
(8)                     if minor_thresholds is None:
(12)                        if mpl.rcParams['_internal.classic_mode']:
(16)                            minor_thresholds = (0, 0)
(12)                        else:
(16)                            minor_thresholds = (1, 0.4)
(8)                     self.minor_thresholds = minor_thresholds
(8)                     self._sublabels = None
(8)                     self._linthresh = linthresh
(4)                 def set_base(self, base):
(8)                     """
(8)                     Change the *base* for labeling.
(8)                     .. warning::
(11)                       Should always match the base used for :class:`LogLocator`
(8)                     """
(8)                     self._base = float(base)
(4)                 def set_label_minor(self, labelOnlyBase):
(8)                     """
(8)                     Switch minor tick labeling on or off.
(8)                     Parameters
(8)                     ----------
(8)                     labelOnlyBase : bool
(12)                        If True, label ticks only at integer powers of base.
(8)                     """
(8)                     self.labelOnlyBase = labelOnlyBase
(4)                 def set_locs(self, locs=None):
(8)                     """
(8)                     Use axis view limits to control which ticks are labeled.
(8)                     The *locs* parameter is ignored in the present algorithm.
(8)                     """
```

```
(8)                 if np.isinf(self.minor_thresholds[0]):
(12)                    self._sublabels = None
(12)                    return
(8)                 linthresh = self._linthresh
(8)                 if linthresh is None:
(12)                    try:
(16)                        linthresh = self.axis.get_transform().linthresh
(12)                    except AttributeError:
(16)                        pass
(8)                 vmin, vmax = self.axis.get_view_interval()
(8)                 if vmin > vmax:
(12)                    vmin, vmax = vmax, vmin
(8)                 if linthresh is None and vmin <= 0:
(12)                    self._sublabels = {1}  # label powers of base
(12)                    return
(8)                 b = self._base
(8)                 if linthresh is not None:  # symlog
(12)                    numdec = 0
(12)                    if vmin < -linthresh:
(16)                        rhs = min(vmax, -linthresh)
(16)                        numdec += math.log(vmin / rhs) / math.log(b)
(12)                    if vmax > linthresh:
(16)                        lhs = max(vmin, linthresh)
(16)                        numdec += math.log(vmax / lhs) / math.log(b)
(8)                 else:
(12)                    vmin = math.log(vmin) / math.log(b)
(12)                    vmax = math.log(vmax) / math.log(b)
(12)                    numdec = abs(vmax - vmin)
(8)                 if numdec > self.minor_thresholds[0]:
(12)                    self._sublabels = {1}
(8)                 elif numdec > self.minor_thresholds[1]:
(12)                    c = np.geomspace(1, b, int(b)//2 + 1)
(12)                    self._sublabels = set(np.round(c))
(8)                 else:
(12)                    self._sublabels = set(np.arange(1, b + 1))
(4)             def _num_to_string(self, x, vmin, vmax):
(8)                 if x > 10000:
(12)                    s = '%1.0e' % x
(8)                 elif x < 1:
(12)                    s = '%1.0e' % x
(8)                 else:
(12)                    s = self._pprint_val(x, vmax - vmin)
(8)                 return s
(4)             def __call__(self, x, pos=None):
(8)                 if x == 0.0:  # Symlog
(12)                    return '0'
(8)                 x = abs(x)
(8)                 b = self._base
(8)                 fx = math.log(x) / math.log(b)
(8)                 is_x_decade = _is_close_to_int(fx)
(8)                 exponent = round(fx) if is_x_decade else np.floor(fx)
(8)                 coeff = round(b ** (fx - exponent))
(8)                 if self.labelOnlyBase and not is_x_decade:
(12)                    return ''
(8)                 if self._sublabels is not None and coeff not in self._sublabels:
(12)                    return ''
(8)                 vmin, vmax = self.axis.get_view_interval()
(8)                 vmin, vmax = mtransforms.nonsingular(vmin, vmax, expander=0.05)
(8)                 s = self._num_to_string(x, vmin, vmax)
(8)                 return self.fix_minus(s)
(4)             def format_data(self, value):
(8)                 with cbook._setattr_cm(self, labelOnlyBase=False):
(12)                    return cbook.strip_math(self.__call__(value))
(4)             def format_data_short(self, value):
(8)                 return ('%-12g' % value).rstrip()
(4)             def _pprint_val(self, x, d):
(8)                 if abs(x) < 1e4 and x == int(x):
(12)                    return '%d' % x
(8)                 fmt = ('%1.3e' if d < 1e-2 else
```

```
803: (15)                           '%1.3f' if d <= 1 else
804: (15)                           '%1.2f' if d <= 10 else
805: (15)                           '%1.1f' if d <= 1e5 else
806: (15)                           '%1.1e')
807: (8)                    s = fmt % x
808: (8)                    tup = s.split('e')
809: (8)                    if len(tup) == 2:
810: (12)                       mantissa = tup[0].rstrip('0').rstrip('.')
811: (12)                       exponent = int(tup[1])
812: (12)                       if exponent:
813: (16)                           s = '%se%d' % (mantissa, exponent)
814: (12)                       else:
815: (16)                           s = mantissa
816: (8)                    else:
817: (12)                       s = s.rstrip('0').rstrip('.')
818: (8)                    return s
819: (0)            class LogFormatterExponent(LogFormatter):
820: (4)                """
821: (4)                Format values for log axis using ``exponent = log_base(value)``.
822: (4)                """
823: (4)                def _num_to_string(self, x, vmin, vmax):
824: (8)                    fx = math.log(x) / math.log(self._base)
825: (8)                    if abs(fx) > 10000:
826: (12)                       s = '%1.0g' % fx
827: (8)                    elif abs(fx) < 1:
828: (12)                       s = '%1.0g' % fx
829: (8)                    else:
830: (12)                       fd = math.log(vmax - vmin) / math.log(self._base)
831: (12)                       s = self._pprint_val(fx, fd)
832: (8)                    return s
833: (0)            class LogFormatterMathtext(LogFormatter):
834: (4)                """
835: (4)                Format values for log axis using ``exponent = log_base(value)``.
836: (4)                """
837: (4)                def _non_decade_format(self, sign_string, base, fx, usetex):
838: (8)                    """Return string for non-decade locations."""
839: (8)                    return r'$\mathdefault{%s%s^{%.2f}}$' % (sign_string, base, fx)
840: (4)                def __call__(self, x, pos=None):
841: (8)                    if x == 0:  # Symlog
842: (12)                       return r'$\mathdefault{0}$'
843: (8)                    sign_string = '-' if x < 0 else ''
844: (8)                    x = abs(x)
845: (8)                    b = self._base
846: (8)                    fx = math.log(x) / math.log(b)
847: (8)                    is_x_decade = _is_close_to_int(fx)
848: (8)                    exponent = round(fx) if is_x_decade else np.floor(fx)
849: (8)                    coeff = round(b ** (fx - exponent))
850: (8)                    if self.labelOnlyBase and not is_x_decade:
851: (12)                       return ''
852: (8)                    if self._sublabels is not None and coeff not in self._sublabels:
853: (12)                       return ''
854: (8)                    if is_x_decade:
855: (12)                       fx = round(fx)
856: (8)                    if b % 1 == 0.0:
857: (12)                       base = '%d' % b
858: (8)                    else:
859: (12)                       base = '%s' % b
860: (8)                    if abs(fx) < mpl.rcParams['axes.formatter.min_exponent']:
861: (12)                       return r'$\mathdefault{%s%g}$' % (sign_string, x)
862: (8)                    elif not is_x_decade:
863: (12)                       usetex = mpl.rcParams['text.usetex']
864: (12)                       return self._non_decade_format(sign_string, base, fx, usetex)
865: (8)                    else:
866: (12)                       return r'$\mathdefault{%s%s^{%d}}$' % (sign_string, base, fx)
867: (0)            class LogFormatterSciNotation(LogFormatterMathtext):
868: (4)                """
869: (4)                Format values following scientific notation in a logarithmic axis.
870: (4)                """
871: (4)                def _non_decade_format(self, sign_string, base, fx, usetex):
```

```
(8)                 """Return string for non-decade locations."""
(8)                 b = float(base)
(8)                 exponent = math.floor(fx)
(8)                 coeff = b ** (fx - exponent)
(8)                 if _is_close_to_int(coeff):
(12)                    coeff = round(coeff)
(8)                 return r'$\mathdefault{%s%g\times%s^{%d}}$' \
(12)                    % (sign_string, coeff, base, exponent)
(0)         class LogitFormatter(Formatter):
(4)             """
(4)             Probability formatter (using Math text).
(4)             """
(4)             def __init__(
(8)                 self,
(8)                 *,
(8)                 use_overline=False,
(8)                 one_half=r"\frac{1}{2}",
(8)                 minor=False,
(8)                 minor_threshold=25,
(8)                 minor_number=6,
(4)             ):
(8)                 r"""
(8)                 Parameters
(8)                 ----------
(8)                 use_overline : bool, default: False
(12)                    If x > 1/2, with x = 1-v, indicate if x should be displayed as
(12)                    $\overline{v}$. The default is to display $1-v$.
(8)                 one_half : str, default: r"\frac{1}{2}"
(12)                    The string used to represent 1/2.
(8)                 minor : bool, default: False
(12)                    Indicate if the formatter is formatting minor ticks or not.
(12)                    Basically minor ticks are not labelled, except when only few ticks
(12)                    are provided, ticks with most space with neighbor ticks are
(12)                    labelled. See other parameters to change the default behavior.
(8)                 minor_threshold : int, default: 25
(12)                    Maximum number of locs for labelling some minor ticks. This
(12)                    parameter have no effect if minor is False.
(8)                 minor_number : int, default: 6
(12)                    Number of ticks which are labelled when the number of ticks is
(12)                    below the threshold.
(8)                 """
(8)                 self._use_overline = use_overline
(8)                 self._one_half = one_half
(8)                 self._minor = minor
(8)                 self._labelled = set()
(8)                 self._minor_threshold = minor_threshold
(8)                 self._minor_number = minor_number
(4)             def use_overline(self, use_overline):
(8)                 r"""
(8)                 Switch display mode with overline for labelling p>1/2.
(8)                 Parameters
(8)                 ----------
(8)                 use_overline : bool, default: False
(12)                    If x > 1/2, with x = 1-v, indicate if x should be displayed as
(12)                    $\overline{v}$. The default is to display $1-v$.
(8)                 """
(8)                 self._use_overline = use_overline
(4)             def set_one_half(self, one_half):
(8)                 r"""
(8)                 Set the way one half is displayed.
(8)                 one_half : str, default: r"\frac{1}{2}"
(12)                    The string used to represent 1/2.
(8)                 """
(8)                 self._one_half = one_half
(4)             def set_minor_threshold(self, minor_threshold):
(8)                 """
(8)                 Set the threshold for labelling minors ticks.
(8)                 Parameters
(8)                 ----------
```

```
(8)                 minor_threshold : int
(12)                    Maximum number of locations for labelling some minor ticks. This
(12)                    parameter have no effect if minor is False.
(8)                 """
(8)                 self._minor_threshold = minor_threshold
(4)             def set_minor_number(self, minor_number):
(8)                 """
(8)                 Set the number of minor ticks to label when some minor ticks are
(8)                 labelled.
(8)                 Parameters
(8)                 ----------
(8)                 minor_number : int
(12)                    Number of ticks which are labelled when the number of ticks is
(12)                    below the threshold.
(8)                 """
(8)                 self._minor_number = minor_number
(4)             def set_locs(self, locs):
(8)                 self.locs = np.array(locs)
(8)                 self._labelled.clear()
(8)                 if not self._minor:
(12)                    return None
(8)                 if all(
(12)                    _is_decade(x, rtol=1e-7)
(12)                    or _is_decade(1 - x, rtol=1e-7)
(12)                    or (_is_close_to_int(2 * x) and
(16)                        int(np.round(2 * x)) == 1)
(12)                    for x in locs
(8)                 ):
(12)                    return None
(8)                 if len(locs) < self._minor_threshold:
(12)                    if len(locs) < self._minor_number:
(16)                        self._labelled.update(locs)
(12)                    else:
(16)                        diff = np.diff(-np.log(1 / self.locs - 1))
(16)                        space_pessimistic = np.minimum(
(20)                            np.concatenate(((np.inf,), diff)),
(20)                            np.concatenate((diff, (np.inf,))),
(16)                        )
(16)                        space_sum = (
(20)                            np.concatenate(((0,), diff))
(20)                            + np.concatenate((diff, (0,)))
(16)                        )
(16)                        good_minor = sorted(
(20)                            range(len(self.locs)),
(20)                            key=lambda i: (space_pessimistic[i], space_sum[i]),
(16)                        )[-self._minor_number:]
(16)                        self._labelled.update(locs[i] for i in good_minor)
(4)             def _format_value(self, x, locs, sci_notation=True):
(8)                 if sci_notation:
(12)                    exponent = math.floor(np.log10(x))
(12)                    min_precision = 0
(8)                 else:
(12)                    exponent = 0
(12)                    min_precision = 1
(8)                 value = x * 10 ** (-exponent)
(8)                 if len(locs) < 2:
(12)                    precision = min_precision
(8)                 else:
(12)                    diff = np.sort(np.abs(locs - x))[1]
(12)                    precision = -np.log10(diff) + exponent
(12)                    precision = (
(16)                        int(np.round(precision))
(16)                        if _is_close_to_int(precision)
(16)                        else math.ceil(precision)
(12)                    )
(12)                    if precision < min_precision:
(16)                        precision = min_precision
(8)                 mantissa = r"%.*f" % (precision, value)
(8)                 if not sci_notation:
```

```
(12)                        return mantissa
(8)                     s = r"%s\cdot10^{%d}" % (mantissa, exponent)
(8)                     return s
(4)                 def _one_minus(self, s):
(8)                     if self._use_overline:
(12)                        return r"\overline{%s}" % s
(8)                     else:
(12)                        return f"1-{s}"
(4)                 def __call__(self, x, pos=None):
(8)                     if self._minor and x not in self._labelled:
(12)                        return ""
(8)                     if x <= 0 or x >= 1:
(12)                        return ""
(8)                     if _is_close_to_int(2 * x) and round(2 * x) == 1:
(12)                        s = self._one_half
(8)                     elif x < 0.5 and _is_decade(x, rtol=1e-7):
(12)                        exponent = round(math.log10(x))
(12)                        s = "10^{%d}" % exponent
(8)                     elif x > 0.5 and _is_decade(1 - x, rtol=1e-7):
(12)                        exponent = round(math.log10(1 - x))
(12)                        s = self._one_minus("10^{%d}" % exponent)
(8)                     elif x < 0.1:
(12)                        s = self._format_value(x, self.locs)
(8)                     elif x > 0.9:
(12)                        s = self._one_minus(self._format_value(1-x, 1-self.locs))
(8)                     else:
(12)                        s = self._format_value(x, self.locs, sci_notation=False)
(8)                     return r"$\mathdefault{%s}$" % s
(4)                 def format_data_short(self, value):
(8)                     if value < 0.1:
(12)                        return f"{value:e}"
(8)                     if value < 0.9:
(12)                        return f"{value:f}"
(8)                     return f"1-{1 - value:e}"
(0)             class EngFormatter(Formatter):
(4)                 """
(4)                 Format axis values using engineering prefixes to represent powers
(4)                 of 1000, plus a specified unit, e.g., 10 MHz instead of 1e7.
(4)                 """
(4)                 ENG_PREFIXES = {
(8)                     -30: "q",
(8)                     -27: "r",
(8)                     -24: "y",
(8)                     -21: "z",
(8)                     -18: "a",
(8)                     -15: "f",
(8)                     -12: "p",
(9)                      -9: "n",
(9)                      -6: "\N{MICRO SIGN}",
(9)                      -3: "m",
(10)                      0: "",
(10)                      3: "k",
(10)                      6: "M",
(10)                      9: "G",
(9)                      12: "T",
(9)                      15: "P",
(9)                      18: "E",
(9)                      21: "Z",
(9)                      24: "Y",
(9)                      27: "R",
(9)                      30: "Q"
(4)                 }
(4)                 def __init__(self, unit="", places=None, sep=" ", *, usetex=None,
(17)                             useMathText=None):
(8)                     r"""
(8)                     Parameters
(8)                     ----------
(8)                     unit : str, default: ""
(12)                        Unit symbol to use, suitable for use with single-letter
```

```
(12)                representations of powers of 1000. For example, 'Hz' or 'm'.
(8)             places : int, default: None
(12)                Precision with which to display the number, specified in
(12)                digits after the decimal point (there will be between one
(12)                and three digits before the decimal point). If it is None,
(12)                the formatting falls back to the floating point format '%g',
(12)                which displays up to 6 *significant* digits, i.e. the equivalent
(12)                value for *places* varies between 0 and 5 (inclusive).
(8)             sep : str, default: " "
(12)                Separator used between the value and the prefix/unit. For
(12)                example, one get '3.14 mV' if ``sep`` is " " (default) and
(12)                '3.14mV' if ``sep`` is "". Besides the default behavior, some
(12)                other useful options may be:
(12)                * ``sep=""`` to append directly the prefix/unit to the value;
(12)                * ``sep="\N{THIN SPACE}"`` (``U+2009``);
(12)                * ``sep="\N{NARROW NO-BREAK SPACE}"`` (``U+202F``);
(12)                * ``sep="\N{NO-BREAK SPACE}"`` (``U+00A0``).
(8)             usetex : bool, default: :rc:`text.usetex`
(12)                To enable/disable the use of TeX's math mode for rendering the
(12)                numbers in the formatter.
(8)             useMathText : bool, default: :rc:`axes.formatter.use_mathtext`
(12)                To enable/disable the use mathtext for rendering the numbers in
(12)                the formatter.
(8)             """
(8)             self.unit = unit
(8)             self.places = places
(8)             self.sep = sep
(8)             self.set_usetex(usetex)
(8)             self.set_useMathText(useMathText)
(4)         def get_usetex(self):
(8)             return self._usetex
(4)         def set_usetex(self, val):
(8)             if val is None:
(12)                self._usetex = mpl.rcParams['text.usetex']
(8)             else:
(12)                self._usetex = val
(4)         usetex = property(fget=get_usetex, fset=set_usetex)
(4)         def get_useMathText(self):
(8)             return self._useMathText
(4)         def set_useMathText(self, val):
(8)             if val is None:
(12)                self._useMathText = mpl.rcParams['axes.formatter.use_mathtext']
(8)             else:
(12)                self._useMathText = val
(4)         useMathText = property(fget=get_useMathText, fset=set_useMathText)
(4)         def __call__(self, x, pos=None):
(8)             s = f"{self.format_eng(x)}{self.unit}"
(8)             if self.sep and s.endswith(self.sep):
(12)                s = s[:-len(self.sep)]
(8)             return self.fix_minus(s)
(4)         def format_eng(self, num):
(8)             """
(8)             Format a number in engineering notation, appending a letter
(8)             representing the power of 1000 of the original number.
(8)             Some examples:
(8)             >>> format_eng(0)        # for self.places = 0
(8)             '0'
(8)             >>> format_eng(1000000)  # for self.places = 1
(8)             '1.0 M'
(8)             >>> format_eng(-1e-6)  # for self.places = 2
(8)             '-1.00 \N{MICRO SIGN}'
(8)             """
(8)             sign = 1
(8)             fmt = "g" if self.places is None else f".{self.places:d}f"
(8)             if num < 0:
(12)                sign = -1
(12)                num = -num
(8)             if num != 0:
(12)                pow10 = int(math.floor(math.log10(num) / 3) * 3)
```

```
(8)                 else:
(12)                    pow10 = 0
(12)                    num = 0.0
(8)                 pow10 = np.clip(pow10, min(self.ENG_PREFIXES), max(self.ENG_PREFIXES))
(8)                 mant = sign * num / (10.0 ** pow10)
(8)                 if (abs(float(format(mant, fmt))) >= 1000
(16)                        and pow10 < max(self.ENG_PREFIXES)):
(12)                    mant /= 1000
(12)                    pow10 += 3
(8)                 prefix = self.ENG_PREFIXES[int(pow10)]
(8)                 if self._usetex or self._useMathText:
(12)                    formatted = f"${mant:{fmt}}${self.sep}{prefix}"
(8)                 else:
(12)                    formatted = f"{mant:{fmt}}{self.sep}{prefix}"
(8)                 return formatted
(0)         class PercentFormatter(Formatter):
(4)             """
(4)             Format numbers as a percentage.
(4)             Parameters
(4)             ----------
(4)             xmax : float
(8)                 Determines how the number is converted into a percentage.
(8)                 *xmax* is the data value that corresponds to 100%.
(8)                 Percentages are computed as ``x / xmax * 100``. So if the data is
(8)                 already scaled to be percentages, *xmax* will be 100. Another common
(8)                 situation is where *xmax* is 1.0.
(4)             decimals : None or int
(8)                 The number of decimal places to place after the point.
(8)                 If *None* (the default), the number will be computed automatically.
(4)             symbol : str or None
(8)                 A string that will be appended to the label. It may be
(8)                 *None* or empty to indicate that no symbol should be used. LaTeX
(8)                 special characters are escaped in *symbol* whenever latex mode is
(8)                 enabled, unless *is_latex* is *True*.
(4)             is_latex : bool
(8)                 If *False*, reserved LaTeX characters in *symbol* will be escaped.
(4)             """
(4)             def __init__(self, xmax=100, decimals=None, symbol='%', is_latex=False):
(8)                 self.xmax = xmax + 0.0
(8)                 self.decimals = decimals
(8)                 self._symbol = symbol
(8)                 self._is_latex = is_latex
(4)             def __call__(self, x, pos=None):
(8)                 """Format the tick as a percentage with the appropriate scaling."""
(8)                 ax_min, ax_max = self.axis.get_view_interval()
(8)                 display_range = abs(ax_max - ax_min)
(8)                 return self.fix_minus(self.format_pct(x, display_range))
(4)             def format_pct(self, x, display_range):
(8)                 """
(8)                 Format the number as a percentage number with the correct
(8)                 number of decimals and adds the percent symbol, if any.
(8)                 If ``self.decimals`` is `None`, the number of digits after the
(8)                 decimal point is set based on the *display_range* of the axis
(8)                 as follows:
(8)                 ============= ======== =======================
(8)                 display_range decimals sample
(8)                 ============= ======== =======================
(8)                 >50           0        ``x = 34.5`` => 35%
(8)                 >5            1        ``x = 34.5`` => 34.5%
(8)                 >0.5          2        ``x = 34.5`` => 34.50%
(8)                 ...           ...      ...
(8)                 ============= ======== =======================
(8)                 This method will not be very good for tiny axis ranges or
(8)                 extremely large ones. It assumes that the values on the chart
(8)                 are percentages displayed on a reasonable scale.
(8)                 """
(8)                 x = self.convert_to_pct(x)
(8)                 if self.decimals is None:
(12)                    scaled_range = self.convert_to_pct(display_range)
```

```
(12)                    if scaled_range <= 0:
(16)                        decimals = 0
(12)                    else:
(16)                        decimals = math.ceil(2.0 - math.log10(2.0 * scaled_range))
(16)                        if decimals > 5:
(20)                            decimals = 5
(16)                        elif decimals < 0:
(20)                            decimals = 0
(8)                 else:
(12)                    decimals = self.decimals
(8)                 s = f'{x:0.{int(decimals)}f}'
(8)                 return s + self.symbol
(4)             def convert_to_pct(self, x):
(8)                 return 100.0 * (x / self.xmax)
(4)             @property
(4)             def symbol(self):
(8)                 r"""
(8)                 The configured percent symbol as a string.
(8)                 If LaTeX is enabled via :rc:`text.usetex`, the special characters
(8)                 ``{'#', '$', '%', '&', '~', '_', '^', '\', '{', '}'}`` are
(8)                 automatically escaped in the string.
(8)                 """
(8)                 symbol = self._symbol
(8)                 if not symbol:
(12)                    symbol = ''
(8)                 elif not self._is_latex and mpl.rcParams['text.usetex']:
(12)                    for spec in r'\#$%&~_^{}':
(16)                        symbol = symbol.replace(spec, '\\' + spec)
(8)                 return symbol
(4)             @symbol.setter
(4)             def symbol(self, symbol):
(8)                 self._symbol = symbol
(0)         class Locator(TickHelper):
(4)             """
(4)             Determine tick locations.
(4)             Note that the same locator should not be used across multiple
(4)             `~matplotlib.axis.Axis` because the locator stores references to the Axis
(4)             data and view limits.
(4)             """
(4)             MAXTICKS = 1000
(4)             def tick_values(self, vmin, vmax):
(8)                 """
(8)                 Return the values of the located ticks given **vmin** and **vmax**.
(8)                 .. note::
(12)                    To get tick locations with the vmin and vmax values defined
(12)                    automatically for the associated ``axis`` simply call
(12)                    the Locator instance::
(16)                        >>> print(type(loc))
(16)                        <type 'Locator'>
(16)                        >>> print(loc())
(16)                        [1, 2, 3, 4]
(8)                 """
(8)                 raise NotImplementedError('Derived must override')
(4)             def set_params(self, **kwargs):
(8)                 """
(8)                 Do nothing, and raise a warning. Any locator class not supporting the
(8)                 set_params() function will call this.
(8)                 """
(8)                 _api.warn_external(
(12)                    "'set_params()' not defined for locator of type " +
(12)                    str(type(self)))
(4)             def __call__(self):
(8)                 """Return the locations of the ticks."""
(8)                 raise NotImplementedError('Derived must override')
(4)             def raise_if_exceeds(self, locs):
(8)                 """
(8)                 Log at WARNING level if *locs* is longer than `Locator.MAXTICKS`.
(8)                 This is intended to be called immediately before returning *locs* from
(8)                 ``__call__`` to inform users in case their Locator returns a huge
```

```
1286: (8)                    number of ticks, causing Matplotlib to run out of memory.
1287: (8)                    The "strange" name of this method dates back to when it would raise an
1288: (8)                    exception instead of emitting a log.
1289: (8)                    """
1290: (8)                    if len(locs) >= self.MAXTICKS:
1291: (12)                       _log.warning(
1292: (16)                           "Locator attempting to generate %s ticks ([%s, ..., %s]), "
1293: (16)                           "which exceeds Locator.MAXTICKS (%s).",
1294: (16)                           len(locs), locs[0], locs[-1], self.MAXTICKS)
1295: (8)                    return locs
1296: (4)                def nonsingular(self, v0, v1):
1297: (8)                    """
1298: (8)                    Adjust a range as needed to avoid singularities.
1299: (8)                    This method gets called during autoscaling, with ``(v0, v1)`` set to
1300: (8)                    the data limits on the Axes if the Axes contains any data, or
1301: (8)                    ``(-inf, +inf)`` if not.
1302: (8)                    - If ``v0 == v1`` (possibly up to some floating point slop), this
1303: (10)                     method returns an expanded interval around this value.
1304: (8)                    - If ``(v0, v1) == (-inf, +inf)``, this method returns appropriate
1305: (10)                     default view limits.
1306: (8)                    - Otherwise, ``(v0, v1)`` is returned without modification.
1307: (8)                    """
1308: (8)                    return mtransforms.nonsingular(v0, v1, expander=.05)
1309: (4)                def view_limits(self, vmin, vmax):
1310: (8)                    """
1311: (8)                    Select a scale for the range from vmin to vmax.
1312: (8)                    Subclasses should override this method to change locator behaviour.
1313: (8)                    """
1314: (8)                    return mtransforms.nonsingular(vmin, vmax)
1315: (0)            class IndexLocator(Locator):
1316: (4)                """
1317: (4)                Place ticks at every nth point plotted.
1318: (4)                IndexLocator assumes index plotting; i.e., that the ticks are placed at
integer
1319: (4)                values in the range between 0 and len(data) inclusive.
1320: (4)                """
1321: (4)                def __init__(self, base, offset):
1322: (8)                    """Place ticks every *base* data point, starting at *offset*."""
1323: (8)                    self._base = base
1324: (8)                    self.offset = offset
1325: (4)                def set_params(self, base=None, offset=None):
1326: (8)                    """Set parameters within this locator"""
1327: (8)                    if base is not None:
1328: (12)                       self._base = base
1329: (8)                    if offset is not None:
1330: (12)                       self.offset = offset
1331: (4)                def __call__(self):
1332: (8)                    """Return the locations of the ticks"""
1333: (8)                    dmin, dmax = self.axis.get_data_interval()
1334: (8)                    return self.tick_values(dmin, dmax)
1335: (4)                def tick_values(self, vmin, vmax):
1336: (8)                    return self.raise_if_exceeds(
1337: (12)                       np.arange(vmin + self.offset, vmax + 1, self._base))
1338: (0)            class FixedLocator(Locator):
1339: (4)                """
1340: (4)                Place ticks at a set of fixed values.
1341: (4)                If *nbins* is None ticks are placed at all values. Otherwise, the *locs*
array of
1342: (4)                possible positions will be subsampled to keep the number of ticks <=
1343: (4)                :math:`nbins* +1`. The subsampling will be done to include the smallest
absolute
1344: (4)                value; for example, if zero is included in the array of possibilities,
then it of
1345: (4)                the chosen ticks.
1346: (4)                """
1347: (4)                def __init__(self, locs, nbins=None):
1348: (8)                    self.locs = np.asarray(locs)
1349: (8)                    _api.check_shape((None,), locs=self.locs)
1350: (8)                    self.nbins = max(nbins, 2) if nbins is not None else None
```

```
(4)             def set_params(self, nbins=None):
(8)                 """Set parameters within this locator."""
(8)                 if nbins is not None:
(12)                    self.nbins = nbins
(4)             def __call__(self):
(8)                 return self.tick_values(None, None)
(4)             def tick_values(self, vmin, vmax):
(8)                 """
(8)                 Return the locations of the ticks.
(8)                 .. note::
(12)                    Because the values are fixed, vmin and vmax are not used in this
(12)                    method.
(8)                 """
(8)                 if self.nbins is None:
(12)                    return self.locs
(8)                 step = max(int(np.ceil(len(self.locs) / self.nbins)), 1)
(8)                 ticks = self.locs[::step]
(8)                 for i in range(1, step):
(12)                    ticks1 = self.locs[i::step]
(12)                    if np.abs(ticks1).min() < np.abs(ticks).min():
(16)                        ticks = ticks1
(8)                 return self.raise_if_exceeds(ticks)
(0)         class NullLocator(Locator):
(4)             """
(4)             No ticks
(4)             """
(4)             def __call__(self):
(8)                 return self.tick_values(None, None)
(4)             def tick_values(self, vmin, vmax):
(8)                 """
(8)                 Return the locations of the ticks.
(8)                 .. note::
(12)                    Because the values are Null, vmin and vmax are not used in this
(12)                    method.
(8)                 """
(8)                 return []
(0)         class LinearLocator(Locator):
(4)             """
(4)             Place ticks at evenly spaced values.
(4)             The first time this function is called it will try to set the
(4)             number of ticks to make a nice tick partitioning.  Thereafter, the
(4)             number of ticks will be fixed so that interactive navigation will
(4)             be nice
(4)             """
(4)             def __init__(self, numticks=None, presets=None):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 numticks : int or None, default None
(12)                    Number of ticks. If None, *numticks* = 11.
(8)                 presets : dict or None, default: None
(12)                    Dictionary mapping ``(vmin, vmax)`` to an array of locations.
(12)                    Overrides *numticks* if there is an entry for the current
(12)                    ``(vmin, vmax)``.
(8)                 """
(8)                 self.numticks = numticks
(8)                 if presets is None:
(12)                    self.presets = {}
(8)                 else:
(12)                    self.presets = presets
(4)             @property
(4)             def numticks(self):
(8)                 return self._numticks if self._numticks is not None else 11
(4)             @numticks.setter
(4)             def numticks(self, numticks):
(8)                 self._numticks = numticks
(4)             def set_params(self, numticks=None, presets=None):
(8)                 """Set parameters within this locator."""
(8)                 if presets is not None:
```

```
(12)                        self.presets = presets
(8)                     if numticks is not None:
(12)                        self.numticks = numticks
(4)                 def __call__(self):
(8)                     """Return the locations of the ticks."""
(8)                     vmin, vmax = self.axis.get_view_interval()
(8)                     return self.tick_values(vmin, vmax)
(4)                 def tick_values(self, vmin, vmax):
(8)                     vmin, vmax = mtransforms.nonsingular(vmin, vmax, expander=0.05)
(8)                     if (vmin, vmax) in self.presets:
(12)                        return self.presets[(vmin, vmax)]
(8)                     if self.numticks == 0:
(12)                        return []
(8)                     ticklocs = np.linspace(vmin, vmax, self.numticks)
(8)                     return self.raise_if_exceeds(ticklocs)
(4)                 def view_limits(self, vmin, vmax):
(8)                     """Try to choose the view limits intelligently."""
(8)                     if vmax < vmin:
(12)                        vmin, vmax = vmax, vmin
(8)                     if vmin == vmax:
(12)                        vmin -= 1
(12)                        vmax += 1
(8)                     if mpl.rcParams['axes.autolimit_mode'] == 'round_numbers':
(12)                        exponent, remainder = divmod(
(16)                            math.log10(vmax - vmin), math.log10(max(self.numticks - 1,
1)))
(12)                        exponent -= (remainder < .5)
(12)                        scale = max(self.numticks - 1, 1) ** (-exponent)
(12)                        vmin = math.floor(scale * vmin) / scale
(12)                        vmax = math.ceil(scale * vmax) / scale
(8)                     return mtransforms.nonsingular(vmin, vmax)
(0)             class MultipleLocator(Locator):
(4)                 """
(4)                 Place ticks at every integer multiple of a base plus an offset.
(4)                 """
(4)                 def __init__(self, base=1.0, offset=0.0):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     base : float > 0
(12)                        Interval between ticks.
(8)                     offset : float
(12)                        Value added to each multiple of *base*.
(12)                        .. versionadded:: 3.8
(8)                     """
(8)                     self._edge = _Edge_integer(base, 0)
(8)                     self._offset = offset
(4)                 def set_params(self, base=None, offset=None):
(8)                     """
(8)                     Set parameters within this locator.
(8)                     Parameters
(8)                     ----------
(8)                     base : float > 0
(12)                        Interval between ticks.
(8)                     offset : float
(12)                        Value added to each multiple of *base*.
(12)                        .. versionadded:: 3.8
(8)                     """
(8)                     if base is not None:
(12)                        self._edge = _Edge_integer(base, 0)
(8)                     if offset is not None:
(12)                        self._offset = offset
(4)                 def __call__(self):
(8)                     """Return the locations of the ticks."""
(8)                     vmin, vmax = self.axis.get_view_interval()
(8)                     return self.tick_values(vmin, vmax)
(4)                 def tick_values(self, vmin, vmax):
(8)                     if vmax < vmin:
(12)                        vmin, vmax = vmax, vmin
```

```
(8)                 step = self._edge.step
(8)                 vmin -= self._offset
(8)                 vmax -= self._offset
(8)                 vmin = self._edge.ge(vmin) * step
(8)                 n = (vmax - vmin + 0.001 * step) // step
(8)                 locs = vmin - step + np.arange(n + 3) * step + self._offset
(8)                 return self.raise_if_exceeds(locs)
(4)             def view_limits(self, dmin, dmax):
(8)                 """
(8)                 Set the view limits to the nearest tick values that contain the data.
(8)                 """
(8)                 if mpl.rcParams['axes.autolimit_mode'] == 'round_numbers':
(12)                    vmin = self._edge.le(dmin - self._offset) * self._edge.step +
self._offset
(12)                    vmax = self._edge.ge(dmax - self._offset) * self._edge.step +
self._offset
(12)                    if vmin == vmax:
(16)                        vmin -= 1
(16)                        vmax += 1
(8)                 else:
(12)                    vmin = dmin
(12)                    vmax = dmax
(8)                 return mtransforms.nonsingular(vmin, vmax)
(0)         def scale_range(vmin, vmax, n=1, threshold=100):
(4)             dv = abs(vmax - vmin)  # > 0 as nonsingular is called before.
(4)             meanv = (vmax + vmin) / 2
(4)             if abs(meanv) / dv < threshold:
(8)                 offset = 0
(4)             else:
(8)                 offset = math.copysign(10 ** (math.log10(abs(meanv)) // 1), meanv)
(4)             scale = 10 ** (math.log10(dv / n) // 1)
(4)             return scale, offset
(0)         class _Edge_integer:
(4)             """
(4)             Helper for `.MaxNLocator`, `.MultipleLocator`, etc.
(4)             Take floating-point precision limitations into account when calculating
(4)             tick locations as integer multiples of a step.
(4)             """
(4)             def __init__(self, step, offset):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 step : float > 0
(12)                    Interval between ticks.
(8)                 offset : float
(12)                    Offset subtracted from the data limits prior to calculating tick
(12)                    locations.
(8)                 """
(8)                 if step <= 0:
(12)                    raise ValueError("'step' must be positive")
(8)                 self.step = step
(8)                 self._offset = abs(offset)
(4)             def closeto(self, ms, edge):
(8)                 if self._offset > 0:
(12)                    digits = np.log10(self._offset / self.step)
(12)                    tol = max(1e-10, 10 ** (digits - 12))
(12)                    tol = min(0.4999, tol)
(8)                 else:
(12)                    tol = 1e-10
(8)                 return abs(ms - edge) < tol
(4)             def le(self, x):
(8)                 """Return the largest n: n*step <= x."""
(8)                 d, m = divmod(x, self.step)
(8)                 if self.closeto(m / self.step, 1):
(12)                    return d + 1
(8)                 return d
(4)             def ge(self, x):
(8)                 """Return the smallest n: n*step >= x."""
(8)                 d, m = divmod(x, self.step)
```

```
1555: (8)                   if self.closeto(m / self.step, 0):
1556: (12)                      return d
1557: (8)                   return d + 1
1558: (0)           class MaxNLocator(Locator):
1559: (4)               """
1560: (4)               Place evenly spaced ticks, with a cap on the total number of ticks.
1561: (4)               Finds nice tick locations with no more than :math:`nbins + 1` ticks being
within the
1562: (4)               view limits. Locations beyond the limits are added to support autoscaling.
1563: (4)               """
1564: (4)               default_params = dict(nbins=10,
1565: (26)                                     steps=None,
1566: (26)                                     integer=False,
1567: (26)                                     symmetric=False,
1568: (26)                                     prune=None,
1569: (26)                                     min_n_ticks=2)
1570: (4)               def __init__(self, nbins=None, **kwargs):
1571: (8)                   """
1572: (8)                   Parameters
1573: (8)                   ----------
1574: (8)                   nbins : int or 'auto', default: 10
1575: (12)                      Maximum number of intervals; one less than max number of
1576: (12)                      ticks.  If the string 'auto', the number of bins will be
1577: (12)                      automatically determined based on the length of the axis.
1578: (8)                   steps : array-like, optional
1579: (12)                      Sequence of acceptable tick multiples, starting with 1 and
1580: (12)                      ending with 10. For example, if ``steps=[1, 2, 4, 5, 10]``,
1581: (12)                      ``20, 40, 60`` or ``0.4, 0.6, 0.8`` would be possible
1582: (12)                      sets of ticks because they are multiples of 2.
1583: (12)                      ``30, 60, 90`` would not be generated because 3 does not
1584: (12)                      appear in this example list of steps.
1585: (8)                   integer : bool, default: False
1586: (12)                      If True, ticks will take only integer values, provided at least
1587: (12)                      *min_n_ticks* integers are found within the view limits.
1588: (8)                   symmetric : bool, default: False
1589: (12)                      If True, autoscaling will result in a range symmetric about zero.
1590: (8)                   prune : {'lower', 'upper', 'both', None}, default: None
1591: (12)                      Remove the 'lower' tick, the 'upper' tick, or ticks on 'both'
sides
1592: (12)                      *if they fall exactly on an axis' edge* (this typically occurs
when
1593: (12)                      :rc:`axes.autolimit_mode` is 'round_numbers').  Removing such
ticks
1594: (12)                      is mostly useful for stacked or ganged plots, where the upper tick
1595: (12)                      of an Axes overlaps with the lower tick of the axes above it.
1596: (8)                   min_n_ticks : int, default: 2
1597: (12)                      Relax *nbins* and *integer* constraints if necessary to obtain
1598: (12)                      this minimum number of ticks.
1599: (8)                   """
1600: (8)                   if nbins is not None:
1601: (12)                      kwargs['nbins'] = nbins
1602: (8)                   self.set_params(**{**self.default_params, **kwargs})
1603: (4)               @staticmethod
1604: (4)               def _validate_steps(steps):
1605: (8)                   if not np.iterable(steps):
1606: (12)                      raise ValueError('steps argument must be an increasing sequence '
1607: (29)                                       'of numbers between 1 and 10 inclusive')
1608: (8)                   steps = np.asarray(steps)
1609: (8)                   if np.any(np.diff(steps) <= 0) or steps[-1] > 10 or steps[0] < 1:
1610: (12)                      raise ValueError('steps argument must be an increasing sequence '
1611: (29)                                       'of numbers between 1 and 10 inclusive')
1612: (8)                   if steps[0] != 1:
1613: (12)                      steps = np.concatenate([[1], steps])
1614: (8)                   if steps[-1] != 10:
1615: (12)                      steps = np.concatenate([steps, [10]])
1616: (8)                   return steps
1617: (4)               @staticmethod
1618: (4)               def _staircase(steps):
1619: (8)                   return np.concatenate([0.1 * steps[:-1], steps, [10 * steps[1]]])
```

```
(4)     def set_params(self, **kwargs):
(8)         """
(8)         Set parameters for this locator.
(8)         Parameters
(8)         ----------
(8)         nbins : int or 'auto', optional
(12)            see `.MaxNLocator`
(8)         steps : array-like, optional
(12)            see `.MaxNLocator`
(8)         integer : bool, optional
(12)            see `.MaxNLocator`
(8)         symmetric : bool, optional
(12)            see `.MaxNLocator`
(8)         prune : {'lower', 'upper', 'both', None}, optional
(12)            see `.MaxNLocator`
(8)         min_n_ticks : int, optional
(12)            see `.MaxNLocator`
(8)         """
(8)         if 'nbins' in kwargs:
(12)            self._nbins = kwargs.pop('nbins')
(12)            if self._nbins != 'auto':
(16)                self._nbins = int(self._nbins)
(8)         if 'symmetric' in kwargs:
(12)            self._symmetric = kwargs.pop('symmetric')
(8)         if 'prune' in kwargs:
(12)            prune = kwargs.pop('prune')
(12)            _api.check_in_list(['upper', 'lower', 'both', None], prune=prune)
(12)            self._prune = prune
(8)         if 'min_n_ticks' in kwargs:
(12)            self._min_n_ticks = max(1, kwargs.pop('min_n_ticks'))
(8)         if 'steps' in kwargs:
(12)            steps = kwargs.pop('steps')
(12)            if steps is None:
(16)                self._steps = np.array([1, 1.5, 2, 2.5, 3, 4, 5, 6, 8, 10])
(12)            else:
(16)                self._steps = self._validate_steps(steps)
(12)            self._extended_steps = self._staircase(self._steps)
(8)         if 'integer' in kwargs:
(12)            self._integer = kwargs.pop('integer')
(8)         if kwargs:
(12)            raise _api.kwarg_error("set_params", kwargs)
(4)     def _raw_ticks(self, vmin, vmax):
(8)         """
(8)         Generate a list of tick locations including the range *vmin* to
(8)         *vmax*.  In some applications, one or both of the end locations
(8)         will not be needed, in which case they are trimmed off
(8)         elsewhere.
(8)         """
(8)         if self._nbins == 'auto':
(12)            if self.axis is not None:
(16)                nbins = np.clip(self.axis.get_tick_space(),
(32)                                max(1, self._min_n_ticks - 1), 9)
(12)            else:
(16)                nbins = 9
(8)         else:
(12)            nbins = self._nbins
(8)         scale, offset = scale_range(vmin, vmax, nbins)
(8)         _vmin = vmin - offset
(8)         _vmax = vmax - offset
(8)         steps = self._extended_steps * scale
(8)         if self._integer:
(12)            igood = (steps < 1) | (np.abs(steps - np.round(steps)) < 0.001)
(12)            steps = steps[igood]
(8)         raw_step = ((_vmax - _vmin) / nbins)
(8)         if hasattr(self.axis, "axes") and self.axis.axes.name == '3d':
(12)            raw_step = raw_step * 23/24
(8)         large_steps = steps >= raw_step
(8)         if mpl.rcParams['axes.autolimit_mode'] == 'round_numbers':
(12)            floored_vmins = (_vmin // steps) * steps
```

```
1689: (12)                      floored_vmaxs = floored_vmins + steps * nbins
1690: (12)                      large_steps = large_steps & (floored_vmaxs >= _vmax)
1691: (8)                   if any(large_steps):
1692: (12)                      istep = np.nonzero(large_steps)[0][0]
1693: (8)                   else:
1694: (12)                      istep = len(steps) - 1
1695: (8)                   for step in steps[:istep+1][::-1]:
1696: (12)                      if (self._integer and
1697: (20)                              np.floor(_vmax) - np.ceil(_vmin) >= self._min_n_ticks -
1):
1698: (16)                          step = max(1, step)
1699: (12)                      best_vmin = (_vmin // step) * step
1700: (12)                      edge = _Edge_integer(step, offset)
1701: (12)                      low = edge.le(_vmin - best_vmin)
1702: (12)                      high = edge.ge(_vmax - best_vmin)
1703: (12)                      ticks = np.arange(low, high + 1) * step + best_vmin
1704: (12)                      nticks = ((ticks <= _vmax) & (ticks >= _vmin)).sum()
1705: (12)                      if nticks >= self._min_n_ticks:
1706: (16)                          break
1707: (8)                   return ticks + offset
1708: (4)               def __call__(self):
1709: (8)                   vmin, vmax = self.axis.get_view_interval()
1710: (8)                   return self.tick_values(vmin, vmax)
1711: (4)               def tick_values(self, vmin, vmax):
1712: (8)                   if self._symmetric:
1713: (12)                      vmax = max(abs(vmin), abs(vmax))
1714: (12)                      vmin = -vmax
1715: (8)                   vmin, vmax = mtransforms.nonsingular(
1716: (12)                      vmin, vmax, expander=1e-13, tiny=1e-14)
1717: (8)                   locs = self._raw_ticks(vmin, vmax)
1718: (8)                   prune = self._prune
1719: (8)                   if prune == 'lower':
1720: (12)                      locs = locs[1:]
1721: (8)                   elif prune == 'upper':
1722: (12)                      locs = locs[:-1]
1723: (8)                   elif prune == 'both':
1724: (12)                      locs = locs[1:-1]
1725: (8)                   return self.raise_if_exceeds(locs)
1726: (4)               def view_limits(self, dmin, dmax):
1727: (8)                   if self._symmetric:
1728: (12)                      dmax = max(abs(dmin), abs(dmax))
1729: (12)                      dmin = -dmax
1730: (8)                   dmin, dmax = mtransforms.nonsingular(
1731: (12)                      dmin, dmax, expander=1e-12, tiny=1e-13)
1732: (8)                   if mpl.rcParams['axes.autolimit_mode'] == 'round_numbers':
1733: (12)                      return self._raw_ticks(dmin, dmax)[[0, -1]]
1734: (8)                   else:
1735: (12)                      return dmin, dmax
1736: (0)           def _is_decade(x, *, base=10, rtol=None):
1737: (4)               """Return True if *x* is an integer power of *base*."""
1738: (4)               if not np.isfinite(x):
1739: (8)                   return False
1740: (4)               if x == 0.0:
1741: (8)                   return True
1742: (4)               lx = np.log(abs(x)) / np.log(base)
1743: (4)               if rtol is None:
1744: (8)                   return np.isclose(lx, np.round(lx))
1745: (4)               else:
1746: (8)                   return np.isclose(lx, np.round(lx), rtol=rtol)
1747: (0)           def _decade_less_equal(x, base):
1748: (4)               """
1749: (4)               Return the largest integer power of *base* that's less or equal to *x*.
1750: (4)               If *x* is negative, the exponent will be *greater*.
1751: (4)               """
1752: (4)               return (x if x == 0 else
1753: (12)                      -_decade_greater_equal(-x, base) if x < 0 else
1754: (12)                      base ** np.floor(np.log(x) / np.log(base)))
1755: (0)           def _decade_greater_equal(x, base):
1756: (4)               """
```

```
1757: (4)               Return the smallest integer power of *base* that's greater or equal to
*x*.
1758: (4)               If *x* is negative, the exponent will be *smaller*.
1759: (4)               """
1760: (4)               return (x if x == 0 else
1761: (12)                      -_decade_less_equal(-x, base) if x < 0 else
1762: (12)                      base ** np.ceil(np.log(x) / np.log(base)))
1763: (0)           def _decade_less(x, base):
1764: (4)               """
1765: (4)               Return the largest integer power of *base* that's less than *x*.
1766: (4)               If *x* is negative, the exponent will be *greater*.
1767: (4)               """
1768: (4)               if x < 0:
1769: (8)                   return -_decade_greater(-x, base)
1770: (4)               less = _decade_less_equal(x, base)
1771: (4)               if less == x:
1772: (8)                   less /= base
1773: (4)               return less
1774: (0)           def _decade_greater(x, base):
1775: (4)               """
1776: (4)               Return the smallest integer power of *base* that's greater than *x*.
1777: (4)               If *x* is negative, the exponent will be *smaller*.
1778: (4)               """
1779: (4)               if x < 0:
1780: (8)                   return -_decade_less(-x, base)
1781: (4)               greater = _decade_greater_equal(x, base)
1782: (4)               if greater == x:
1783: (8)                   greater *= base
1784: (4)               return greater
1785: (0)           def _is_close_to_int(x):
1786: (4)               return math.isclose(x, round(x))
1787: (0)           class LogLocator(Locator):
1788: (4)               """
1789: (4)               Place logarithmically spaced ticks.
1790: (4)               Places ticks at the values ``subs[j] * base**i``.
1791: (4)               """
1792: (4)               @_api.delete_parameter("3.8", "numdecs")
1793: (4)               def __init__(self, base=10.0, subs=(1.0,), numdecs=4, numticks=None):
1794: (8)                   """
1795: (8)                   Parameters
1796: (8)                   ----------
1797: (8)                   base : float, default: 10.0
1798: (12)                      The base of the log used, so major ticks are placed at
``base**n``, where
1799: (12)                      ``n`` is an integer.
1800: (8)                   subs : None or {'auto', 'all'} or sequence of float, default: (1.0,)
1801: (12)                      Gives the multiples of integer powers of the base at which to
place ticks.
1802: (12)                      The default of ``(1.0, )`` places ticks only at integer powers of
the base.
1803: (12)                      Permitted string values are ``'auto'`` and ``'all'``. Both of
these use an
1804: (12)                      algorithm based on the axis view limits to determine whether and
how to put
1805: (12)                      ticks between integer powers of the base:
1806: (12)                      - ``'auto'``: Ticks are placed only between integer powers.
1807: (12)                      - ``'all'``: Ticks are placed between *and* at integer powers.
1808: (12)                      - ``None``: Equivalent to ``'auto'``.
1809: (8)                   numticks : None or int, default: None
1810: (12)                      The maximum number of ticks to allow on a given axis. The default
of
1811: (12)                      ``None`` will try to choose intelligently as long as this Locator
has
1812: (12)                      already been assigned to an axis using
`~.axis.Axis.get_tick_space`, but
1813: (12)                      otherwise falls back to 9.
1814: (8)                   """
1815: (8)                   if numticks is None:
1816: (12)                      if mpl.rcParams['_internal.classic_mode']:
```

```
1817: (16)                            numticks = 15
1818: (12)                        else:
1819: (16)                            numticks = 'auto'
1820: (8)                    self._base = float(base)
1821: (8)                    self._set_subs(subs)
1822: (8)                    self._numdecs = numdecs
1823: (8)                    self.numticks = numticks
1824: (4)                @_api.delete_parameter("3.8", "numdecs")
1825: (4)                def set_params(self, base=None, subs=None, numdecs=None, numticks=None):
1826: (8)                    """Set parameters within this locator."""
1827: (8)                    if base is not None:
1828: (12)                       self._base = float(base)
1829: (8)                    if subs is not None:
1830: (12)                       self._set_subs(subs)
1831: (8)                    if numdecs is not None:
1832: (12)                       self._numdecs = numdecs
1833: (8)                    if numticks is not None:
1834: (12)                       self.numticks = numticks
1835: (4)                numdecs = _api.deprecate_privatize_attribute(
1836: (8)                    "3.8", addendum="This attribute has no effect.")
1837: (4)                def _set_subs(self, subs):
1838: (8)                    """
1839: (8)                    Set the minor ticks for the log scaling every ``base**i*subs[j]``.
1840: (8)                    """
1841: (8)                    if subs is None:  # consistency with previous bad API
1842: (12)                       self._subs = 'auto'
1843: (8)                    elif isinstance(subs, str):
1844: (12)                       _api.check_in_list(('all', 'auto'), subs=subs)
1845: (12)                       self._subs = subs
1846: (8)                    else:
1847: (12)                       try:
1848: (16)                           self._subs = np.asarray(subs, dtype=float)
1849: (12)                       except ValueError as e:
1850: (16)                           raise ValueError("subs must be None, 'all', 'auto' or "
1851: (33)                                            "a sequence of floats, not "
1852: (33)                                            f"{subs}.") from e
1853: (12)                       if self._subs.ndim != 1:
1854: (16)                           raise ValueError("A sequence passed to subs must be "
1855: (33)                                            "1-dimensional, not "
1856: (33)                                            f"{self._subs.ndim}-dimensional.")
1857: (4)                def __call__(self):
1858: (8)                    """Return the locations of the ticks."""
1859: (8)                    vmin, vmax = self.axis.get_view_interval()
1860: (8)                    return self.tick_values(vmin, vmax)
1861: (4)                def tick_values(self, vmin, vmax):
1862: (8)                    if self.numticks == 'auto':
1863: (12)                       if self.axis is not None:
1864: (16)                           numticks = np.clip(self.axis.get_tick_space(), 2, 9)
1865: (12)                       else:
1866: (16)                           numticks = 9
1867: (8)                    else:
1868: (12)                       numticks = self.numticks
1869: (8)                    b = self._base
1870: (8)                    if vmin <= 0.0:
1871: (12)                       if self.axis is not None:
1872: (16)                           vmin = self.axis.get_minpos()
1873: (12)                       if vmin <= 0.0 or not np.isfinite(vmin):
1874: (16)                           raise ValueError(
1875: (20)                               "Data has no positive values, and therefore cannot be log-
scaled.")
1876: (8)                    _log.debug('vmin %s vmax %s', vmin, vmax)
1877: (8)                    if vmax < vmin:
1878: (12)                       vmin, vmax = vmax, vmin
1879: (8)                    log_vmin = math.log(vmin) / math.log(b)
1880: (8)                    log_vmax = math.log(vmax) / math.log(b)
1881: (8)                    numdec = math.floor(log_vmax) - math.ceil(log_vmin)
1882: (8)                    if isinstance(self._subs, str):
1883: (12)                       if numdec > 10 or b < 3:
1884: (16)                           if self._subs == 'auto':
```

```
1885: (20)                                return np.array([])  # no minor or major ticks
1886: (16)                            else:
1887: (20)                                subs = np.array([1.0])  # major ticks
1888: (12)                        else:
1889: (16)                            _first = 2.0 if self._subs == 'auto' else 1.0
1890: (16)                            subs = np.arange(_first, b)
1891: (8)                    else:
1892: (12)                        subs = self._subs
1893: (8)                    stride = (max(math.ceil(numdec / (numticks - 1)), 1)
1894: (18)                             if mpl.rcParams['_internal.classic_mode'] else
1895: (18)                             numdec // numticks + 1)
1896: (8)                    if stride >= numdec:
1897: (12)                        stride = max(1, numdec - 1)
1898: (8)                    have_subs = len(subs) > 1 or (len(subs) == 1 and subs[0] != 1.0)
1899: (8)                    decades = np.arange(math.floor(log_vmin) - stride,
1900: (28)                                        math.ceil(log_vmax) + 2 * stride, stride)
1901: (8)                    if have_subs:
1902: (12)                        if stride == 1:
1903: (16)                            ticklocs = np.concatenate(
1904: (20)                                [subs * decade_start for decade_start in b ** decades])
1905: (12)                        else:
1906: (16)                            ticklocs = np.array([])
1907: (8)                    else:
1908: (12)                        ticklocs = b ** decades
1909: (8)                    _log.debug('ticklocs %r', ticklocs)
1910: (8)                    if (len(subs) > 1
1911: (16)                            and stride == 1
1912: (16)                            and ((vmin <= ticklocs) & (ticklocs <= vmax)).sum() <= 1):
1913: (12)                        return AutoLocator().tick_values(vmin, vmax)
1914: (8)                    else:
1915: (12)                        return self.raise_if_exceeds(ticklocs)
1916: (4)                def view_limits(self, vmin, vmax):
1917: (8)                    """Try to choose the view limits intelligently."""
1918: (8)                    b = self._base
1919: (8)                    vmin, vmax = self.nonsingular(vmin, vmax)
1920: (8)                    if mpl.rcParams['axes.autolimit_mode'] == 'round_numbers':
1921: (12)                        vmin = _decade_less_equal(vmin, b)
1922: (12)                        vmax = _decade_greater_equal(vmax, b)
1923: (8)                    return vmin, vmax
1924: (4)                def nonsingular(self, vmin, vmax):
1925: (8)                    if vmin > vmax:
1926: (12)                        vmin, vmax = vmax, vmin
1927: (8)                    if not np.isfinite(vmin) or not np.isfinite(vmax):
1928: (12)                        vmin, vmax = 1, 10  # Initial range, no data plotted yet.
1929: (8)                    elif vmax <= 0:
1930: (12)                        _api.warn_external(
1931: (16)                            "Data has no positive values, and therefore cannot be "
1932: (16)                            "log-scaled.")
1933: (12)                        vmin, vmax = 1, 10
1934: (8)                    else:
1935: (12)                        minpos = min(axis.get_minpos() for axis in
self.axis._get_shared_axis())
1936: (12)                        if not np.isfinite(minpos):
1937: (16)                            minpos = 1e-300  # This should never take effect.
1938: (12)                        if vmin <= 0:
1939: (16)                            vmin = minpos
1940: (12)                        if vmin == vmax:
1941: (16)                            vmin = _decade_less(vmin, self._base)
1942: (16)                            vmax = _decade_greater(vmax, self._base)
1943: (8)                    return vmin, vmax
1944: (0)            class SymmetricalLogLocator(Locator):
1945: (4)                """
1946: (4)                Place ticks spaced linearly near zero and spaced logarithmically beyond a
threshold.
1947: (4)                """
1948: (4)                def __init__(self, transform=None, subs=None, linthresh=None, base=None):
1949: (8)                    """
1950: (8)                    Parameters
1951: (8)                    ----------
```

```
(8)                 transform : `~.scale.SymmetricalLogTransform`, optional
(12)                    If set, defines the *base* and *linthresh* of the symlog
transform.
(8)                 base, linthresh : float, optional
(12)                    The *base* and *linthresh* of the symlog transform, as documented
(12)                    for `.SymmetricalLogScale`.  These parameters are only used if
(12)                    *transform* is not set.
(8)                 subs : sequence of float, default: [1]
(12)                    The multiples of integer powers of the base where ticks are
placed,
(12)                    i.e., ticks are placed at
(12)                    ``[sub * base**i for i in ... for sub in subs]``.
(8)                 Notes
(8)                 -----
(8)                 Either *transform*, or both *base* and *linthresh*, must be given.
(8)                 """
(8)                 if transform is not None:
(12)                    self._base = transform.base
(12)                    self._linthresh = transform.linthresh
(8)                 elif linthresh is not None and base is not None:
(12)                    self._base = base
(12)                    self._linthresh = linthresh
(8)                 else:
(12)                    raise ValueError("Either transform, or both linthresh "
(29)                                     "and base, must be provided.")
(8)                 if subs is None:
(12)                    self._subs = [1.0]
(8)                 else:
(12)                    self._subs = subs
(8)                 self.numticks = 15
(4)             def set_params(self, subs=None, numticks=None):
(8)                 """Set parameters within this locator."""
(8)                 if numticks is not None:
(12)                    self.numticks = numticks
(8)                 if subs is not None:
(12)                    self._subs = subs
(4)             def __call__(self):
(8)                 """Return the locations of the ticks."""
(8)                 vmin, vmax = self.axis.get_view_interval()
(8)                 return self.tick_values(vmin, vmax)
(4)             def tick_values(self, vmin, vmax):
(8)                 linthresh = self._linthresh
(8)                 if vmax < vmin:
(12)                    vmin, vmax = vmax, vmin
(8)                 if -linthresh <= vmin < vmax <= linthresh:
(12)                    return sorted({vmin, 0, vmax})
(8)                 has_a = (vmin < -linthresh)
(8)                 has_c = (vmax > linthresh)
(8)                 has_b = (has_a and vmax > -linthresh) or (has_c and vmin < linthresh)
(8)                 base = self._base
(8)                 def get_log_range(lo, hi):
(12)                    lo = np.floor(np.log(lo) / np.log(base))
(12)                    hi = np.ceil(np.log(hi) / np.log(base))
(12)                    return lo, hi
(8)                 a_lo, a_hi = (0, 0)
(8)                 if has_a:
(12)                    a_upper_lim = min(-linthresh, vmax)
(12)                    a_lo, a_hi = get_log_range(abs(a_upper_lim), abs(vmin) + 1)
(8)                 c_lo, c_hi = (0, 0)
(8)                 if has_c:
(12)                    c_lower_lim = max(linthresh, vmin)
(12)                    c_lo, c_hi = get_log_range(c_lower_lim, vmax + 1)
(8)                 total_ticks = (a_hi - a_lo) + (c_hi - c_lo)
(8)                 if has_b:
(12)                    total_ticks += 1
(8)                 stride = max(total_ticks // (self.numticks - 1), 1)
(8)                 decades = []
(8)                 if has_a:
(12)                    decades.extend(-1 * (base ** (np.arange(a_lo, a_hi,
```

```
(52)                                                                stride)[::-1])))
(8)                     if has_b:
(12)                        decades.append(0.0)
(8)                     if has_c:
(12)                        decades.extend(base ** (np.arange(c_lo, c_hi, stride)))
(8)                     subs = np.asarray(self._subs)
(8)                     if len(subs) > 1 or subs[0] != 1.0:
(12)                        ticklocs = []
(12)                        for decade in decades:
(16)                            if decade == 0:
(20)                                ticklocs.append(decade)
(16)                            else:
(20)                                ticklocs.extend(subs * decade)
(8)                     else:
(12)                        ticklocs = decades
(8)                     return self.raise_if_exceeds(np.array(ticklocs))
(4)                 def view_limits(self, vmin, vmax):
(8)                     """Try to choose the view limits intelligently."""
(8)                     b = self._base
(8)                     if vmax < vmin:
(12)                        vmin, vmax = vmax, vmin
(8)                     if mpl.rcParams['axes.autolimit_mode'] == 'round_numbers':
(12)                        vmin = _decade_less_equal(vmin, b)
(12)                        vmax = _decade_greater_equal(vmax, b)
(12)                        if vmin == vmax:
(16)                            vmin = _decade_less(vmin, b)
(16)                            vmax = _decade_greater(vmax, b)
(8)                     return mtransforms.nonsingular(vmin, vmax)
(0)             class AsinhLocator(Locator):
(4)                 """
(4)                 Place ticks spaced evenly on an inverse-sinh scale.
(4)                 Generally used with the `~.scale.AsinhScale` class.
(4)                 .. note::
(7)                    This API is provisional and may be revised in the future
(7)                    based on early user feedback.
(4)                 """
(4)                 def __init__(self, linear_width, numticks=11, symthresh=0.2,
(17)                             base=10, subs=None):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     linear_width : float
(12)                        The scale parameter defining the extent
(12)                        of the quasi-linear region.
(8)                     numticks : int, default: 11
(12)                        The approximate number of major ticks that will fit
(12)                        along the entire axis
(8)                     symthresh : float, default: 0.2
(12)                        The fractional threshold beneath which data which covers
(12)                        a range that is approximately symmetric about zero
(12)                        will have ticks that are exactly symmetric.
(8)                     base : int, default: 10
(12)                        The number base used for rounding tick locations
(12)                        on a logarithmic scale. If this is less than one,
(12)                        then rounding is to the nearest integer multiple
(12)                        of powers of ten.
(8)                     subs : tuple, default: None
(12)                        Multiples of the number base, typically used
(12)                        for the minor ticks, e.g. (2, 5) when base=10.
(8)                     """
(8)                     super().__init__()
(8)                     self.linear_width = linear_width
(8)                     self.numticks = numticks
(8)                     self.symthresh = symthresh
(8)                     self.base = base
(8)                     self.subs = subs
(4)                 def set_params(self, numticks=None, symthresh=None,
(19)                               base=None, subs=None):
(8)                     """Set parameters within this locator."""
```

```
2088: (8)                    if numticks is not None:
2089: (12)                       self.numticks = numticks
2090: (8)                    if symthresh is not None:
2091: (12)                       self.symthresh = symthresh
2092: (8)                    if base is not None:
2093: (12)                       self.base = base
2094: (8)                    if subs is not None:
2095: (12)                       self.subs = subs if len(subs) > 0 else None
2096: (4)                def __call__(self):
2097: (8)                    vmin, vmax = self.axis.get_view_interval()
2098: (8)                    if (vmin * vmax) < 0 and abs(1 + vmax / vmin) < self.symthresh:
2099: (12)                       bound = max(abs(vmin), abs(vmax))
2100: (12)                       return self.tick_values(-bound, bound)
2101: (8)                    else:
2102: (12)                       return self.tick_values(vmin, vmax)
2103: (4)                def tick_values(self, vmin, vmax):
2104: (8)                    ymin, ymax = self.linear_width * np.arcsinh(np.array([vmin, vmax])
2105: (52)                                                          / self.linear_width)
2106: (8)                    ys = np.linspace(ymin, ymax, self.numticks)
2107: (8)                    zero_dev = abs(ys / (ymax - ymin))
2108: (8)                    if ymin * ymax < 0:
2109: (12)                       ys = np.hstack([ys[(zero_dev > 0.5 / self.numticks)], 0.0])
2110: (8)                    xs = self.linear_width * np.sinh(ys / self.linear_width)
2111: (8)                    zero_xs = (ys == 0)
2112: (8)                    with np.errstate(divide="ignore"):  # base ** log(0) = base ** -inf =
0.
2113: (12)                       if self.base > 1:
2114: (16)                           pows = (np.sign(xs)
2115: (24)                                   * self.base ** np.floor(np.log(abs(xs)) /
math.log(self.base)))
2116: (16)                           qs = np.outer(pows, self.subs).flatten() if self.subs else
pows
2117: (12)                       else:  # No need to adjust sign(pows), as it cancels out when
computing qs.
2118: (16)                           pows = np.where(zero_xs, 1, 10**np.floor(np.log10(abs(xs))))
2119: (16)                           qs = pows * np.round(xs / pows)
2120: (8)                    ticks = np.array(sorted(set(qs)))
2121: (8)                    return ticks if len(ticks) >= 2 else np.linspace(vmin, vmax,
self.numticks)
2122: (0)            class LogitLocator(MaxNLocator):
2123: (4)                """
2124: (4)                Place ticks spaced evenly on a logit scale.
2125: (4)                """
2126: (4)                def __init__(self, minor=False, *, nbins="auto"):
2127: (8)                    """
2128: (8)                    Parameters
2129: (8)                    ----------
2130: (8)                    nbins : int or 'auto', optional
2131: (12)                       Number of ticks. Only used if minor is False.
2132: (8)                    minor : bool, default: False
2133: (12)                       Indicate if this locator is for minor ticks or not.
2134: (8)                    """
2135: (8)                    self._minor = minor
2136: (8)                    super().__init__(nbins=nbins, steps=[1, 2, 5, 10])
2137: (4)                def set_params(self, minor=None, **kwargs):
2138: (8)                    """Set parameters within this locator."""
2139: (8)                    if minor is not None:
2140: (12)                       self._minor = minor
2141: (8)                    super().set_params(**kwargs)
2142: (4)                @property
2143: (4)                def minor(self):
2144: (8)                    return self._minor
2145: (4)                @minor.setter
2146: (4)                def minor(self, value):
2147: (8)                    self.set_params(minor=value)
2148: (4)                def tick_values(self, vmin, vmax):
2149: (8)                    if hasattr(self.axis, "axes") and self.axis.axes.name == "polar":
2150: (12)                       raise NotImplementedError("Polar axis cannot be logit scaled yet")
2151: (8)                    if self._nbins == "auto":
```

```
(12)                        if self.axis is not None:
(16)                            nbins = self.axis.get_tick_space()
(16)                            if nbins < 2:
(20)                                nbins = 2
(12)                        else:
(16)                            nbins = 9
(8)                     else:
(12)                        nbins = self._nbins
(8)                     def ideal_ticks(x):
(12)                        return 10 ** x if x < 0 else 1 - (10 ** (-x)) if x > 0 else 0.5
(8)                     vmin, vmax = self.nonsingular(vmin, vmax)
(8)                     binf = int(
(12)                        np.floor(np.log10(vmin))
(12)                        if vmin < 0.5
(12)                        else 0
(12)                        if vmin < 0.9
(12)                        else -np.ceil(np.log10(1 - vmin))
(8)                     )
(8)                     bsup = int(
(12)                        np.ceil(np.log10(vmax))
(12)                        if vmax <= 0.5
(12)                        else 1
(12)                        if vmax <= 0.9
(12)                        else -np.floor(np.log10(1 - vmax))
(8)                     )
(8)                     numideal = bsup - binf - 1
(8)                     if numideal >= 2:
(12)                        if numideal > nbins:
(16)                            subsampling_factor = math.ceil(numideal / nbins)
(16)                            if self._minor:
(20)                                ticklocs = [
(24)                                    ideal_ticks(b)
(24)                                    for b in range(binf, bsup + 1)
(24)                                    if (b % subsampling_factor) != 0
(20)                                ]
(16)                            else:
(20)                                ticklocs = [
(24)                                    ideal_ticks(b)
(24)                                    for b in range(binf, bsup + 1)
(24)                                    if (b % subsampling_factor) == 0
(20)                                ]
(16)                            return self.raise_if_exceeds(np.array(ticklocs))
(12)                        if self._minor:
(16)                            ticklocs = []
(16)                            for b in range(binf, bsup):
(20)                                if b < -1:
(24)                                    ticklocs.extend(np.arange(2, 10) * 10 ** b)
(20)                                elif b == -1:
(24)                                    ticklocs.extend(np.arange(2, 5) / 10)
(20)                                elif b == 0:
(24)                                    ticklocs.extend(np.arange(6, 9) / 10)
(20)                                else:
(24)                                    ticklocs.extend(
(28)                                        1 - np.arange(2, 10)[::-1] * 10 ** (-b - 1)
(24)                                    )
(16)                            return self.raise_if_exceeds(np.array(ticklocs))
(12)                        ticklocs = [ideal_ticks(b) for b in range(binf, bsup + 1)]
(12)                        return self.raise_if_exceeds(np.array(ticklocs))
(8)                     if self._minor:
(12)                        return []
(8)                     return super().tick_values(vmin, vmax)
(4)                 def nonsingular(self, vmin, vmax):
(8)                     standard_minpos = 1e-7
(8)                     initial_range = (standard_minpos, 1 - standard_minpos)
(8)                     if vmin > vmax:
(12)                        vmin, vmax = vmax, vmin
(8)                     if not np.isfinite(vmin) or not np.isfinite(vmax):
(12)                        vmin, vmax = initial_range  # Initial range, no data plotted yet.
(8)                     elif vmax <= 0 or vmin >= 1:
```

```
2221: (12)                        _api.warn_external(
2222: (16)                            "Data has no values between 0 and 1, and therefore cannot be "
2223: (16)                            "logit-scaled."
2224: (12)                        )
2225: (12)                        vmin, vmax = initial_range
2226: (8)                    else:
2227: (12)                        minpos = (
2228: (16)                            self.axis.get_minpos()
2229: (16)                            if self.axis is not None
2230: (16)                            else standard_minpos
2231: (12)                        )
2232: (12)                        if not np.isfinite(minpos):
2233: (16)                            minpos = standard_minpos  # This should never take effect.
2234: (12)                        if vmin <= 0:
2235: (16)                            vmin = minpos
2236: (12)                        if vmax >= 1:
2237: (16)                            vmax = 1 - minpos
2238: (12)                        if vmin == vmax:
2239: (16)                            vmin, vmax = 0.1 * vmin, 1 - 0.1 * vmin
2240: (8)                    return vmin, vmax
2241: (0)            class AutoLocator(MaxNLocator):
2242: (4)                """
2243: (4)                Place evenly spaced ticks, with the step size and maximum number of ticks
chosen
2244: (4)                automatically.
2245: (4)                This is a subclass of `~matplotlib.ticker.MaxNLocator`, with parameters
2246: (4)                *nbins = 'auto'* and *steps = [1, 2, 2.5, 5, 10]*.
2247: (4)                """
2248: (4)                def __init__(self):
2249: (8)                    """
2250: (8)                    To know the values of the non-public parameters, please have a
2251: (8)                    look to the defaults of `~matplotlib.ticker.MaxNLocator`.
2252: (8)                    """
2253: (8)                    if mpl.rcParams['_internal.classic_mode']:
2254: (12)                        nbins = 9
2255: (12)                        steps = [1, 2, 5, 10]
2256: (8)                    else:
2257: (12)                        nbins = 'auto'
2258: (12)                        steps = [1, 2, 2.5, 5, 10]
2259: (8)                    super().__init__(nbins=nbins, steps=steps)
2260: (0)            class AutoMinorLocator(Locator):
2261: (4)                """
2262: (4)                Place evenly spaced minor ticks, with the step size and maximum number of
ticks
2263: (4)                chosen automatically.
2264: (4)                The Axis scale must be linear with evenly spaced major ticks .
2265: (4)                """
2266: (4)                def __init__(self, n=None):
2267: (8)                    """
2268: (8)                    *n* is the number of subdivisions of the interval between
2269: (8)                    major ticks; e.g., n=2 will place a single minor tick midway
2270: (8)                    between major ticks.
2271: (8)                    If *n* is omitted or None, the value stored in rcParams will be used.
2272: (8)                    In case *n* is set to 'auto', it will be set to 4 or 5. If the
distance
2273: (8)                    between the major ticks equals 1, 2.5, 5 or 10 it can be perfectly
2274: (8)                    divided in 5 equidistant sub-intervals with a length multiple of
2275: (8)                    0.05. Otherwise it is divided in 4 sub-intervals.
2276: (8)                    """
2277: (8)                    self.ndivs = n
2278: (4)                def __call__(self):
2279: (8)                    if self.axis.get_scale() == 'log':
2280: (12)                        _api.warn_external('AutoMinorLocator does not work on logarithmic
scales')
2281: (12)                        return []
2282: (8)                    majorlocs = np.unique(self.axis.get_majorticklocs())
2283: (8)                    if len(majorlocs) < 2:
2284: (12)                        return []
2285: (8)                    majorstep = majorlocs[1] - majorlocs[0]
```

```
2286: (8)                   if self.ndivs is None:
2287: (12)                      self.ndivs = mpl.rcParams[
2288: (16)                          'ytick.minor.ndivs' if self.axis.axis_name == 'y'
2289: (16)                          else 'xtick.minor.ndivs']  # for x and z axis
2290: (8)                   if self.ndivs == 'auto':
2291: (12)                      majorstep_mantissa = 10 ** (np.log10(majorstep) % 1)
2292: (12)                      ndivs = 5 if np.isclose(majorstep_mantissa, [1, 2.5, 5, 10]).any()
else 4
2293: (8)                   else:
2294: (12)                      ndivs = self.ndivs
2295: (8)                   minorstep = majorstep / ndivs
2296: (8)                   vmin, vmax = sorted(self.axis.get_view_interval())
2297: (8)                   t0 = majorlocs[0]
2298: (8)                   tmin = round((vmin - t0) / minorstep)
2299: (8)                   tmax = round((vmax - t0) / minorstep) + 1
2300: (8)                   locs = (np.arange(tmin, tmax) * minorstep) + t0
2301: (8)                   return self.raise_if_exceeds(locs)
2302: (4)               def tick_values(self, vmin, vmax):
2303: (8)                   raise NotImplementedError(
2304: (12)                      f"Cannot get tick locations for a {type(self).__name__}")

----------------------------------------

File 68 - transforms.py:

1: (0)              """
2: (0)              Matplotlib includes a framework for arbitrary geometric
3: (0)              transformations that is used determine the final position of all
4: (0)              elements drawn on the canvas.
5: (0)              Transforms are composed into trees of `TransformNode` objects
6: (0)              whose actual value depends on their children.  When the contents of
7: (0)              children change, their parents are automatically invalidated.  The
8: (0)              next time an invalidated transform is accessed, it is recomputed to
9: (0)              reflect those changes.  This invalidation/caching approach prevents
10: (0)             unnecessary recomputations of transforms, and contributes to better
11: (0)             interactive performance.
12: (0)             For example, here is a graph of the transform tree used to plot data
13: (0)             to the graph:
14: (0)             .. image:: ../_static/transforms.png
15: (0)             The framework can be used for both affine and non-affine
16: (0)             transformations.  However, for speed, we want to use the backend
17: (0)             renderers to perform affine transformations whenever possible.
18: (0)             Therefore, it is possible to perform just the affine or non-affine
19: (0)             part of a transformation on a set of data.  The affine is always
20: (0)             assumed to occur after the non-affine.  For any transform::
21: (2)               full transform == non-affine part + affine part
22: (0)             The backends are not expected to handle non-affine transformations
23: (0)             themselves.
24: (0)             See the tutorial :ref:`transforms_tutorial` for examples
25: (0)             of how to use transforms.
26: (0)             """
27: (0)             import copy
28: (0)             import functools
29: (0)             import textwrap
30: (0)             import weakref
31: (0)             import math
32: (0)             import numpy as np
33: (0)             from numpy.linalg import inv
34: (0)             from matplotlib import _api
35: (0)             from matplotlib._path import (
36: (4)                 affine_transform, count_bboxes_overlapping_bbox, update_path_extents)
37: (0)             from .path import Path
38: (0)             DEBUG = False
39: (0)             def _make_str_method(*args, **kwargs):
40: (4)                 """
41: (4)                 Generate a ``__str__`` method for a `.Transform` subclass.
42: (4)                 After ::
43: (8)                     class T:
44: (12)                        __str__ = _make_str_method("attr", key="other")
```

```
45: (4)                 ``str(T(...))`` will be
46: (4)                 .. code-block:: text
47: (8)                     {type(T).__name__}(
48: (12)                        {self.attr},
49: (12)                        key={self.other})
50: (4)                 """
51: (4)                 indent = functools.partial(textwrap.indent, prefix=" " * 4)
52: (4)                 def strrepr(x): return repr(x) if isinstance(x, str) else str(x)
53: (4)                 return lambda self: (
54: (8)                     type(self).__name__ + "("
55: (8)                     + ",".join([*(indent("\n" + strrepr(getattr(self, arg)))
56: (22)                                  for arg in args),
57: (20)                                 *(indent("\n" + k + "=" + strrepr(getattr(self, arg)))
58: (22)                                  for k, arg in kwargs.items())])
59: (8)                     + ")")
60: (0)             class TransformNode:
61: (4)                 """
62: (4)                 The base class for anything that participates in the transform tree
63: (4)                 and needs to invalidate its parents or be invalidated.  This includes
64: (4)                 classes that are not really transforms, such as bounding boxes, since some
65: (4)                 transforms depend on bounding boxes to compute their values.
66: (4)                 """
67: (4)                 INVALID_NON_AFFINE = _api.deprecated("3.8")(_api.classproperty(lambda cls:
1))
68: (4)                 INVALID_AFFINE = _api.deprecated("3.8")(_api.classproperty(lambda cls: 2))
69: (4)                 INVALID = _api.deprecated("3.8")(_api.classproperty(lambda cls: 3))
70: (4)                 _VALID, _INVALID_AFFINE_ONLY, _INVALID_FULL = range(3)
71: (4)                 is_affine = False
72: (4)                 is_bbox = _api.deprecated("3.9")(_api.classproperty(lambda cls: False))
73: (4)                 pass_through = False
74: (4)                 """
75: (4)                 If pass_through is True, all ancestors will always be
76: (4)                 invalidated, even if 'self' is already invalid.
77: (4)                 """
78: (4)                 def __init__(self, shorthand_name=None):
79: (8)                     """
80: (8)                     Parameters
81: (8)                     ----------
82: (8)                     shorthand_name : str
83: (12)                        A string representing the "name" of the transform. The name
carries
84: (12)                        no significance other than to improve the readability of
85: (12)                        ``str(transform)`` when DEBUG=True.
86: (8)                     """
87: (8)                     self._parents = {}
88: (8)                     self._invalid = self._INVALID_FULL
89: (8)                     self._shorthand_name = shorthand_name or ''
90: (4)                 if DEBUG:
91: (8)                     def __str__(self):
92: (12)                        return self._shorthand_name or repr(self)
93: (4)                 def __getstate__(self):
94: (8)                     return {**self.__dict__,
95: (16)                             '_parents': {k: v() for k, v in self._parents.items()}}
96: (4)                 def __setstate__(self, data_dict):
97: (8)                     self.__dict__ = data_dict
98: (8)                     self._parents = {
99: (12)                        k: weakref.ref(v, lambda _, pop=self._parents.pop, k=k: pop(k))
100: (12)                       for k, v in self._parents.items() if v is not None}
101: (4)                def __copy__(self):
102: (8)                    other = copy.copy(super())
103: (8)                    other._parents = {}
104: (8)                    for key, val in vars(self).items():
105: (12)                       if isinstance(val, TransformNode) and id(self) in val._parents:
106: (16)                           other.set_children(val)  # val == getattr(other, key)
107: (8)                    return other
108: (4)                def invalidate(self):
109: (8)                    """
110: (8)                    Invalidate this `TransformNode` and triggers an invalidation of its
111: (8)                    ancestors.  Should be called any time the transform changes.
```

```
112: (8)                    """
113: (8)                    return self._invalidate_internal(
114: (12)                       level=self._INVALID_AFFINE_ONLY if self.is_affine else
self._INVALID_FULL,
115: (12)                       invalidating_node=self)
116: (4)                def _invalidate_internal(self, level, invalidating_node):
117: (8)                    """
118: (8)                    Called by :meth:`invalidate` and subsequently ascends the transform
119: (8)                    stack calling each TransformNode's _invalidate_internal method.
120: (8)                    """
121: (8)                    if level <= self._invalid and not self.pass_through:
122: (12)                       return
123: (8)                    self._invalid = level
124: (8)                    for parent in list(self._parents.values()):
125: (12)                       parent = parent()  # Dereference the weak reference.
126: (12)                       if parent is not None:
127: (16)                           parent._invalidate_internal(level=level,
invalidating_node=self)
128: (4)                def set_children(self, *children):
129: (8)                    """
130: (8)                    Set the children of the transform, to let the invalidation
131: (8)                    system know which transforms can invalidate this transform.
132: (8)                    Should be called from the constructor of any transforms that
133: (8)                    depend on other transforms.
134: (8)                    """
135: (8)                    id_self = id(self)
136: (8)                    for child in children:
137: (12)                       ref = weakref.ref(
138: (16)                           self, lambda _, pop=child._parents.pop, k=id_self: pop(k))
139: (12)                       child._parents[id_self] = ref
140: (4)                def frozen(self):
141: (8)                    """
142: (8)                    Return a frozen copy of this transform node.  The frozen copy will not
143: (8)                    be updated when its children change.  Useful for storing a previously
144: (8)                    known state of a transform where ``copy.deepcopy()`` might normally be
145: (8)                    used.
146: (8)                    """
147: (8)                    return self
148: (0)            class BboxBase(TransformNode):
149: (4)                """
150: (4)                The base class of all bounding boxes.
151: (4)                This class is immutable; `Bbox` is a mutable subclass.
152: (4)                The canonical representation is as two points, with no
153: (4)                restrictions on their ordering.  Convenience properties are
154: (4)                provided to get the left, bottom, right and top edges and width
155: (4)                and height, but these are not stored explicitly.
156: (4)                """
157: (4)                is_bbox = _api.deprecated("3.9")(_api.classproperty(lambda cls: True))
158: (4)                is_affine = True
159: (4)                if DEBUG:
160: (8)                    @staticmethod
161: (8)                    def _check(points):
162: (12)                       if isinstance(points, np.ma.MaskedArray):
163: (16)                           _api.warn_external("Bbox bounds are a masked array.")
164: (12)                       points = np.asarray(points)
165: (12)                       if any((points[1, :] - points[0, :]) == 0):
166: (16)                           _api.warn_external("Singular Bbox.")
167: (4)                def frozen(self):
168: (8)                    return Bbox(self.get_points().copy())
169: (4)                frozen.__doc__ = TransformNode.__doc__
170: (4)                def __array__(self, *args, **kwargs):
171: (8)                    return self.get_points()
172: (4)                @property
173: (4)                def x0(self):
174: (8)                    """
175: (8)                    The first of the pair of *x* coordinates that define the bounding box.
176: (8)                    This is not guaranteed to be less than :attr:`x1` (for that, use
177: (8)                    :attr:`xmin`).
178: (8)                    """
```

```
(8)                 return self.get_points()[0, 0]
(4)             @property
(4)             def y0(self):
(8)                 """
(8)                 The first of the pair of *y* coordinates that define the bounding box.
(8)                 This is not guaranteed to be less than :attr:`y1` (for that, use
(8)                 :attr:`ymin`).
(8)                 """
(8)                 return self.get_points()[0, 1]
(4)             @property
(4)             def x1(self):
(8)                 """
(8)                 The second of the pair of *x* coordinates that define the bounding box.
(8)                 This is not guaranteed to be greater than :attr:`x0` (for that, use
(8)                 :attr:`xmax`).
(8)                 """
(8)                 return self.get_points()[1, 0]
(4)             @property
(4)             def y1(self):
(8)                 """
(8)                 The second of the pair of *y* coordinates that define the bounding box.
(8)                 This is not guaranteed to be greater than :attr:`y0` (for that, use
(8)                 :attr:`ymax`).
(8)                 """
(8)                 return self.get_points()[1, 1]
(4)             @property
(4)             def p0(self):
(8)                 """
(8)                 The first pair of (*x*, *y*) coordinates that define the bounding box.
(8)                 This is not guaranteed to be the bottom-left corner (for that, use
(8)                 :attr:`min`).
(8)                 """
(8)                 return self.get_points()[0]
(4)             @property
(4)             def p1(self):
(8)                 """
(8)                 The second pair of (*x*, *y*) coordinates that define the bounding box.
(8)                 This is not guaranteed to be the top-right corner (for that, use
(8)                 :attr:`max`).
(8)                 """
(8)                 return self.get_points()[1]
(4)             @property
(4)             def xmin(self):
(8)                 """The left edge of the bounding box."""
(8)                 return np.min(self.get_points()[:, 0])
(4)             @property
(4)             def ymin(self):
(8)                 """The bottom edge of the bounding box."""
(8)                 return np.min(self.get_points()[:, 1])
(4)             @property
(4)             def xmax(self):
(8)                 """The right edge of the bounding box."""
(8)                 return np.max(self.get_points()[:, 0])
(4)             @property
(4)             def ymax(self):
(8)                 """The top edge of the bounding box."""
(8)                 return np.max(self.get_points()[:, 1])
(4)             @property
(4)             def min(self):
(8)                 """The bottom-left corner of the bounding box."""
(8)                 return np.min(self.get_points(), axis=0)
(4)             @property
(4)             def max(self):
(8)                 """The top-right corner of the bounding box."""
(8)                 return np.max(self.get_points(), axis=0)
(4)             @property
```

```
    def intervalx(self):
        """
        The pair of *x* coordinates that define the bounding box.
        This is not guaranteed to be sorted from left to right.
        """
        return self.get_points()[:, 0]
    @property
    def intervaly(self):
        """
        The pair of *y* coordinates that define the bounding box.
        This is not guaranteed to be sorted from bottom to top.
        """
        return self.get_points()[:, 1]
    @property
    def width(self):
        """The (signed) width of the bounding box."""
        points = self.get_points()
        return points[1, 0] - points[0, 0]
    @property
    def height(self):
        """The (signed) height of the bounding box."""
        points = self.get_points()
        return points[1, 1] - points[0, 1]
    @property
    def size(self):
        """The (signed) width and height of the bounding box."""
        points = self.get_points()
        return points[1] - points[0]
    @property
    def bounds(self):
        """Return (:attr:`x0`, :attr:`y0`, :attr:`width`, :attr:`height`)."""
        (x0, y0), (x1, y1) = self.get_points()
        return (x0, y0, x1 - x0, y1 - y0)
    @property
    def extents(self):
        """Return (:attr:`x0`, :attr:`y0`, :attr:`x1`, :attr:`y1`)."""
        return self.get_points().flatten()  # flatten returns a copy.
    def get_points(self):
        raise NotImplementedError
    def containsx(self, x):
        """
        Return whether *x* is in the closed (:attr:`x0`, :attr:`x1`) interval.
        """
        x0, x1 = self.intervalx
        return x0 <= x <= x1 or x0 >= x >= x1
    def containsy(self, y):
        """
        Return whether *y* is in the closed (:attr:`y0`, :attr:`y1`) interval.
        """
        y0, y1 = self.intervaly
        return y0 <= y <= y1 or y0 >= y >= y1
    def contains(self, x, y):
        """
        Return whether ``(x, y)`` is in the bounding box or on its edge.
        """
        return self.containsx(x) and self.containsy(y)
    def overlaps(self, other):
        """
        Return whether this bounding box overlaps with the other bounding box.
        Parameters
        ----------
        other : `.BboxBase`
        """
        ax1, ay1, ax2, ay2 = self.extents
        bx1, by1, bx2, by2 = other.extents
        if ax2 < ax1:
            ax2, ax1 = ax1, ax2
        if ay2 < ay1:
            ay2, ay1 = ay1, ay2
```

```
        if bx2 < bx1:
            bx2, bx1 = bx1, bx2
        if by2 < by1:
            by2, by1 = by1, by2
        return ax1 <= bx2 and bx1 <= ax2 and ay1 <= by2 and by1 <= ay2
    def fully_containsx(self, x):
        """
        Return whether *x* is in the open (:attr:`x0`, :attr:`x1`) interval.
        """
        x0, x1 = self.intervalx
        return x0 < x < x1 or x0 > x > x1
    def fully_containsy(self, y):
        """
        Return whether *y* is in the open (:attr:`y0`, :attr:`y1`) interval.
        """
        y0, y1 = self.intervaly
        return y0 < y < y1 or y0 > y > y1
    def fully_contains(self, x, y):
        """
        Return whether ``x, y`` is in the bounding box, but not on its edge.
        """
        return self.fully_containsx(x) and self.fully_containsy(y)
    def fully_overlaps(self, other):
        """
        Return whether this bounding box overlaps with the other bounding box,
        not including the edges.
        Parameters
        ----------
        other : `.BboxBase`
        """
        ax1, ay1, ax2, ay2 = self.extents
        bx1, by1, bx2, by2 = other.extents
        if ax2 < ax1:
            ax2, ax1 = ax1, ax2
        if ay2 < ay1:
            ay2, ay1 = ay1, ay2
        if bx2 < bx1:
            bx2, bx1 = bx1, bx2
        if by2 < by1:
            by2, by1 = by1, by2
        return ax1 < bx2 and bx1 < ax2 and ay1 < by2 and by1 < ay2
    def transformed(self, transform):
        """
        Construct a `Bbox` by statically transforming this one by *transform*.
        """
        pts = self.get_points()
        ll, ul, lr = transform.transform(np.array(
            [pts[0], [pts[0, 0], pts[1, 1]], [pts[1, 0], pts[0, 1]]]))
        return Bbox([ll, [lr[0], ul[1]]])
    coefs = {'C':  (0.5, 0.5),
             'SW': (0, 0),
             'S':  (0.5, 0),
             'SE': (1.0, 0),
             'E':  (1.0, 0.5),
             'NE': (1.0, 1.0),
             'N':  (0.5, 1.0),
             'NW': (0, 1.0),
             'W':  (0, 0.5)}
    def anchored(self, c, container=None):
        """
        Return a copy of the `Bbox` anchored to *c* within *container*.
        Parameters
        ----------
        c : (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...}
            Either an (*x*, *y*) pair of relative coordinates (0 is left or
            bottom, 1 is right or top), 'C' (center), or a cardinal direction
            ('SW', southwest, is bottom left, etc.).
        container : `Bbox`, optional
            The box within which the `Bbox` is positioned.
```

```
(8)                     See Also
(8)                     --------
(8)                     .Axes.set_anchor
(8)                     """
(8)                     if container is None:
(12)                        _api.warn_deprecated(
(16)                            "3.8", message="Calling anchored() with no container bbox "
(16)                            "returns a frozen copy of the original bbox and is deprecated
"
(16)                            "since %(since)s.")
(12)                        container = self
(8)                     l, b, w, h = container.bounds
(8)                     L, B, W, H = self.bounds
(8)                     cx, cy = self.coefs[c] if isinstance(c, str) else c
(8)                     return Bbox(self._points +
(20)                                [(l + cx * (w - W)) - L,
(21)                                 (b + cy * (h - H)) - B])
(4)                 def shrunk(self, mx, my):
(8)                     """
(8)                     Return a copy of the `Bbox`, shrunk by the factor *mx*
(8)                     in the *x* direction and the factor *my* in the *y* direction.
(8)                     The lower left corner of the box remains unchanged.  Normally
(8)                     *mx* and *my* will be less than 1, but this is not enforced.
(8)                     """
(8)                     w, h = self.size
(8)                     return Bbox([self._points[0],
(21)                                 self._points[0] + [mx * w, my * h]])
(4)                 def shrunk_to_aspect(self, box_aspect, container=None, fig_aspect=1.0):
(8)                     """
(8)                     Return a copy of the `Bbox`, shrunk so that it is as
(8)                     large as it can be while having the desired aspect ratio,
(8)                     *box_aspect*.  If the box coordinates are relative (i.e.
(8)                     fractions of a larger box such as a figure) then the
(8)                     physical aspect ratio of that figure is specified with
(8)                     *fig_aspect*, so that *box_aspect* can also be given as a
(8)                     ratio of the absolute dimensions, not the relative dimensions.
(8)                     """
(8)                     if box_aspect <= 0 or fig_aspect <= 0:
(12)                        raise ValueError("'box_aspect' and 'fig_aspect' must be positive")
(8)                     if container is None:
(12)                        container = self
(8)                     w, h = container.size
(8)                     H = w * box_aspect / fig_aspect
(8)                     if H <= h:
(12)                        W = w
(8)                     else:
(12)                        W = h * fig_aspect / box_aspect
(12)                        H = h
(8)                     return Bbox([self._points[0],
(21)                                 self._points[0] + (W, H)])
(4)                 def splitx(self, *args):
(8)                     """
(8)                     Return a list of new `Bbox` objects formed by splitting the original
(8)                     one with vertical lines at fractional positions given by *args*.
(8)                     """
(8)                     xf = [0, *args, 1]
(8)                     x0, y0, x1, y1 = self.extents
(8)                     w = x1 - x0
(8)                     return [Bbox([[x0 + xf0 * w, y0], [x0 + xf1 * w, y1]])
(16)                            for xf0, xf1 in zip(xf[:-1], xf[1:])]
(4)                 def splity(self, *args):
(8)                     """
(8)                     Return a list of new `Bbox` objects formed by splitting the original
(8)                     one with horizontal lines at fractional positions given by *args*.
(8)                     """
(8)                     yf = [0, *args, 1]
(8)                     x0, y0, x1, y1 = self.extents
(8)                     h = y1 - y0
(8)                     return [Bbox([[x0, y0 + yf0 * h], [x1, y0 + yf1 * h]])
```

```
(16)                            for yf0, yf1 in zip(yf[:-1], yf[1:])]
(4)                 def count_contains(self, vertices):
(8)                     """
(8)                     Count the number of vertices contained in the `Bbox`.
(8)                     Any vertices with a non-finite x or y value are ignored.
(8)                     Parameters
(8)                     ----------
(8)                     vertices : (N, 2) array
(8)                     """
(8)                     if len(vertices) == 0:
(12)                        return 0
(8)                     vertices = np.asarray(vertices)
(8)                     with np.errstate(invalid='ignore'):
(12)                        return (((self.min < vertices) &
(21)                                 (vertices < self.max)).all(axis=1).sum())
(4)                 def count_overlaps(self, bboxes):
(8)                     """
(8)                     Count the number of bounding boxes that overlap this one.
(8)                     Parameters
(8)                     ----------
(8)                     bboxes : sequence of `.BboxBase`
(8)                     """
(8)                     return count_bboxes_overlapping_bbox(
(12)                        self, np.atleast_3d([np.array(x) for x in bboxes]))
(4)                 def expanded(self, sw, sh):
(8)                     """
(8)                     Construct a `Bbox` by expanding this one around its center by the
(8)                     factors *sw* and *sh*.
(8)                     """
(8)                     width = self.width
(8)                     height = self.height
(8)                     deltaw = (sw * width - width) / 2.0
(8)                     deltah = (sh * height - height) / 2.0
(8)                     a = np.array([[-deltaw, -deltah], [deltaw, deltah]])
(8)                     return Bbox(self._points + a)
(4)                 @_api.rename_parameter("3.8", "p", "w_pad")
(4)                 def padded(self, w_pad, h_pad=None):
(8)                     """
(8)                     Construct a `Bbox` by padding this one on all four sides.
(8)                     Parameters
(8)                     ----------
(8)                     w_pad : float
(12)                        Width pad
(8)                     h_pad : float, optional
(12)                        Height pad.  Defaults to *w_pad*.
(8)                     """
(8)                     points = self.get_points()
(8)                     if h_pad is None:
(12)                        h_pad = w_pad
(8)                     return Bbox(points + [[-w_pad, -h_pad], [w_pad, h_pad]])
(4)                 def translated(self, tx, ty):
(8)                     """Construct a `Bbox` by translating this one by *tx* and *ty*."""
(8)                     return Bbox(self._points + (tx, ty))
(4)                 def corners(self):
(8)                     """
(8)                     Return the corners of this rectangle as an array of points.
(8)                     Specifically, this returns the array
(8)                     ``[[x0, y0], [x0, y1], [x1, y0], [x1, y1]]``.
(8)                     """
(8)                     (x0, y0), (x1, y1) = self.get_points()
(8)                     return np.array([[x0, y0], [x0, y1], [x1, y0], [x1, y1]])
(4)                 def rotated(self, radians):
(8)                     """
(8)                     Return the axes-aligned bounding box that bounds the result of
rotating
(8)                     this `Bbox` by an angle of *radians*.
(8)                     """
(8)                     corners = self.corners()
(8)                     corners_rotated = Affine2D().rotate(radians).transform(corners)
```

```
519: (8)                    bbox = Bbox.unit()
520: (8)                    bbox.update_from_data_xy(corners_rotated, ignore=True)
521: (8)                    return bbox
522: (4)                @staticmethod
523: (4)                def union(bboxes):
524: (8)                    """Return a `Bbox` that contains all of the given *bboxes*."""
525: (8)                    if not len(bboxes):
526: (12)                       raise ValueError("'bboxes' cannot be empty")
527: (8)                    x0 = np.min([bbox.xmin for bbox in bboxes])
528: (8)                    x1 = np.max([bbox.xmax for bbox in bboxes])
529: (8)                    y0 = np.min([bbox.ymin for bbox in bboxes])
530: (8)                    y1 = np.max([bbox.ymax for bbox in bboxes])
531: (8)                    return Bbox([[x0, y0], [x1, y1]])
532: (4)                @staticmethod
533: (4)                def intersection(bbox1, bbox2):
534: (8)                    """
535: (8)                    Return the intersection of *bbox1* and *bbox2* if they intersect, or
536: (8)                    None if they don't.
537: (8)                    """
538: (8)                    x0 = np.maximum(bbox1.xmin, bbox2.xmin)
539: (8)                    x1 = np.minimum(bbox1.xmax, bbox2.xmax)
540: (8)                    y0 = np.maximum(bbox1.ymin, bbox2.ymin)
541: (8)                    y1 = np.minimum(bbox1.ymax, bbox2.ymax)
542: (8)                    return Bbox([[x0, y0], [x1, y1]]) if x0 <= x1 and y0 <= y1 else None
543: (0)            _default_minpos = np.array([np.inf, np.inf])
544: (0)            class Bbox(BboxBase):
545: (4)                """
546: (4)                A mutable bounding box.
547: (4)                Examples
548: (4)                --------
549: (4)                **Create from known bounds**
550: (4)                The default constructor takes the boundary "points" ``[[xmin, ymin],
551: (4)                [xmax, ymax]]``.
552: (8)                    >>> Bbox([[1, 1], [3, 7]])
553: (8)                    Bbox([[1.0, 1.0], [3.0, 7.0]])
554: (4)                Alternatively, a Bbox can be created from the flattened points array, the
555: (4)                so-called "extents" ``(xmin, ymin, xmax, ymax)``
556: (8)                    >>> Bbox.from_extents(1, 1, 3, 7)
557: (8)                    Bbox([[1.0, 1.0], [3.0, 7.0]])
558: (4)                or from the "bounds" ``(xmin, ymin, width, height)``.
559: (8)                    >>> Bbox.from_bounds(1, 1, 2, 6)
560: (8)                    Bbox([[1.0, 1.0], [3.0, 7.0]])
561: (4)                **Create from collections of points**
562: (4)                The "empty" object for accumulating Bboxs is the null bbox, which is a
563: (4)                stand-in for the empty set.
564: (8)                    >>> Bbox.null()
565: (8)                    Bbox([[inf, inf], [-inf, -inf]])
566: (4)                Adding points to the null bbox will give you the bbox of those points.
567: (8)                    >>> box = Bbox.null()
568: (8)                    >>> box.update_from_data_xy([[1, 1]])
569: (8)                    >>> box
570: (8)                    Bbox([[1.0, 1.0], [1.0, 1.0]])
571: (8)                    >>> box.update_from_data_xy([[2, 3], [3, 2]], ignore=False)
572: (8)                    >>> box
573: (8)                    Bbox([[1.0, 1.0], [3.0, 3.0]])
574: (4)                Setting ``ignore=True`` is equivalent to starting over from a null bbox.
575: (8)                    >>> box.update_from_data_xy([[1, 1]], ignore=True)
576: (8)                    >>> box
577: (8)                    Bbox([[1.0, 1.0], [1.0, 1.0]])
578: (4)                .. warning::
579: (8)                    It is recommended to always specify ``ignore`` explicitly.  If not,
the
580: (8)                    default value of ``ignore`` can be changed at any time by code with
581: (8)                    access to your Bbox, for example using the method `~.Bbox.ignore`.
582: (4)                **Properties of the ``null`` bbox**
583: (4)                .. note::
584: (8)                    The current behavior of `Bbox.null()` may be surprising as it does
585: (8)                    not have all of the properties of the "empty set", and as such does
586: (8)                    not behave like a "zero" object in the mathematical sense. We may
```

```
(8)                 change that in the future (with a deprecation period).
(4)             The null bbox is the identity for intersections
(8)                 >>> Bbox.intersection(Bbox([[1, 1], [3, 7]]), Bbox.null())
(8)                 Bbox([[1.0, 1.0], [3.0, 7.0]])
(4)             except with itself, where it returns the full space.
(8)                 >>> Bbox.intersection(Bbox.null(), Bbox.null())
(8)                 Bbox([[-inf, -inf], [inf, inf]])
(4)             A union containing null will always return the full space (not the other
(4)             set!)
(8)                 >>> Bbox.union([Bbox([[0, 0], [0, 0]]), Bbox.null()])
(8)                 Bbox([[-inf, -inf], [inf, inf]])
(4)             """
(4)             def __init__(self, points, **kwargs):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 points : `~numpy.ndarray`
(12)                    A (2, 2) array of the form ``[[x0, y0], [x1, y1]]``.
(8)                 """
(8)                 super().__init__(**kwargs)
(8)                 points = np.asarray(points, float)
(8)                 if points.shape != (2, 2):
(12)                    raise ValueError('Bbox points must be of the form '
(29)                                     '"[[x0, y0], [x1, y1]]".')
(8)                 self._points = points
(8)                 self._minpos = _default_minpos.copy()
(8)                 self._ignore = True
(8)                 self._points_orig = self._points.copy()
(4)             if DEBUG:
(8)                 ___init__ = __init__
(8)                 def __init__(self, points, **kwargs):
(12)                    self._check(points)
(12)                    self.___init__(points, **kwargs)
(8)                 def invalidate(self):
(12)                    self._check(self._points)
(12)                    super().invalidate()
(4)             def frozen(self):
(8)                 frozen_bbox = super().frozen()
(8)                 frozen_bbox._minpos = self.minpos.copy()
(8)                 return frozen_bbox
(4)             @staticmethod
(4)             def unit():
(8)                 """Create a new unit `Bbox` from (0, 0) to (1, 1)."""
(8)                 return Bbox([[0, 0], [1, 1]])
(4)             @staticmethod
(4)             def null():
(8)                 """Create a new null `Bbox` from (inf, inf) to (-inf, -inf)."""
(8)                 return Bbox([[np.inf, np.inf], [-np.inf, -np.inf]])
(4)             @staticmethod
(4)             def from_bounds(x0, y0, width, height):
(8)                 """
(8)                 Create a new `Bbox` from *x0*, *y0*, *width* and *height*.
(8)                 *width* and *height* may be negative.
(8)                 """
(8)                 return Bbox.from_extents(x0, y0, x0 + width, y0 + height)
(4)             @staticmethod
(4)             def from_extents(*args, minpos=None):
(8)                 """
(8)                 Create a new Bbox from *left*, *bottom*, *right* and *top*.
(8)                 The *y*-axis increases upwards.
(8)                 Parameters
(8)                 ----------
(8)                 left, bottom, right, top : float
(12)                    The four extents of the bounding box.
(8)                 minpos : float or None
(12)                    If this is supplied, the Bbox will have a minimum positive value
(12)                    set. This is useful when dealing with logarithmic scales and other
(12)                    scales where negative bounds result in floating point errors.
(8)                 """
```

```
(8)                 bbox = Bbox(np.reshape(args, (2, 2)))
(8)                 if minpos is not None:
(12)                    bbox._minpos[:] = minpos
(8)                 return bbox
(4)             def __format__(self, fmt):
(8)                 return (
(12)                    'Bbox(x0={0.x0:{1}}, y0={0.y0:{1}}, x1={0.x1:{1}}, y1={0.y1:
{1}})'.
(12)                    format(self, fmt))
(4)             def __str__(self):
(8)                 return format(self, '')
(4)             def __repr__(self):
(8)                 return 'Bbox([[{0.x0}, {0.y0}], [{0.x1}, {0.y1}]])'.format(self)
(4)             def ignore(self, value):
(8)                 """
(8)                 Set whether the existing bounds of the box should be ignored
(8)                 by subsequent calls to :meth:`update_from_data_xy`.
(8)                 value : bool
(12)                    - When ``True``, subsequent calls to `update_from_data_xy` will
(14)                      ignore the existing bounds of the `Bbox`.
(12)                    - When ``False``, subsequent calls to `update_from_data_xy` will
(14)                      include the existing bounds of the `Bbox`.
(8)                 """
(8)                 self._ignore = value
(4)             def update_from_path(self, path, ignore=None, updatex=True, updatey=True):
(8)                 """
(8)                 Update the bounds of the `Bbox` to contain the vertices of the
(8)                 provided path. After updating, the bounds will have positive *width*
(8)                 and *height*; *x0* and *y0* will be the minimal values.
(8)                 Parameters
(8)                 ----------
(8)                 path : `~matplotlib.path.Path`
(8)                 ignore : bool, optional
(11)                   - When ``True``, ignore the existing bounds of the `Bbox`.
(11)                   - When ``False``, include the existing bounds of the `Bbox`.
(11)                   - When ``None``, use the last value passed to :meth:`ignore`.
(8)                 updatex, updatey : bool, default: True
(12)                    When ``True``, update the x/y values.
(8)                 """
(8)                 if ignore is None:
(12)                    ignore = self._ignore
(8)                 if path.vertices.size == 0:
(12)                    return
(8)                 points, minpos, changed = update_path_extents(
(12)                    path, None, self._points, self._minpos, ignore)
(8)                 if changed:
(12)                    self.invalidate()
(12)                    if updatex:
(16)                        self._points[:, 0] = points[:, 0]
(16)                        self._minpos[0] = minpos[0]
(12)                    if updatey:
(16)                        self._points[:, 1] = points[:, 1]
(16)                        self._minpos[1] = minpos[1]
(4)             def update_from_data_x(self, x, ignore=None):
(8)                 """
(8)                 Update the x-bounds of the `Bbox` based on the passed in data. After
(8)                 updating, the bounds will have positive *width*, and *x0* will be the
(8)                 minimal value.
(8)                 Parameters
(8)                 ----------
(8)                 x : `~numpy.ndarray`
(12)                    Array of x-values.
(8)                 ignore : bool, optional
(11)                   - When ``True``, ignore the existing bounds of the `Bbox`.
(11)                   - When ``False``, include the existing bounds of the `Bbox`.
(11)                   - When ``None``, use the last value passed to :meth:`ignore`.
(8)                 """
(8)                 x = np.ravel(x)
(8)                 self.update_from_data_xy(np.column_stack([x, np.ones(x.size)]),
```

```
(33)                                    ignore=ignore, updatey=False)
(4)                 def update_from_data_y(self, y, ignore=None):
(8)                     """
(8)                     Update the y-bounds of the `Bbox` based on the passed in data. After
(8)                     updating, the bounds will have positive *height*, and *y0* will be the
(8)                     minimal value.
(8)                     Parameters
(8)                     ----------
(8)                     y : `~numpy.ndarray`
(12)                        Array of y-values.
(8)                     ignore : bool, optional
(12)                        - When ``True``, ignore the existing bounds of the `Bbox`.
(12)                        - When ``False``, include the existing bounds of the `Bbox`.
(12)                        - When ``None``, use the last value passed to :meth:`ignore`.
(8)                     """
(8)                     y = np.ravel(y)
(8)                     self.update_from_data_xy(np.column_stack([np.ones(y.size), y]),
(33)                                    ignore=ignore, updatex=False)
(4)                 def update_from_data_xy(self, xy, ignore=None, updatex=True,
updatey=True):
(8)                     """
(8)                     Update the `Bbox` bounds based on the passed in *xy* coordinates.
(8)                     After updating, the bounds will have positive *width* and *height*;
(8)                     *x0* and *y0* will be the minimal values.
(8)                     Parameters
(8)                     ----------
(8)                     xy : (N, 2) array-like
(12)                        The (x, y) coordinates.
(8)                     ignore : bool, optional
(12)                        - When ``True``, ignore the existing bounds of the `Bbox`.
(12)                        - When ``False``, include the existing bounds of the `Bbox`.
(12)                        - When ``None``, use the last value passed to :meth:`ignore`.
(8)                     updatex, updatey : bool, default: True
(13)                         When ``True``, update the x/y values.
(8)                     """
(8)                     if len(xy) == 0:
(12)                        return
(8)                     path = Path(xy)
(8)                     self.update_from_path(path, ignore=ignore,
(30)                                          updatex=updatex, updatey=updatey)
(4)                 @BboxBase.x0.setter
(4)                 def x0(self, val):
(8)                     self._points[0, 0] = val
(8)                     self.invalidate()
(4)                 @BboxBase.y0.setter
(4)                 def y0(self, val):
(8)                     self._points[0, 1] = val
(8)                     self.invalidate()
(4)                 @BboxBase.x1.setter
(4)                 def x1(self, val):
(8)                     self._points[1, 0] = val
(8)                     self.invalidate()
(4)                 @BboxBase.y1.setter
(4)                 def y1(self, val):
(8)                     self._points[1, 1] = val
(8)                     self.invalidate()
(4)                 @BboxBase.p0.setter
(4)                 def p0(self, val):
(8)                     self._points[0] = val
(8)                     self.invalidate()
(4)                 @BboxBase.p1.setter
(4)                 def p1(self, val):
(8)                     self._points[1] = val
(8)                     self.invalidate()
(4)                 @BboxBase.intervalx.setter
(4)                 def intervalx(self, interval):
(8)                     self._points[:, 0] = interval
(8)                     self.invalidate()
(4)                 @BboxBase.intervaly.setter
```

```
    def intervaly(self, interval):
        self._points[:, 1] = interval
        self.invalidate()
    @BboxBase.bounds.setter
    def bounds(self, bounds):
        l, b, w, h = bounds
        points = np.array([[l, b], [l + w, b + h]], float)
        if np.any(self._points != points):
            self._points = points
            self.invalidate()
    @property
    def minpos(self):
        """
        The minimum positive value in both directions within the Bbox.
        This is useful when dealing with logarithmic scales and other scales
        where negative bounds result in floating point errors, and will be used

        as the minimum extent instead of *p0*.
        """
        return self._minpos
    @minpos.setter
    def minpos(self, val):
        self._minpos[:] = val
    @property
    def minposx(self):
        """
        The minimum positive value in the *x*-direction within the Bbox.
        This is useful when dealing with logarithmic scales and other scales
        where negative bounds result in floating point errors, and will be used

        as the minimum *x*-extent instead of *x0*.
        """
        return self._minpos[0]
    @minposx.setter
    def minposx(self, val):
        self._minpos[0] = val
    @property
    def minposy(self):
        """
        The minimum positive value in the *y*-direction within the Bbox.
        This is useful when dealing with logarithmic scales and other scales
        where negative bounds result in floating point errors, and will be used

        as the minimum *y*-extent instead of *y0*.
        """
        return self._minpos[1]
    @minposy.setter
    def minposy(self, val):
        self._minpos[1] = val
    def get_points(self):
        """
        Get the points of the bounding box as an array of the form
        ``[[x0, y0], [x1, y1]]``.
        """
        self._invalid = 0
        return self._points
    def set_points(self, points):
        """
        Set the points of the bounding box directly from an array of the form
        ``[[x0, y0], [x1, y1]]``.  No error checking is performed, as this
        method is mainly for internal use.
        """
        if np.any(self._points != points):
            self._points = points
            self.invalidate()
    def set(self, other):
        """
        Set this bounding box from the "frozen" bounds of another `Bbox`.
        """
```

```
858: (8)                    if np.any(self._points != other.get_points()):
859: (12)                       self._points = other.get_points()
860: (12)                       self.invalidate()
861: (4)                def mutated(self):
862: (8)                    """Return whether the bbox has changed since init."""
863: (8)                    return self.mutatedx() or self.mutatedy()
864: (4)                def mutatedx(self):
865: (8)                    """Return whether the x-limits have changed since init."""
866: (8)                    return (self._points[0, 0] != self._points_orig[0, 0] or
867: (16)                           self._points[1, 0] != self._points_orig[1, 0])
868: (4)                def mutatedy(self):
869: (8)                    """Return whether the y-limits have changed since init."""
870: (8)                    return (self._points[0, 1] != self._points_orig[0, 1] or
871: (16)                           self._points[1, 1] != self._points_orig[1, 1])
872: (0)            class TransformedBbox(BboxBase):
873: (4)                """
874: (4)                A `Bbox` that is automatically transformed by a given
875: (4)                transform.  When either the child bounding box or transform
876: (4)                changes, the bounds of this bbox will update accordingly.
877: (4)                """
878: (4)                def __init__(self, bbox, transform, **kwargs):
879: (8)                    """
880: (8)                    Parameters
881: (8)                    ----------
882: (8)                    bbox : `Bbox`
883: (8)                    transform : `Transform`
884: (8)                    """
885: (8)                    _api.check_isinstance(BboxBase, bbox=bbox)
886: (8)                    _api.check_isinstance(Transform, transform=transform)
887: (8)                    if transform.input_dims != 2 or transform.output_dims != 2:
888: (12)                       raise ValueError(
889: (16)                           "The input and output dimensions of 'transform' must be 2")
890: (8)                    super().__init__(**kwargs)
891: (8)                    self._bbox = bbox
892: (8)                    self._transform = transform
893: (8)                    self.set_children(bbox, transform)
894: (8)                    self._points = None
895: (4)                __str__ = _make_str_method("_bbox", "_transform")
896: (4)                def get_points(self):
897: (8)                    if self._invalid:
898: (12)                       p = self._bbox.get_points()
899: (12)                       points = self._transform.transform(
900: (16)                           [[p[0, 0], p[0, 1]],
901: (17)                            [p[1, 0], p[0, 1]],
902: (17)                            [p[0, 0], p[1, 1]],
903: (17)                            [p[1, 0], p[1, 1]]])
904: (12)                       points = np.ma.filled(points, 0.0)
905: (12)                       xs = min(points[:, 0]), max(points[:, 0])
906: (12)                       if p[0, 0] > p[1, 0]:
907: (16)                           xs = xs[::-1]
908: (12)                       ys = min(points[:, 1]), max(points[:, 1])
909: (12)                       if p[0, 1] > p[1, 1]:
910: (16)                           ys = ys[::-1]
911: (12)                       self._points = np.array([
912: (16)                           [xs[0], ys[0]],
913: (16)                           [xs[1], ys[1]]
914: (12)                       ])
915: (12)                       self._invalid = 0
916: (8)                    return self._points
917: (4)                if DEBUG:
918: (8)                    _get_points = get_points
919: (8)                    def get_points(self):
920: (12)                       points = self._get_points()
921: (12)                       self._check(points)
922: (12)                       return points
923: (4)                def contains(self, x, y):
924: (8)                    return self._bbox.contains(*self._transform.inverted().transform((x,
y)))
925: (4)                def fully_contains(self, x, y):
```

```
926: (8)                    return
self._bbox.fully_contains(*self._transform.inverted().transform((x, y)))
927: (0)            class LockableBbox(BboxBase):
928: (4)                """
929: (4)                A `Bbox` where some elements may be locked at certain values.
930: (4)                When the child bounding box changes, the bounds of this bbox will update
931: (4)                accordingly with the exception of the locked elements.
932: (4)                """
933: (4)                def __init__(self, bbox, x0=None, y0=None, x1=None, y1=None, **kwargs):
934: (8)                    """
935: (8)                    Parameters
936: (8)                    ----------
937: (8)                    bbox : `Bbox`
938: (12)                       The child bounding box to wrap.
939: (8)                    x0 : float or None
940: (12)                       The locked value for x0, or None to leave unlocked.
941: (8)                    y0 : float or None
942: (12)                       The locked value for y0, or None to leave unlocked.
943: (8)                    x1 : float or None
944: (12)                       The locked value for x1, or None to leave unlocked.
945: (8)                    y1 : float or None
946: (12)                       The locked value for y1, or None to leave unlocked.
947: (8)                    """
948: (8)                    _api.check_isinstance(BboxBase, bbox=bbox)
949: (8)                    super().__init__(**kwargs)
950: (8)                    self._bbox = bbox
951: (8)                    self.set_children(bbox)
952: (8)                    self._points = None
953: (8)                    fp = [x0, y0, x1, y1]
954: (8)                    mask = [val is None for val in fp]
955: (8)                    self._locked_points = np.ma.array(fp, float, mask=mask).reshape((2,
2))
956: (4)                __str__ = _make_str_method("_bbox", "_locked_points")
957: (4)                def get_points(self):
958: (8)                    if self._invalid:
959: (12)                       points = self._bbox.get_points()
960: (12)                       self._points = np.where(self._locked_points.mask,
961: (36)                                               points,
962: (36)                                               self._locked_points)
963: (12)                       self._invalid = 0
964: (8)                    return self._points
965: (4)                if DEBUG:
966: (8)                    _get_points = get_points
967: (8)                    def get_points(self):
968: (12)                       points = self._get_points()
969: (12)                       self._check(points)
970: (12)                       return points
971: (4)                @property
972: (4)                def locked_x0(self):
973: (8)                    """
974: (8)                    float or None: The value used for the locked x0.
975: (8)                    """
976: (8)                    if self._locked_points.mask[0, 0]:
977: (12)                       return None
978: (8)                    else:
979: (12)                       return self._locked_points[0, 0]
980: (4)                @locked_x0.setter
981: (4)                def locked_x0(self, x0):
982: (8)                    self._locked_points.mask[0, 0] = x0 is None
983: (8)                    self._locked_points.data[0, 0] = x0
984: (8)                    self.invalidate()
985: (4)                @property
986: (4)                def locked_y0(self):
987: (8)                    """
988: (8)                    float or None: The value used for the locked y0.
989: (8)                    """
990: (8)                    if self._locked_points.mask[0, 1]:
991: (12)                       return None
992: (8)                    else:
```

```
(12)                        return self._locked_points[0, 1]
(4)                 @locked_y0.setter
(4)                 def locked_y0(self, y0):
(8)                     self._locked_points.mask[0, 1] = y0 is None
(8)                     self._locked_points.data[0, 1] = y0
(8)                     self.invalidate()
(4)                 @property
(4)                 def locked_x1(self):
(8)                     """
(8)                     float or None: The value used for the locked x1.
(8)                     """
(8)                     if self._locked_points.mask[1, 0]:
(12)                        return None
(8)                     else:
(12)                        return self._locked_points[1, 0]
(4)                 @locked_x1.setter
(4)                 def locked_x1(self, x1):
(8)                     self._locked_points.mask[1, 0] = x1 is None
(8)                     self._locked_points.data[1, 0] = x1
(8)                     self.invalidate()
(4)                 @property
(4)                 def locked_y1(self):
(8)                     """
(8)                     float or None: The value used for the locked y1.
(8)                     """
(8)                     if self._locked_points.mask[1, 1]:
(12)                        return None
(8)                     else:
(12)                        return self._locked_points[1, 1]
(4)                 @locked_y1.setter
(4)                 def locked_y1(self, y1):
(8)                     self._locked_points.mask[1, 1] = y1 is None
(8)                     self._locked_points.data[1, 1] = y1
(8)                     self.invalidate()
(0)             class Transform(TransformNode):
(4)                 """
(4)                 The base class of all `TransformNode` instances that
(4)                 actually perform a transformation.
(4)                 All non-affine transformations should be subclasses of this class.
(4)                 New affine transformations should be subclasses of `Affine2D`.
(4)                 Subclasses of this class should override the following members (at
(4)                 minimum):
(4)                 - :attr:`input_dims`
(4)                 - :attr:`output_dims`
(4)                 - :meth:`transform`
(4)                 - :meth:`inverted` (if an inverse exists)
(4)                 The following attributes may be overridden if the default is unsuitable:
(4)                 - :attr:`is_separable` (defaults to True for 1D -> 1D transforms, False
(6)                   otherwise)
(4)                 - :attr:`has_inverse` (defaults to True if :meth:`inverted` is overridden,
(6)                   False otherwise)
(4)                 If the transform needs to do something non-standard with
(4)                 `matplotlib.path.Path` objects, such as adding curves
(4)                 where there were once line segments, it should override:
(4)                 - :meth:`transform_path`
(4)                 """
(4)                 input_dims = None
(4)                 """
(4)                 The number of input dimensions of this transform.
(4)                 Must be overridden (with integers) in the subclass.
(4)                 """
(4)                 output_dims = None
(4)                 """
(4)                 The number of output dimensions of this transform.
(4)                 Must be overridden (with integers) in the subclass.
(4)                 """
(4)                 is_separable = False
(4)                 """True if this transform is separable in the x- and y- dimensions."""
(4)                 has_inverse = False
```

```
(4)                """True if this transform has a corresponding inverse transform."""
(4)                def __init_subclass__(cls):
(8)                    if (sum("is_separable" in vars(parent) for parent in cls.__mro__) == 1
(16)                           and cls.input_dims == cls.output_dims == 1):
(12)                       cls.is_separable = True
(8)                    if (sum("has_inverse" in vars(parent) for parent in cls.__mro__) == 1
(16)                           and hasattr(cls, "inverted")
(16)                           and cls.inverted is not Transform.inverted):
(12)                       cls.has_inverse = True
(4)                def __add__(self, other):
(8)                    """
(8)                    Compose two transforms together so that *self* is followed by *other*.
(8)                    ``A + B`` returns a transform ``C`` so that
(8)                    ``C.transform(x) == B.transform(A.transform(x))``.
(8)                    """
(8)                    return (composite_transform_factory(self, other)
(16)                           if isinstance(other, Transform) else
(16)                           NotImplemented)
(4)                def _iter_break_from_left_to_right(self):
(8)                    """
(8)                    Return an iterator breaking down this transform stack from left to
(8)                    right recursively. If self == ((A, N), A) then the result will be an
(8)                    iterator which yields I : ((A, N), A), followed by A : (N, A),
(8)                    followed by (A, N) : (A), but not ((A, N), A) : I.
(8)                    This is equivalent to flattening the stack then yielding
(8)                    ``flat_stack[:i], flat_stack[i:]`` where i=0..(n-1).
(8)                    """
(8)                    yield IdentityTransform(), self
(4)                @property
(4)                def depth(self):
(8)                    """
(8)                    Return the number of transforms which have been chained
(8)                    together to form this Transform instance.
(8)                    .. note::
(12)                       For the special case of a Composite transform, the maximum depth
(12)                       of the two is returned.
(8)                    """
(8)                    return 1
(4)                def contains_branch(self, other):
(8)                    """
(8)                    Return whether the given transform is a sub-tree of this transform.
(8)                    This routine uses transform equality to identify sub-trees, therefore
(8)                    in many situations it is object id which will be used.
(8)                    For the case where the given transform represents the whole
(8)                    of this transform, returns True.
(8)                    """
(8)                    if self.depth < other.depth:
(12)                       return False
(8)                    for _, sub_tree in self._iter_break_from_left_to_right():
(12)                       if sub_tree == other:
(16)                           return True
(8)                    return False
(4)                def contains_branch_seperately(self, other_transform):
(8)                    """
(8)                    Return whether the given branch is a sub-tree of this transform on
(8)                    each separate dimension.
(8)                    A common use for this method is to identify if a transform is a
blended
(8)                    transform containing an Axes' data transform. e.g.::
(12)                       x_isdata, y_isdata =
trans.contains_branch_seperately(ax.transData)
(8)                    """
(8)                    if self.output_dims != 2:
(12)                       raise ValueError('contains_branch_seperately only supports '
(29)                                        'transforms with 2 output dimensions')
(8)                    return (self.contains_branch(other_transform), ) * 2
(4)                def __sub__(self, other):
(8)                    """
(8)                    Compose *self* with the inverse of *other*, cancelling identical terms
```

```
(8)                     if any::
(12)                        A - B == A + B.inverted()
(12)                        (A' + B) - B == A'
(12)                        A - (A + B') == B'.inverted() == B'^-1
(8)                     Cancellation (rather than naively returning ``A + B.inverted()``) is
(8)                     important for multiple reasons:
(8)                     - It avoids floating-point inaccuracies when computing the inverse of
(10)                      B: ``B - B`` is guaranteed to cancel out exactly (resulting in the
(10)                      identity transform), whereas ``B + B.inverted()`` may differ by a
(10)                      small epsilon.
(8)                     - ``B.inverted()`` always returns a frozen transform: if one computes
(10)                      ``A + B + B.inverted()`` and later mutates ``B``, then
(10)                      ``B.inverted()`` won't be updated and the last two terms won't
cancel
(10)                      out anymore; on the other hand, ``A + B - B`` will always be equal
to
(10)                      ``A`` even if ``B`` is mutated.
(8)                     """
(8)                     if not isinstance(other, Transform):
(12)                        return NotImplemented
(8)                     for remainder, sub_tree in self._iter_break_from_left_to_right():
(12)                        if sub_tree == other:
(16)                            return remainder
(8)                     for remainder, sub_tree in other._iter_break_from_left_to_right():
(12)                        if sub_tree == self:
(16)                            if not remainder.has_inverse:
(20)                                raise ValueError(
(24)                                    "The shortcut cannot be computed since 'other' "
(24)                                    "includes a non-invertible component")
(16)                            return remainder.inverted()
(8)                     if other.has_inverse:
(12)                        return self + other.inverted()
(8)                     else:
(12)                        raise ValueError('It is not possible to compute transA - transB '
(29)                                         'since transB cannot be inverted and there is no
'
(29)                                         'shortcut possible.')
(4)                 def __array__(self, *args, **kwargs):
(8)                     """Array interface to get at this Transform's affine matrix."""
(8)                     return self.get_affine().get_matrix()
(4)                 def transform(self, values):
(8)                     """
(8)                     Apply this transformation on the given array of *values*.
(8)                     Parameters
(8)                     ----------
(8)                     values : array-like
(12)                        The input values as an array of length :attr:`input_dims` or
(12)                        shape (N, :attr:`input_dims`).
(8)                     Returns
(8)                     -------
(8)                     array
(12)                        The output values as an array of length :attr:`output_dims` or
(12)                        shape (N, :attr:`output_dims`), depending on the input.
(8)                     """
(8)                     values = np.asanyarray(values)
(8)                     ndim = values.ndim
(8)                     values = values.reshape((-1, self.input_dims))
(8)                     res = self.transform_affine(self.transform_non_affine(values))
(8)                     if ndim == 0:
(12)                        assert not np.ma.is_masked(res)  # just to be on the safe side
(12)                        return res[0, 0]
(8)                     if ndim == 1:
(12)                        return res.reshape(-1)
(8)                     elif ndim == 2:
(12)                        return res
(8)                     raise ValueError(
(12)                        "Input values must have shape (N, {dims}) or ({dims},)"
(12)                        .format(dims=self.input_dims))
(4)                 def transform_affine(self, values):
```

```
(8)                 """
(8)                 Apply only the affine part of this transformation on the
(8)                 given array of values.
(8)                 ``transform(values)`` is always equivalent to
(8)                 ``transform_affine(transform_non_affine(values))``.
(8)                 In non-affine transformations, this is generally a no-op.  In
(8)                 affine transformations, this is equivalent to
(8)                 ``transform(values)``.
(8)                 Parameters
(8)                 ----------
(8)                 values : array
(12)                    The input values as an array of length :attr:`input_dims` or
(12)                    shape (N, :attr:`input_dims`).
(8)                 Returns
(8)                 -------
(8)                 array
(12)                    The output values as an array of length :attr:`output_dims` or
(12)                    shape (N, :attr:`output_dims`), depending on the input.
(8)                 """
(8)                 return self.get_affine().transform(values)
(4)             def transform_non_affine(self, values):
(8)                 """
(8)                 Apply only the non-affine part of this transformation.
(8)                 ``transform(values)`` is always equivalent to
(8)                 ``transform_affine(transform_non_affine(values))``.
(8)                 In non-affine transformations, this is generally equivalent to
(8)                 ``transform(values)``.  In affine transformations, this is
(8)                 always a no-op.
(8)                 Parameters
(8)                 ----------
(8)                 values : array
(12)                    The input values as an array of length :attr:`input_dims` or
(12)                    shape (N, :attr:`input_dims`).
(8)                 Returns
(8)                 -------
(8)                 array
(12)                    The output values as an array of length :attr:`output_dims` or
(12)                    shape (N, :attr:`output_dims`), depending on the input.
(8)                 """
(8)                 return values
(4)             def transform_bbox(self, bbox):
(8)                 """
(8)                 Transform the given bounding box.
(8)                 For smarter transforms including caching (a common requirement in
(8)                 Matplotlib), see `TransformedBbox`.
(8)                 """
(8)                 return Bbox(self.transform(bbox.get_points()))
(4)             def get_affine(self):
(8)                 """Get the affine part of this transform."""
(8)                 return IdentityTransform()
(4)             def get_matrix(self):
(8)                 """Get the matrix for the affine part of this transform."""
(8)                 return self.get_affine().get_matrix()
(4)             def transform_point(self, point):
(8)                 """
(8)                 Return a transformed point.
(8)                 This function is only kept for backcompatibility; the more general
(8)                 `.transform` method is capable of transforming both a list of points
(8)                 and a single point.
(8)                 The point is given as a sequence of length :attr:`input_dims`.
(8)                 The transformed point is returned as a sequence of length
(8)                 :attr:`output_dims`.
(8)                 """
(8)                 if len(point) != self.input_dims:
(12)                    raise ValueError("The length of 'point' must be
'self.input_dims'")
(8)                 return self.transform(point)
(4)             def transform_path(self, path):
(8)                 """
```

```
(8)                     Apply the transform to `.Path` *path*, returning a new `.Path`.
(8)                     In some cases, this transform may insert curves into the path
(8)                     that began as line segments.
(8)                     """
(8)                     return
self.transform_path_affine(self.transform_path_non_affine(path))
(4)                 def transform_path_affine(self, path):
(8)                     """
(8)                     Apply the affine part of this transform to `.Path` *path*, returning a
(8)                     new `.Path`.
(8)                     ``transform_path(path)`` is equivalent to
(8)                     ``transform_path_affine(transform_path_non_affine(values))``.
(8)                     """
(8)                     return self.get_affine().transform_path_affine(path)
(4)                 def transform_path_non_affine(self, path):
(8)                     """
(8)                     Apply the non-affine part of this transform to `.Path` *path*,
(8)                     returning a new `.Path`.
(8)                     ``transform_path(path)`` is equivalent to
(8)                     ``transform_path_affine(transform_path_non_affine(values))``.
(8)                     """
(8)                     x = self.transform_non_affine(path.vertices)
(8)                     return Path._fast_from_codes_and_verts(x, path.codes, path)
(4)                 def transform_angles(self, angles, pts, radians=False, pushoff=1e-5):
(8)                     """
(8)                     Transform a set of angles anchored at specific locations.
(8)                     Parameters
(8)                     ----------
(8)                     angles : (N,) array-like
(12)                        The angles to transform.
(8)                     pts : (N, 2) array-like
(12)                        The points where the angles are anchored.
(8)                     radians : bool, default: False
(12)                        Whether *angles* are radians or degrees.
(8)                     pushoff : float
(12)                        For each point in *pts* and angle in *angles*, the transformed
(12)                        angle is computed by transforming a segment of length *pushoff*
(12)                        starting at that point and making that angle relative to the
(12)                        horizontal axis, and measuring the angle between the horizontal
(12)                        axis and the transformed segment.
(8)                     Returns
(8)                     -------
(8)                     (N,) array
(8)                     """
(8)                     if self.input_dims != 2 or self.output_dims != 2:
(12)                        raise NotImplementedError('Only defined in 2D')
(8)                     angles = np.asarray(angles)
(8)                     pts = np.asarray(pts)
(8)                     _api.check_shape((None, 2), pts=pts)
(8)                     _api.check_shape((None,), angles=angles)
(8)                     if len(angles) != len(pts):
(12)                        raise ValueError("There must be as many 'angles' as 'pts'")
(8)                     if not radians:
(12)                        angles = np.deg2rad(angles)
(8)                     pts2 = pts + pushoff * np.column_stack([np.cos(angles),
(48)                                                            np.sin(angles)])
(8)                     tpts = self.transform(pts)
(8)                     tpts2 = self.transform(pts2)
(8)                     d = tpts2 - tpts
(8)                     a = np.arctan2(d[:, 1], d[:, 0])
(8)                     if not radians:
(12)                        a = np.rad2deg(a)
(8)                     return a
(4)                 def inverted(self):
(8)                     """
(8)                     Return the corresponding inverse transformation.
(8)                     It holds ``x == self.inverted().transform(self.transform(x))``.
(8)                     The return value of this method should be treated as
(8)                     temporary.  An update to *self* does not cause a corresponding
```

```
1331: (8)                update to its inverted copy.
1332: (8)                """
1333: (8)                raise NotImplementedError()
1334: (0)            class TransformWrapper(Transform):
1335: (4)                """
1336: (4)                A helper class that holds a single child transform and acts
1337: (4)                equivalently to it.
1338: (4)                This is useful if a node of the transform tree must be replaced at
1339: (4)                run time with a transform of a different type.  This class allows
1340: (4)                that replacement to correctly trigger invalidation.
1341: (4)                `TransformWrapper` instances must have the same input and output
dimensions
1342: (4)                during their entire lifetime, so the child transform may only be replaced
1343: (4)                with another child transform of the same dimensions.
1344: (4)                """
1345: (4)                pass_through = True
1346: (4)                def __init__(self, child):
1347: (8)                    """
1348: (8)                    *child*: A `Transform` instance.  This child may later
1349: (8)                    be replaced with :meth:`set`.
1350: (8)                    """
1351: (8)                    _api.check_isinstance(Transform, child=child)
1352: (8)                    super().__init__()
1353: (8)                    self.set(child)
1354: (4)                def __eq__(self, other):
1355: (8)                    return self._child.__eq__(other)
1356: (4)                __str__ = _make_str_method("_child")
1357: (4)                def frozen(self):
1358: (8)                    return self._child.frozen()
1359: (4)                def set(self, child):
1360: (8)                    """
1361: (8)                    Replace the current child of this transform with another one.
1362: (8)                    The new child must have the same number of input and output
1363: (8)                    dimensions as the current child.
1364: (8)                    """
1365: (8)                    if hasattr(self, "_child"):  # Absent during init.
1366: (12)                       self.invalidate()
1367: (12)                       new_dims = (child.input_dims, child.output_dims)
1368: (12)                       old_dims = (self._child.input_dims, self._child.output_dims)
1369: (12)                       if new_dims != old_dims:
1370: (16)                           raise ValueError(
1371: (20)                               f"The input and output dims of the new child {new_dims} "
1372: (20)                               f"do not match those of current child {old_dims}")
1373: (12)                       self._child._parents.pop(id(self), None)
1374: (8)                    self._child = child
1375: (8)                    self.set_children(child)
1376: (8)                    self.transform = child.transform
1377: (8)                    self.transform_affine = child.transform_affine
1378: (8)                    self.transform_non_affine = child.transform_non_affine
1379: (8)                    self.transform_path = child.transform_path
1380: (8)                    self.transform_path_affine = child.transform_path_affine
1381: (8)                    self.transform_path_non_affine = child.transform_path_non_affine
1382: (8)                    self.get_affine = child.get_affine
1383: (8)                    self.inverted = child.inverted
1384: (8)                    self.get_matrix = child.get_matrix
1385: (8)                    self._invalid = 0
1386: (8)                    self.invalidate()
1387: (8)                    self._invalid = 0
1388: (4)                input_dims = property(lambda self: self._child.input_dims)
1389: (4)                output_dims = property(lambda self: self._child.output_dims)
1390: (4)                is_affine = property(lambda self: self._child.is_affine)
1391: (4)                is_separable = property(lambda self: self._child.is_separable)
1392: (4)                has_inverse = property(lambda self: self._child.has_inverse)
1393: (0)            class AffineBase(Transform):
1394: (4)                """
1395: (4)                The base class of all affine transformations of any number of dimensions.
1396: (4)                """
1397: (4)                is_affine = True
1398: (4)                def __init__(self, *args, **kwargs):
```

```
1399: (8)                   super().__init__(*args, **kwargs)
1400: (8)                   self._inverted = None
1401: (4)               def __array__(self, *args, **kwargs):
1402: (8)                   return self.get_matrix()
1403: (4)               def __eq__(self, other):
1404: (8)                   if getattr(other, "is_affine", False) and hasattr(other,
"get_matrix"):
1405: (12)                      return (self.get_matrix() == other.get_matrix()).all()
1406: (8)                   return NotImplemented
1407: (4)               def transform(self, values):
1408: (8)                   return self.transform_affine(values)
1409: (4)               def transform_affine(self, values):
1410: (8)                   raise NotImplementedError('Affine subclasses should override this '
1411: (34)                                            'method.')
1412: (4)               @_api.rename_parameter("3.8", "points", "values")
1413: (4)               def transform_non_affine(self, values):
1414: (8)                   return values
1415: (4)               def transform_path(self, path):
1416: (8)                   return self.transform_path_affine(path)
1417: (4)               def transform_path_affine(self, path):
1418: (8)                   return Path(self.transform_affine(path.vertices),
1419: (20)                              path.codes, path._interpolation_steps)
1420: (4)               def transform_path_non_affine(self, path):
1421: (8)                   return path
1422: (4)               def get_affine(self):
1423: (8)                   return self
1424: (0)           class Affine2DBase(AffineBase):
1425: (4)               """
1426: (4)               The base class of all 2D affine transformations.
1427: (4)               2D affine transformations are performed using a 3x3 numpy array::
1428: (8)                   a c e
1429: (8)                   b d f
1430: (8)                   0 0 1
1431: (4)               This class provides the read-only interface.  For a mutable 2D
1432: (4)               affine transformation, use `Affine2D`.
1433: (4)               Subclasses of this class will generally only need to override a
1434: (4)               constructor and `~.Transform.get_matrix` that generates a custom 3x3
matrix.
1435: (4)               """
1436: (4)               input_dims = 2
1437: (4)               output_dims = 2
1438: (4)               def frozen(self):
1439: (8)                   return Affine2D(self.get_matrix().copy())
1440: (4)               @property
1441: (4)               def is_separable(self):
1442: (8)                   mtx = self.get_matrix()
1443: (8)                   return mtx[0, 1] == mtx[1, 0] == 0.0
1444: (4)               def to_values(self):
1445: (8)                   """
1446: (8)                   Return the values of the matrix as an ``(a, b, c, d, e, f)`` tuple.
1447: (8)                   """
1448: (8)                   mtx = self.get_matrix()
1449: (8)                   return tuple(mtx[:2].swapaxes(0, 1).flat)
1450: (4)               @_api.rename_parameter("3.8", "points", "values")
1451: (4)               def transform_affine(self, values):
1452: (8)                   mtx = self.get_matrix()
1453: (8)                   if isinstance(values, np.ma.MaskedArray):
1454: (12)                      tpoints = affine_transform(values.data, mtx)
1455: (12)                      return np.ma.MaskedArray(tpoints, mask=np.ma.getmask(values))
1456: (8)                   return affine_transform(values, mtx)
1457: (4)               if DEBUG:
1458: (8)                   _transform_affine = transform_affine
1459: (8)                   @_api.rename_parameter("3.8", "points", "values")
1460: (8)                   def transform_affine(self, values):
1461: (12)                      if not isinstance(values, np.ndarray):
1462: (16)                          _api.warn_external(
1463: (20)                              f'A non-numpy array of type {type(values)} was passed in '
1464: (20)                              f'for transformation, which results in poor performance.')
1465: (12)                      return self._transform_affine(values)
```

```
(4)                 def inverted(self):
(8)                     if self._inverted is None or self._invalid:
(12)                        mtx = self.get_matrix()
(12)                        shorthand_name = None
(12)                        if self._shorthand_name:
(16)                            shorthand_name = '(%s)-1' % self._shorthand_name
(12)                        self._inverted = Affine2D(inv(mtx), shorthand_name=shorthand_name)
(12)                        self._invalid = 0
(8)                     return self._inverted
(0)             class Affine2D(Affine2DBase):
(4)                 """
(4)                 A mutable 2D affine transformation.
(4)                 """
(4)                 def __init__(self, matrix=None, **kwargs):
(8)                     """
(8)                     Initialize an Affine transform from a 3x3 numpy float array::
(10)                      a c e
(10)                      b d f
(10)                      0 0 1
(8)                     If *matrix* is None, initialize with the identity transform.
(8)                     """
(8)                     super().__init__(**kwargs)
(8)                     if matrix is None:
(12)                        matrix = IdentityTransform._mtx
(8)                     self._mtx = matrix.copy()
(8)                     self._invalid = 0
(4)                 _base_str = _make_str_method("_mtx")
(4)                 def __str__(self):
(8)                     return (self._base_str()
(16)                            if (self._mtx != np.diag(np.diag(self._mtx))).any()
(16)                            else f"Affine2D().scale({self._mtx[0, 0]}, {self._mtx[1, 1]})"
(16)                            if self._mtx[0, 0] != self._mtx[1, 1]
(16)                            else f"Affine2D().scale({self._mtx[0, 0]})")
(4)                 @staticmethod
(4)                 def from_values(a, b, c, d, e, f):
(8)                     """
(8)                     Create a new Affine2D instance from the given values::
(10)                      a c e
(10)                      b d f
(10)                      0 0 1
(8)                     .
(8)                     """
(8)                     return Affine2D(
(12)                        np.array([a, c, e, b, d, f, 0.0, 0.0, 1.0], float).reshape((3,
3)))
(4)                 def get_matrix(self):
(8)                     """
(8)                     Get the underlying transformation matrix as a 3x3 array::
(10)                      a c e
(10)                      b d f
(10)                      0 0 1
(8)                     .
(8)                     """
(8)                     if self._invalid:
(12)                        self._inverted = None
(12)                        self._invalid = 0
(8)                     return self._mtx
(4)                 def set_matrix(self, mtx):
(8)                     """
(8)                     Set the underlying transformation matrix from a 3x3 array::
(10)                      a c e
(10)                      b d f
(10)                      0 0 1
(8)                     .
(8)                     """
(8)                     self._mtx = mtx
(8)                     self.invalidate()
(4)                 def set(self, other):
(8)                     """
```

```
(8)                 Set this transformation from the frozen copy of another
(8)                 `Affine2DBase` object.
(8)                 """
(8)                 _api.check_isinstance(Affine2DBase, other=other)
(8)                 self._mtx = other.get_matrix()
(8)                 self.invalidate()
(4)             def clear(self):
(8)                 """
(8)                 Reset the underlying matrix to the identity transform.
(8)                 """
(8)                 self._mtx = IdentityTransform._mtx.copy()
(8)                 self.invalidate()
(8)                 return self
(4)             def rotate(self, theta):
(8)                 """
(8)                 Add a rotation (in radians) to this transform in place.
(8)                 Returns *self*, so this method can easily be chained with more
(8)                 calls to :meth:`rotate`, :meth:`rotate_deg`, :meth:`translate`
(8)                 and :meth:`scale`.
(8)                 """
(8)                 a = math.cos(theta)
(8)                 b = math.sin(theta)
(8)                 mtx = self._mtx
(8)                 (xx, xy, x0), (yx, yy, y0), _ = mtx.tolist()
(8)                 mtx[0, 0] = a * xx - b * yx
(8)                 mtx[0, 1] = a * xy - b * yy
(8)                 mtx[0, 2] = a * x0 - b * y0
(8)                 mtx[1, 0] = b * xx + a * yx
(8)                 mtx[1, 1] = b * xy + a * yy
(8)                 mtx[1, 2] = b * x0 + a * y0
(8)                 self.invalidate()
(8)                 return self
(4)             def rotate_deg(self, degrees):
(8)                 """
(8)                 Add a rotation (in degrees) to this transform in place.
(8)                 Returns *self*, so this method can easily be chained with more
(8)                 calls to :meth:`rotate`, :meth:`rotate_deg`, :meth:`translate`
(8)                 and :meth:`scale`.
(8)                 """
(8)                 return self.rotate(math.radians(degrees))
(4)             def rotate_around(self, x, y, theta):
(8)                 """
(8)                 Add a rotation (in radians) around the point (x, y) in place.
(8)                 Returns *self*, so this method can easily be chained with more
(8)                 calls to :meth:`rotate`, :meth:`rotate_deg`, :meth:`translate`
(8)                 and :meth:`scale`.
(8)                 """
(8)                 return self.translate(-x, -y).rotate(theta).translate(x, y)
(4)             def rotate_deg_around(self, x, y, degrees):
(8)                 """
(8)                 Add a rotation (in degrees) around the point (x, y) in place.
(8)                 Returns *self*, so this method can easily be chained with more
(8)                 calls to :meth:`rotate`, :meth:`rotate_deg`, :meth:`translate`
(8)                 and :meth:`scale`.
(8)                 """
(8)                 x, y = float(x), float(y)
(8)                 return self.translate(-x, -y).rotate_deg(degrees).translate(x, y)
(4)             def translate(self, tx, ty):
(8)                 """
(8)                 Add a translation in place.
(8)                 Returns *self*, so this method can easily be chained with more
(8)                 calls to :meth:`rotate`, :meth:`rotate_deg`, :meth:`translate`
(8)                 and :meth:`scale`.
(8)                 """
(8)                 self._mtx[0, 2] += tx
(8)                 self._mtx[1, 2] += ty
(8)                 self.invalidate()
(8)                 return self
(4)             def scale(self, sx, sy=None):
```

```
(8)                 """
(8)                 Add a scale in place.
(8)                 If *sy* is None, the same scale is applied in both the *x*- and
(8)                 *y*-directions.
(8)                 Returns *self*, so this method can easily be chained with more
(8)                 calls to :meth:`rotate`, :meth:`rotate_deg`, :meth:`translate`
(8)                 and :meth:`scale`.
(8)                 """
(8)                 if sy is None:
(12)                    sy = sx
(8)                 self._mtx[0, 0] *= sx
(8)                 self._mtx[0, 1] *= sx
(8)                 self._mtx[0, 2] *= sx
(8)                 self._mtx[1, 0] *= sy
(8)                 self._mtx[1, 1] *= sy
(8)                 self._mtx[1, 2] *= sy
(8)                 self.invalidate()
(8)                 return self
(4)             def skew(self, xShear, yShear):
(8)                 """
(8)                 Add a skew in place.
(8)                 *xShear* and *yShear* are the shear angles along the *x*- and
(8)                 *y*-axes, respectively, in radians.
(8)                 Returns *self*, so this method can easily be chained with more
(8)                 calls to :meth:`rotate`, :meth:`rotate_deg`, :meth:`translate`
(8)                 and :meth:`scale`.
(8)                 """
(8)                 rx = math.tan(xShear)
(8)                 ry = math.tan(yShear)
(8)                 mtx = self._mtx
(8)                 (xx, xy, x0), (yx, yy, y0), _ = mtx.tolist()
(8)                 mtx[0, 0] += rx * yx
(8)                 mtx[0, 1] += rx * yy
(8)                 mtx[0, 2] += rx * y0
(8)                 mtx[1, 0] += ry * xx
(8)                 mtx[1, 1] += ry * xy
(8)                 mtx[1, 2] += ry * x0
(8)                 self.invalidate()
(8)                 return self
(4)             def skew_deg(self, xShear, yShear):
(8)                 """
(8)                 Add a skew in place.
(8)                 *xShear* and *yShear* are the shear angles along the *x*- and
(8)                 *y*-axes, respectively, in degrees.
(8)                 Returns *self*, so this method can easily be chained with more
(8)                 calls to :meth:`rotate`, :meth:`rotate_deg`, :meth:`translate`
(8)                 and :meth:`scale`.
(8)                 """
(8)                 return self.skew(math.radians(xShear), math.radians(yShear))
(0)         class IdentityTransform(Affine2DBase):
(4)             """
(4)             A special class that does one thing, the identity transform, in a
(4)             fast way.
(4)             """
(4)             _mtx = np.identity(3)
(4)             def frozen(self):
(8)                 return self
(4)             __str__ = _make_str_method()
(4)             def get_matrix(self):
(8)                 return self._mtx
(4)             @_api.rename_parameter("3.8", "points", "values")
(4)             def transform(self, values):
(8)                 return np.asanyarray(values)
(4)             @_api.rename_parameter("3.8", "points", "values")
(4)             def transform_affine(self, values):
(8)                 return np.asanyarray(values)
(4)             @_api.rename_parameter("3.8", "points", "values")
(4)             def transform_non_affine(self, values):
(8)                 return np.asanyarray(values)
```

```
(4)         def transform_path(self, path):
(8)             return path
(4)         def transform_path_affine(self, path):
(8)             return path
(4)         def transform_path_non_affine(self, path):
(8)             return path
(4)         def get_affine(self):
(8)             return self
(4)         def inverted(self):
(8)             return self
(0)     class _BlendedMixin:
(4)         """Common methods for `BlendedGenericTransform` and `BlendedAffine2D`."""
(4)         def __eq__(self, other):
(8)             if isinstance(other, (BlendedAffine2D, BlendedGenericTransform)):
(12)                return (self._x == other._x) and (self._y == other._y)
(8)             elif self._x == self._y:
(12)                return self._x == other
(8)             else:
(12)                return NotImplemented
(4)         def contains_branch_seperately(self, transform):
(8)             return (self._x.contains_branch(transform),
(16)                     self._y.contains_branch(transform))
(4)         __str__ = _make_str_method("_x", "_y")
(0)     class BlendedGenericTransform(_BlendedMixin, Transform):
(4)         """
(4)         A "blended" transform uses one transform for the *x*-direction, and
(4)         another transform for the *y*-direction.
(4)         This "generic" version can handle any given child transform in the
(4)         *x*- and *y*-directions.
(4)         """
(4)         input_dims = 2
(4)         output_dims = 2
(4)         is_separable = True
(4)         pass_through = True
(4)         def __init__(self, x_transform, y_transform, **kwargs):
(8)             """
(8)             Create a new "blended" transform using *x_transform* to transform the
(8)             *x*-axis and *y_transform* to transform the *y*-axis.
(8)             You will generally not call this constructor directly but use the
(8)             `blended_transform_factory` function instead, which can determine
(8)             automatically which kind of blended transform to create.
(8)             """
(8)             Transform.__init__(self, **kwargs)
(8)             self._x = x_transform
(8)             self._y = y_transform
(8)             self.set_children(x_transform, y_transform)
(8)             self._affine = None
(4)         @property
(4)         def depth(self):
(8)             return max(self._x.depth, self._y.depth)
(4)         def contains_branch(self, other):
(8)             return False
(4)         is_affine = property(lambda self: self._x.is_affine and self._y.is_affine)
(4)         has_inverse = property(
(8)             lambda self: self._x.has_inverse and self._y.has_inverse)
(4)         def frozen(self):
(8)             return blended_transform_factory(self._x.frozen(), self._y.frozen())
(4)         @_api.rename_parameter("3.8", "points", "values")
(4)         def transform_non_affine(self, values):
(8)             if self._x.is_affine and self._y.is_affine:
(12)                return values
(8)             x = self._x
(8)             y = self._y
(8)             if x == y and x.input_dims == 2:
(12)                return x.transform_non_affine(values)
(8)             if x.input_dims == 2:
(12)                x_points = x.transform_non_affine(values)[:, 0:1]
(8)             else:
(12)                x_points = x.transform_non_affine(values[:, 0])
```

```
1741: (12)                    x_points = x_points.reshape((len(x_points), 1))
1742: (8)                 if y.input_dims == 2:
1743: (12)                    y_points = y.transform_non_affine(values)[:, 1:]
1744: (8)                 else:
1745: (12)                    y_points = y.transform_non_affine(values[:, 1])
1746: (12)                    y_points = y_points.reshape((len(y_points), 1))
1747: (8)                 if (isinstance(x_points, np.ma.MaskedArray) or
1748: (16)                        isinstance(y_points, np.ma.MaskedArray)):
1749: (12)                    return np.ma.concatenate((x_points, y_points), 1)
1750: (8)                 else:
1751: (12)                    return np.concatenate((x_points, y_points), 1)
1752: (4)             def inverted(self):
1753: (8)                 return BlendedGenericTransform(self._x.inverted(), self._y.inverted())
1754: (4)             def get_affine(self):
1755: (8)                 if self._invalid or self._affine is None:
1756: (12)                    if self._x == self._y:
1757: (16)                        self._affine = self._x.get_affine()
1758: (12)                    else:
1759: (16)                        x_mtx = self._x.get_affine().get_matrix()
1760: (16)                        y_mtx = self._y.get_affine().get_matrix()
1761: (16)                        mtx = np.array([x_mtx[0], y_mtx[1], [0.0, 0.0, 1.0]])
1762: (16)                        self._affine = Affine2D(mtx)
1763: (12)                    self._invalid = 0
1764: (8)                 return self._affine
1765: (0)         class BlendedAffine2D(_BlendedMixin, Affine2DBase):
1766: (4)             """
1767: (4)             A "blended" transform uses one transform for the *x*-direction, and
1768: (4)             another transform for the *y*-direction.
1769: (4)             This version is an optimization for the case where both child
1770: (4)             transforms are of type `Affine2DBase`.
1771: (4)             """
1772: (4)             is_separable = True
1773: (4)             def __init__(self, x_transform, y_transform, **kwargs):
1774: (8)                 """
1775: (8)                 Create a new "blended" transform using *x_transform* to transform the
1776: (8)                 *x*-axis and *y_transform* to transform the *y*-axis.
1777: (8)                 Both *x_transform* and *y_transform* must be 2D affine transforms.
1778: (8)                 You will generally not call this constructor directly but use the
1779: (8)                 `blended_transform_factory` function instead, which can determine
1780: (8)                 automatically which kind of blended transform to create.
1781: (8)                 """
1782: (8)                 is_affine = x_transform.is_affine and y_transform.is_affine
1783: (8)                 is_separable = x_transform.is_separable and y_transform.is_separable
1784: (8)                 is_correct = is_affine and is_separable
1785: (8)                 if not is_correct:
1786: (12)                    raise ValueError("Both *x_transform* and *y_transform* must be 2D
"
1787: (29)                                         "affine transforms")
1788: (8)                 Transform.__init__(self, **kwargs)
1789: (8)                 self._x = x_transform
1790: (8)                 self._y = y_transform
1791: (8)                 self.set_children(x_transform, y_transform)
1792: (8)                 Affine2DBase.__init__(self)
1793: (8)                 self._mtx = None
1794: (4)             def get_matrix(self):
1795: (8)                 if self._invalid:
1796: (12)                    if self._x == self._y:
1797: (16)                        self._mtx = self._x.get_matrix()
1798: (12)                    else:
1799: (16)                        x_mtx = self._x.get_matrix()
1800: (16)                        y_mtx = self._y.get_matrix()
1801: (16)                        self._mtx = np.array([x_mtx[0], y_mtx[1], [0.0, 0.0, 1.0]])
1802: (12)                    self._inverted = None
1803: (12)                    self._invalid = 0
1804: (8)                 return self._mtx
1805: (0)         def blended_transform_factory(x_transform, y_transform):
1806: (4)             """
1807: (4)             Create a new "blended" transform using *x_transform* to transform
1808: (4)             the *x*-axis and *y_transform* to transform the *y*-axis.
```

```
1809: (4)               A faster version of the blended transform is returned for the case
1810: (4)               where both child transforms are affine.
1811: (4)               """
1812: (4)               if (isinstance(x_transform, Affine2DBase) and
1813: (12)                      isinstance(y_transform, Affine2DBase)):
1814: (8)                   return BlendedAffine2D(x_transform, y_transform)
1815: (4)               return BlendedGenericTransform(x_transform, y_transform)
1816: (0)           class CompositeGenericTransform(Transform):
1817: (4)               """
1818: (4)               A composite transform formed by applying transform *a* then
1819: (4)               transform *b*.
1820: (4)               This "generic" version can handle any two arbitrary
1821: (4)               transformations.
1822: (4)               """
1823: (4)               pass_through = True
1824: (4)               def __init__(self, a, b, **kwargs):
1825: (8)                   """
1826: (8)                   Create a new composite transform that is the result of
1827: (8)                   applying transform *a* then transform *b*.
1828: (8)                   You will generally not call this constructor directly but write ``a +
1829: (8)                   b`` instead, which will automatically choose the best kind of
composite
1830: (8)                   transform instance to create.
1831: (8)                   """
1832: (8)                   if a.output_dims != b.input_dims:
1833: (12)                      raise ValueError("The output dimension of 'a' must be equal to "
1834: (29)                                       "the input dimensions of 'b'")
1835: (8)                   self.input_dims = a.input_dims
1836: (8)                   self.output_dims = b.output_dims
1837: (8)                   super().__init__(**kwargs)
1838: (8)                   self._a = a
1839: (8)                   self._b = b
1840: (8)                   self.set_children(a, b)
1841: (4)               def frozen(self):
1842: (8)                   self._invalid = 0
1843: (8)                   frozen = composite_transform_factory(
1844: (12)                      self._a.frozen(), self._b.frozen())
1845: (8)                   if not isinstance(frozen, CompositeGenericTransform):
1846: (12)                      return frozen.frozen()
1847: (8)                   return frozen
1848: (4)               def _invalidate_internal(self, level, invalidating_node):
1849: (8)                   if invalidating_node is self._a and not self._b.is_affine:
1850: (12)                      level = Transform._INVALID_FULL
1851: (8)                   super()._invalidate_internal(level, invalidating_node)
1852: (4)               def __eq__(self, other):
1853: (8)                   if isinstance(other, (CompositeGenericTransform, CompositeAffine2D)):
1854: (12)                      return self is other or (self._a == other._a
1855: (37)                                               and self._b == other._b)
1856: (8)                   else:
1857: (12)                      return False
1858: (4)               def _iter_break_from_left_to_right(self):
1859: (8)                   for left, right in self._a._iter_break_from_left_to_right():
1860: (12)                      yield left, right + self._b
1861: (8)                   for left, right in self._b._iter_break_from_left_to_right():
1862: (12)                      yield self._a + left, right
1863: (4)               def contains_branch_seperately(self, other_transform):
1864: (8)                   if self.output_dims != 2:
1865: (12)                      raise ValueError('contains_branch_seperately only supports '
1866: (29)                                       'transforms with 2 output dimensions')
1867: (8)                   if self == other_transform:
1868: (12)                      return (True, True)
1869: (8)                   return self._b.contains_branch_seperately(other_transform)
1870: (4)               depth = property(lambda self: self._a.depth + self._b.depth)
1871: (4)               is_affine = property(lambda self: self._a.is_affine and self._b.is_affine)
1872: (4)               is_separable = property(
1873: (8)                   lambda self: self._a.is_separable and self._b.is_separable)
1874: (4)               has_inverse = property(
1875: (8)                   lambda self: self._a.has_inverse and self._b.has_inverse)
1876: (4)               __str__ = _make_str_method("_a", "_b")
```

```
(4)                 @_api.rename_parameter("3.8", "points", "values")
(4)                 def transform_affine(self, values):
(8)                     return self.get_affine().transform(values)
(4)                 @_api.rename_parameter("3.8", "points", "values")
(4)                 def transform_non_affine(self, values):
(8)                     if self._a.is_affine and self._b.is_affine:
(12)                        return values
(8)                     elif not self._a.is_affine and self._b.is_affine:
(12)                        return self._a.transform_non_affine(values)
(8)                     else:
(12)                        return self._b.transform_non_affine(self._a.transform(values))
(4)                 def transform_path_non_affine(self, path):
(8)                     if self._a.is_affine and self._b.is_affine:
(12)                        return path
(8)                     elif not self._a.is_affine and self._b.is_affine:
(12)                        return self._a.transform_path_non_affine(path)
(8)                     else:
(12)                        return self._b.transform_path_non_affine(
(36)                                                 self._a.transform_path(path))
(4)                 def get_affine(self):
(8)                     if not self._b.is_affine:
(12)                        return self._b.get_affine()
(8)                     else:
(12)                        return Affine2D(np.dot(self._b.get_affine().get_matrix(),
(35)                                                self._a.get_affine().get_matrix()))
(4)                 def inverted(self):
(8)                     return CompositeGenericTransform(
(12)                        self._b.inverted(), self._a.inverted())
(0)             class CompositeAffine2D(Affine2DBase):
(4)                 """
(4)                 A composite transform formed by applying transform *a* then transform *b*.
(4)                 This version is an optimization that handles the case where both *a*
(4)                 and *b* are 2D affines.
(4)                 """
(4)                 def __init__(self, a, b, **kwargs):
(8)                     """
(8)                     Create a new composite transform that is the result of
(8)                     applying `Affine2DBase` *a* then `Affine2DBase` *b*.
(8)                     You will generally not call this constructor directly but write ``a +
(8)                     b`` instead, which will automatically choose the best kind of
composite
(8)                     transform instance to create.
(8)                     """
(8)                     if not a.is_affine or not b.is_affine:
(12)                        raise ValueError("'a' and 'b' must be affine transforms")
(8)                     if a.output_dims != b.input_dims:
(12)                        raise ValueError("The output dimension of 'a' must be equal to "
(29)                                         "the input dimensions of 'b'")
(8)                     self.input_dims = a.input_dims
(8)                     self.output_dims = b.output_dims
(8)                     super().__init__(**kwargs)
(8)                     self._a = a
(8)                     self._b = b
(8)                     self.set_children(a, b)
(8)                     self._mtx = None
(4)                 @property
(4)                 def depth(self):
(8)                     return self._a.depth + self._b.depth
(4)                 def _iter_break_from_left_to_right(self):
(8)                     for left, right in self._a._iter_break_from_left_to_right():
(12)                        yield left, right + self._b
(8)                     for left, right in self._b._iter_break_from_left_to_right():
(12)                        yield self._a + left, right
(4)                 __str__ = _make_str_method("_a", "_b")
(4)                 def get_matrix(self):
(8)                     if self._invalid:
(12)                        self._mtx = np.dot(
(16)                            self._b.get_matrix(),
(16)                            self._a.get_matrix())
```

```
(12)                    self._inverted = None
(12)                    self._invalid = 0
(8)                 return self._mtx
(0)         def composite_transform_factory(a, b):
(4)             """
(4)             Create a new composite transform that is the result of applying
(4)             transform a then transform b.
(4)             Shortcut versions of the blended transform are provided for the
(4)             case where both child transforms are affine, or one or the other
(4)             is the identity transform.
(4)             Composite transforms may also be created using the '+' operator,
(4)             e.g.::
(6)               c = a + b
(4)             """
(4)             if isinstance(a, IdentityTransform):
(8)                 return b
(4)             elif isinstance(b, IdentityTransform):
(8)                 return a
(4)             elif isinstance(a, Affine2D) and isinstance(b, Affine2D):
(8)                 return CompositeAffine2D(a, b)
(4)             return CompositeGenericTransform(a, b)
(0)         class BboxTransform(Affine2DBase):
(4)             """
(4)             `BboxTransform` linearly transforms points from one `Bbox` to another.
(4)             """
(4)             is_separable = True
(4)             def __init__(self, boxin, boxout, **kwargs):
(8)                 """
(8)                 Create a new `BboxTransform` that linearly transforms
(8)                 points from *boxin* to *boxout*.
(8)                 """
(8)                 _api.check_isinstance(BboxBase, boxin=boxin, boxout=boxout)
(8)                 super().__init__(**kwargs)
(8)                 self._boxin = boxin
(8)                 self._boxout = boxout
(8)                 self.set_children(boxin, boxout)
(8)                 self._mtx = None
(8)                 self._inverted = None
(4)             __str__ = _make_str_method("_boxin", "_boxout")
(4)             def get_matrix(self):
(8)                 if self._invalid:
(12)                    inl, inb, inw, inh = self._boxin.bounds
(12)                    outl, outb, outw, outh = self._boxout.bounds
(12)                    x_scale = outw / inw
(12)                    y_scale = outh / inh
(12)                    if DEBUG and (x_scale == 0 or y_scale == 0):
(16)                        raise ValueError(
(20)                            "Transforming from or to a singular bounding box")
(12)                    self._mtx = np.array([[x_scale, 0.0    , (-inl*x_scale+outl)],
(34)                                          [0.0    , y_scale, (-inb*y_scale+outb)],
(34)                                          [0.0    , 0.0    , 1.0        ]],
(33)                                         float)
(12)                    self._inverted = None
(12)                    self._invalid = 0
(8)                 return self._mtx
(0)         class BboxTransformTo(Affine2DBase):
(4)             """
(4)             `BboxTransformTo` is a transformation that linearly transforms points from
(4)             the unit bounding box to a given `Bbox`.
(4)             """
(4)             is_separable = True
(4)             def __init__(self, boxout, **kwargs):
(8)                 """
(8)                 Create a new `BboxTransformTo` that linearly transforms
(8)                 points from the unit bounding box to *boxout*.
(8)                 """
(8)                 _api.check_isinstance(BboxBase, boxout=boxout)
(8)                 super().__init__(**kwargs)
(8)                 self._boxout = boxout
```

```
2014: (8)                   self.set_children(boxout)
2015: (8)                   self._mtx = None
2016: (8)                   self._inverted = None
2017: (4)               __str__ = _make_str_method("_boxout")
2018: (4)               def get_matrix(self):
2019: (8)                   if self._invalid:
2020: (12)                      outl, outb, outw, outh = self._boxout.bounds
2021: (12)                      if DEBUG and (outw == 0 or outh == 0):
2022: (16)                          raise ValueError("Transforming to a singular bounding box.")
2023: (12)                      self._mtx = np.array([[outw,  0.0, outl],
2024: (34)                                            [ 0.0, outh, outb],
2025: (34)                                            [ 0.0,  0.0,  1.0]],
2026: (33)                                           float)
2027: (12)                      self._inverted = None
2028: (12)                      self._invalid = 0
2029: (8)                   return self._mtx
2030: (0)           @_api.deprecated("3.9")
2031: (0)           class BboxTransformToMaxOnly(BboxTransformTo):
2032: (4)               """
2033: (4)               `BboxTransformToMaxOnly` is a transformation that linearly transforms
points from
2034: (4)               the unit bounding box to a given `Bbox` with a fixed upper left of (0, 0).
2035: (4)               """
2036: (4)               def get_matrix(self):
2037: (8)                   if self._invalid:
2038: (12)                      xmax, ymax = self._boxout.max
2039: (12)                      if DEBUG and (xmax == 0 or ymax == 0):
2040: (16)                          raise ValueError("Transforming to a singular bounding box.")
2041: (12)                      self._mtx = np.array([[xmax,  0.0, 0.0],
2042: (34)                                            [ 0.0, ymax, 0.0],
2043: (34)                                            [ 0.0,  0.0, 1.0]],
2044: (33)                                           float)
2045: (12)                      self._inverted = None
2046: (12)                      self._invalid = 0
2047: (8)                   return self._mtx
2048: (0)           class BboxTransformFrom(Affine2DBase):
2049: (4)               """
2050: (4)               `BboxTransformFrom` linearly transforms points from a given `Bbox` to the
2051: (4)               unit bounding box.
2052: (4)               """
2053: (4)               is_separable = True
2054: (4)               def __init__(self, boxin, **kwargs):
2055: (8)                   _api.check_isinstance(BboxBase, boxin=boxin)
2056: (8)                   super().__init__(**kwargs)
2057: (8)                   self._boxin = boxin
2058: (8)                   self.set_children(boxin)
2059: (8)                   self._mtx = None
2060: (8)                   self._inverted = None
2061: (4)               __str__ = _make_str_method("_boxin")
2062: (4)               def get_matrix(self):
2063: (8)                   if self._invalid:
2064: (12)                      inl, inb, inw, inh = self._boxin.bounds
2065: (12)                      if DEBUG and (inw == 0 or inh == 0):
2066: (16)                          raise ValueError("Transforming from a singular bounding box.")
2067: (12)                      x_scale = 1.0 / inw
2068: (12)                      y_scale = 1.0 / inh
2069: (12)                      self._mtx = np.array([[x_scale, 0.0    , (-inl*x_scale)],
2070: (34)                                            [0.0    , y_scale, (-inb*y_scale)],
2071: (34)                                            [0.0    , 0.0    , 1.0        ]],
2072: (33)                                           float)
2073: (12)                      self._inverted = None
2074: (12)                      self._invalid = 0
2075: (8)                   return self._mtx
2076: (0)           class ScaledTranslation(Affine2DBase):
2077: (4)               """
2078: (4)               A transformation that translates by *xt* and *yt*, after *xt* and *yt*
2079: (4)               have been transformed by *scale_trans*.
2080: (4)               """
2081: (4)               def __init__(self, xt, yt, scale_trans, **kwargs):
```

```
2082: (8)                   super().__init__(**kwargs)
2083: (8)                   self._t = (xt, yt)
2084: (8)                   self._scale_trans = scale_trans
2085: (8)                   self.set_children(scale_trans)
2086: (8)                   self._mtx = None
2087: (8)                   self._inverted = None
2088: (4)               __str__ = _make_str_method("_t")
2089: (4)               def get_matrix(self):
2090: (8)                   if self._invalid:
2091: (12)                      self._mtx = IdentityTransform._mtx.copy()
2092: (12)                      self._mtx[:2, 2] = self._scale_trans.transform(self._t)
2093: (12)                      self._invalid = 0
2094: (12)                      self._inverted = None
2095: (8)                   return self._mtx
2096: (0)           class AffineDeltaTransform(Affine2DBase):
2097: (4)               r"""
2098: (4)               A transform wrapper for transforming displacements between pairs of
points.
2099: (4)               This class is intended to be used to transform displacements ("position
2100: (4)               deltas") between pairs of points (e.g., as the ``offset_transform``
2101: (4)               of `.Collection`\s): given a transform ``t`` such that ``t =
2102: (4)               AffineDeltaTransform(t) + offset``, ``AffineDeltaTransform``
2103: (4)               satisfies ``AffineDeltaTransform(a - b) == AffineDeltaTransform(a) -
2104: (4)               AffineDeltaTransform(b)``.
2105: (4)               This is implemented by forcing the offset components of the transform
2106: (4)               matrix to zero.
2107: (4)               This class is experimental as of 3.3, and the API may change.
2108: (4)               """
2109: (4)               def __init__(self, transform, **kwargs):
2110: (8)                   super().__init__(**kwargs)
2111: (8)                   self._base_transform = transform
2112: (4)               __str__ = _make_str_method("_base_transform")
2113: (4)               def get_matrix(self):
2114: (8)                   if self._invalid:
2115: (12)                      self._mtx = self._base_transform.get_matrix().copy()
2116: (12)                      self._mtx[:2, -1] = 0
2117: (8)                   return self._mtx
2118: (0)           class TransformedPath(TransformNode):
2119: (4)               """
2120: (4)               A `TransformedPath` caches a non-affine transformed copy of the
2121: (4)               `~.path.Path`.  This cached copy is automatically updated when the
2122: (4)               non-affine part of the transform changes.
2123: (4)               .. note::
2124: (8)                   Paths are considered immutable by this class. Any update to the
2125: (8)                   path's vertices/codes will not trigger a transform recomputation.
2126: (4)               """
2127: (4)               def __init__(self, path, transform):
2128: (8)                   """
2129: (8)                   Parameters
2130: (8)                   ----------
2131: (8)                   path : `~.path.Path`
2132: (8)                   transform : `Transform`
2133: (8)                   """
2134: (8)                   _api.check_isinstance(Transform, transform=transform)
2135: (8)                   super().__init__()
2136: (8)                   self._path = path
2137: (8)                   self._transform = transform
2138: (8)                   self.set_children(transform)
2139: (8)                   self._transformed_path = None
2140: (8)                   self._transformed_points = None
2141: (4)               def _revalidate(self):
2142: (8)                   if (self._invalid == self._INVALID_FULL
2143: (16)                          or self._transformed_path is None):
2144: (12)                      self._transformed_path = \
2145: (16)                          self._transform.transform_path_non_affine(self._path)
2146: (12)                      self._transformed_points = \
2147: (16)                          Path._fast_from_codes_and_verts(
2148: (20)                              self._transform.transform_non_affine(self._path.vertices),
2149: (20)                              None, self._path)
```

```
(8)                 self._invalid = 0
(4)             def get_transformed_points_and_affine(self):
(8)                 """
(8)                 Return a copy of the child path, with the non-affine part of
(8)                 the transform already applied, along with the affine part of
(8)                 the path necessary to complete the transformation.  Unlike
(8)                 :meth:`get_transformed_path_and_affine`, no interpolation will
(8)                 be performed.
(8)                 """
(8)                 self._revalidate()
(8)                 return self._transformed_points, self.get_affine()
(4)             def get_transformed_path_and_affine(self):
(8)                 """
(8)                 Return a copy of the child path, with the non-affine part of
(8)                 the transform already applied, along with the affine part of
(8)                 the path necessary to complete the transformation.
(8)                 """
(8)                 self._revalidate()
(8)                 return self._transformed_path, self.get_affine()
(4)             def get_fully_transformed_path(self):
(8)                 """
(8)                 Return a fully-transformed copy of the child path.
(8)                 """
(8)                 self._revalidate()
(8)                 return self._transform.transform_path_affine(self._transformed_path)
(4)             def get_affine(self):
(8)                 return self._transform.get_affine()
(0)         class TransformedPatchPath(TransformedPath):
(4)             """
(4)             A `TransformedPatchPath` caches a non-affine transformed copy of the
(4)             `~.patches.Patch`. This cached copy is automatically updated when the
(4)             non-affine part of the transform or the patch changes.
(4)             """
(4)             def __init__(self, patch):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 patch : `~.patches.Patch`
(8)                 """
(8)                 super().__init__(patch.get_path(), patch.get_transform())
(8)                 self._patch = patch
(4)             def _revalidate(self):
(8)                 patch_path = self._patch.get_path()
(8)                 if patch_path != self._path:
(12)                    self._path = patch_path
(12)                    self._transformed_path = None
(8)                 super()._revalidate()
(0)         def nonsingular(vmin, vmax, expander=0.001, tiny=1e-15, increasing=True):
(4)             """
(4)             Modify the endpoints of a range as needed to avoid singularities.
(4)             Parameters
(4)             ----------
(4)             vmin, vmax : float
(8)                 The initial endpoints.
(4)             expander : float, default: 0.001
(8)                 Fractional amount by which *vmin* and *vmax* are expanded if
(8)                 the original interval is too small, based on *tiny*.
(4)             tiny : float, default: 1e-15
(8)                 Threshold for the ratio of the interval to the maximum absolute
(8)                 value of its endpoints.  If the interval is smaller than
(8)                 this, it will be expanded.  This value should be around
(8)                 1e-15 or larger; otherwise the interval will be approaching
(8)                 the double precision resolution limit.
(4)             increasing : bool, default: True
(8)                 If True, swap *vmin*, *vmax* if *vmin* > *vmax*.
(4)             Returns
(4)             -------
(4)             vmin, vmax : float
(8)                 Endpoints, expanded and/or swapped if necessary.
```

```
(8)                 If either input is inf or NaN, or if both inputs are 0 or very
(8)                 close to zero, it returns -*expander*, *expander*.
(4)             """
(4)             if (not np.isfinite(vmin)) or (not np.isfinite(vmax)):
(8)                 return -expander, expander
(4)             swapped = False
(4)             if vmax < vmin:
(8)                 vmin, vmax = vmax, vmin
(8)                 swapped = True
(4)             vmin, vmax = map(float, [vmin, vmax])
(4)             maxabsvalue = max(abs(vmin), abs(vmax))
(4)             if maxabsvalue < (1e6 / tiny) * np.finfo(float).tiny:
(8)                 vmin = -expander
(8)                 vmax = expander
(4)             elif vmax - vmin <= maxabsvalue * tiny:
(8)                 if vmax == 0 and vmin == 0:
(12)                    vmin = -expander
(12)                    vmax = expander
(8)                 else:
(12)                    vmin -= expander*abs(vmin)
(12)                    vmax += expander*abs(vmax)
(4)             if swapped and not increasing:
(8)                 vmin, vmax = vmax, vmin
(4)             return vmin, vmax
(0)         def interval_contains(interval, val):
(4)             """
(4)             Check, inclusively, whether an interval includes a given value.
(4)             Parameters
(4)             ----------
(4)             interval : (float, float)
(8)                 The endpoints of the interval.
(4)             val : float
(8)                 Value to check is within interval.
(4)             Returns
(4)             -------
(4)             bool
(8)                 Whether *val* is within the *interval*.
(4)             """
(4)             a, b = interval
(4)             if a > b:
(8)                 a, b = b, a
(4)             return a <= val <= b
(0)         def _interval_contains_close(interval, val, rtol=1e-10):
(4)             """
(4)             Check, inclusively, whether an interval includes a given value, with the
(4)             interval expanded by a small tolerance to admit floating point errors.
(4)             Parameters
(4)             ----------
(4)             interval : (float, float)
(8)                 The endpoints of the interval.
(4)             val : float
(8)                 Value to check is within interval.
(4)             rtol : float, default: 1e-10
(8)                 Relative tolerance slippage allowed outside of the interval.
(8)                 For an interval ``[a, b]``, values
(8)                 ``a - rtol * (b - a) <= val <= b + rtol * (b - a)`` are considered
(8)                 inside the interval.
(4)             Returns
(4)             -------
(4)             bool
(8)                 Whether *val* is within the *interval* (with tolerance).
(4)             """
(4)             a, b = interval
(4)             if a > b:
(8)                 a, b = b, a
(4)             rtol = (b - a) * rtol
(4)             return a - rtol <= val <= b + rtol
(0)         def interval_contains_open(interval, val):
(4)             """
```

```
2288: (4)               Check, excluding endpoints, whether an interval includes a given value.
2289: (4)               Parameters
2290: (4)               ----------
2291: (4)               interval : (float, float)
2292: (8)                   The endpoints of the interval.
2293: (4)               val : float
2294: (8)                   Value to check is within interval.
2295: (4)               Returns
2296: (4)               -------
2297: (4)               bool
2298: (8)                   Whether *val* is within the *interval*.
2299: (4)               """
2300: (4)               a, b = interval
2301: (4)               return a < val < b or a > val > b
2302: (0)           def offset_copy(trans, fig=None, x=0.0, y=0.0, units='inches'):
2303: (4)               """
2304: (4)               Return a new transform with an added offset.
2305: (4)               Parameters
2306: (4)               ----------
2307: (4)               trans : `Transform` subclass
2308: (8)                   Any transform, to which offset will be applied.
2309: (4)               fig : `~matplotlib.figure.Figure`, default: None
2310: (8)                   Current figure. It can be None if *units* are 'dots'.
2311: (4)               x, y : float, default: 0.0
2312: (8)                   The offset to apply.
2313: (4)               units : {'inches', 'points', 'dots'}, default: 'inches'
2314: (8)                   Units of the offset.
2315: (4)               Returns
2316: (4)               -------
2317: (4)               `Transform` subclass
2318: (8)                   Transform with applied offset.
2319: (4)               """
2320: (4)               _api.check_in_list(['dots', 'points', 'inches'], units=units)
2321: (4)               if units == 'dots':
2322: (8)                   return trans + Affine2D().translate(x, y)
2323: (4)               if fig is None:
2324: (8)                   raise ValueError('For units of inches or points a fig kwarg is
needed')
2325: (4)               if units == 'points':
2326: (8)                   x /= 72.0
2327: (8)                   y /= 72.0
2328: (4)               return trans + ScaledTranslation(x, y, fig.dpi_scale_trans)

----------------------------------------

File 69 - typing.py:

1: (0)              """
2: (0)              Typing support for Matplotlib
3: (0)              This module contains Type aliases which are useful for Matplotlib and
potentially
4: (0)              downstream libraries.
5: (0)              .. admonition:: Provisional status of typing
6: (4)                  The ``typing`` module and type stub files are considered provisional and
may change
7: (4)                  at any time without a deprecation period.
8: (0)              """
9: (0)              from collections.abc import Hashable, Sequence
10: (0)             import pathlib
11: (0)             from typing import Any, Literal, TypeVar, Union
12: (0)             from . import path
13: (0)             from ._enums import JoinStyle, CapStyle
14: (0)             from .markers import MarkerStyle
15: (0)             RGBColorType = Union[tuple[float, float, float], str]
16: (0)             RGBAColorType = Union[
17: (4)                 str,  # "none" or "#RRGGBBAA"/"#RGBA" hex strings
18: (4)                 tuple[float, float, float, float],
19: (4)                 tuple[RGBColorType, float],
20: (4)                 tuple[tuple[float, float, float, float], float],
```

```
21: (0)             ]
22: (0)             ColorType = Union[RGBColorType, RGBAColorType]
23: (0)             RGBColourType = RGBColorType
24: (0)             RGBAColourType = RGBAColorType
25: (0)             ColourType = ColorType
26: (0)             LineStyleType = Union[str, tuple[float, Sequence[float]]]
27: (0)             DrawStyleType = Literal["default", "steps", "steps-pre", "steps-mid", "steps-
post"]
28: (0)             MarkEveryType = Union[
29: (4)                 None, int, tuple[int, int], slice, list[int], float, tuple[float, float],
list[bool]
30: (0)             ]
31: (0)             MarkerType = Union[str, path.Path, MarkerStyle]
32: (0)             FillStyleType = Literal["full", "left", "right", "bottom", "top", "none"]
33: (0)             JoinStyleType = Union[JoinStyle, Literal["miter", "round", "bevel"]]
34: (0)             CapStyleType = Union[CapStyle, Literal["butt", "projecting", "round"]]
35: (0)             RcStyleType = Union[
36: (4)                 str,
37: (4)                 dict[str, Any],
38: (4)                 pathlib.Path,
39: (4)                 Sequence[Union[str, pathlib.Path, dict[str, Any]]],
40: (0)             ]
41: (0)             _HT = TypeVar("_HT", bound=Hashable)
42: (0)             HashableList = list[Union[_HT, "HashableList[_HT]"]]
43: (0)             """A nested list of Hashable values."""
```

File 70 - units.py:

```
1: (0)              """
2: (0)              The classes here provide support for using custom classes with
3: (0)              Matplotlib, e.g., those that do not expose the array interface but know
4: (0)              how to convert themselves to arrays.  It also supports classes with
5: (0)              units and units conversion.  Use cases include converters for custom
6: (0)              objects, e.g., a list of datetime objects, as well as for objects that
7: (0)              are unit aware.  We don't assume any particular units implementation;
8: (0)              rather a units implementation must register with the Registry converter
9: (0)              dictionary and provide a `ConversionInterface`.  For example,
10: (0)             here is a complete implementation which supports plotting with native
11: (0)             datetime objects::
12: (4)                 import matplotlib.units as units
13: (4)                 import matplotlib.dates as dates
14: (4)                 import matplotlib.ticker as ticker
15: (4)                 import datetime
16: (4)                 class DateConverter(units.ConversionInterface):
17: (8)                     @staticmethod
18: (8)                     def convert(value, unit, axis):
19: (12)                        "Convert a datetime value to a scalar or array."
20: (12)                        return dates.date2num(value)
21: (8)                     @staticmethod
22: (8)                     def axisinfo(unit, axis):
23: (12)                        "Return major and minor tick locators and formatters."
24: (12)                        if unit != 'date':
25: (16)                            return None
26: (12)                        majloc = dates.AutoDateLocator()
27: (12)                        majfmt = dates.AutoDateFormatter(majloc)
28: (12)                        return units.AxisInfo(majloc=majloc, majfmt=majfmt, label='date')
29: (8)                     @staticmethod
30: (8)                     def default_units(x, axis):
31: (12)                        "Return the default unit for x or None."
32: (12)                        return 'date'
33: (4)                 units.registry[datetime.date] = DateConverter()
34: (0)             """
35: (0)             from decimal import Decimal
36: (0)             from numbers import Number
37: (0)             import numpy as np
38: (0)             from numpy import ma
39: (0)             from matplotlib import cbook
```

```
(0)             class ConversionError(TypeError):
(4)                 pass
(0)             def _is_natively_supported(x):
(4)                 """
(4)                 Return whether *x* is of a type that Matplotlib natively supports or an
(4)                 array of objects of such types.
(4)                 """
(4)                 if np.iterable(x):
(8)                     for thisx in x:
(12)                        if thisx is ma.masked:
(16)                            continue
(12)                        return isinstance(thisx, Number) and not isinstance(thisx,
Decimal)
(4)                 else:
(8)                     return isinstance(x, Number) and not isinstance(x, Decimal)
(0)             class AxisInfo:
(4)                 """
(4)                 Information to support default axis labeling, tick labeling, and limits.
(4)                 An instance of this class must be returned by
(4)                 `ConversionInterface.axisinfo`.
(4)                 """
(4)                 def __init__(self, majloc=None, minloc=None,
(17)                             majfmt=None, minfmt=None, label=None,
(17)                             default_limits=None):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     majloc, minloc : Locator, optional
(12)                        Tick locators for the major and minor ticks.
(8)                     majfmt, minfmt : Formatter, optional
(12)                        Tick formatters for the major and minor ticks.
(8)                     label : str, optional
(12)                        The default axis label.
(8)                     default_limits : optional
(12)                        The default min and max limits of the axis if no data has
(12)                        been plotted.
(8)                     Notes
(8)                     -----
(8)                     If any of the above are ``None``, the axis will simply use the
(8)                     default value.
(8)                     """
(8)                     self.majloc = majloc
(8)                     self.minloc = minloc
(8)                     self.majfmt = majfmt
(8)                     self.minfmt = minfmt
(8)                     self.label = label
(8)                     self.default_limits = default_limits
(0)             class ConversionInterface:
(4)                 """
(4)                 The minimal interface for a converter to take custom data types (or
(4)                 sequences) and convert them to values Matplotlib can use.
(4)                 """
(4)                 @staticmethod
(4)                 def axisinfo(unit, axis):
(8)                     """Return an `.AxisInfo` for the axis with the specified units."""
(8)                     return None
(4)                 @staticmethod
(4)                 def default_units(x, axis):
(8)                     """Return the default unit for *x* or ``None`` for the given axis."""
(8)                     return None
(4)                 @staticmethod
(4)                 def convert(obj, unit, axis):
(8)                     """
(8)                     Convert *obj* using *unit* for the specified *axis*.
(8)                     If *obj* is a sequence, return the converted sequence.  The output
must
(8)                     be a sequence of scalars that can be used by the numpy array layer.
(8)                     """
(8)                     return obj
```

```
107: (0)            class DecimalConverter(ConversionInterface):
108: (4)                """Converter for decimal.Decimal data to float."""
109: (4)                @staticmethod
110: (4)                def convert(value, unit, axis):
111: (8)                    """
112: (8)                    Convert Decimals to floats.
113: (8)                    The *unit* and *axis* arguments are not used.
114: (8)                    Parameters
115: (8)                    ----------
116: (8)                    value : decimal.Decimal or iterable
117: (12)                       Decimal or list of Decimal need to be converted
118: (8)                    """
119: (8)                    if isinstance(value, Decimal):
120: (12)                       return float(value)
121: (8)                    elif isinstance(value, ma.MaskedArray):
122: (12)                       return ma.asarray(value, dtype=float)
123: (8)                    else:
124: (12)                       return np.asarray(value, dtype=float)
125: (0)            class Registry(dict):
126: (4)                """Register types with conversion interface."""
127: (4)                def get_converter(self, x):
128: (8)                    """Get the converter interface instance for *x*, or None."""
129: (8)                    x = cbook._unpack_to_numpy(x)
130: (8)                    if isinstance(x, np.ndarray):
131: (12)                       x = np.ma.getdata(x).ravel()
132: (12)                       if not x.size:
133: (16)                           return self.get_converter(np.array([0], dtype=x.dtype))
134: (8)                    for cls in type(x).__mro__:  # Look up in the cache.
135: (12)                       try:
136: (16)                           return self[cls]
137: (12)                       except KeyError:
138: (16)                           pass
139: (8)                    try:  # If cache lookup fails, look up based on first element...
140: (12)                       first = cbook._safe_first_finite(x)
141: (8)                    except (TypeError, StopIteration):
142: (12)                       pass
143: (8)                    else:
144: (12)                       if type(first) is not type(x):
145: (16)                           return self.get_converter(first)
146: (8)                    return None
147: (0)            registry = Registry()
148: (0)            registry[Decimal] = DecimalConverter()
```

File 71 - widgets.py:

```
1: (0)              """
2: (0)              GUI neutral widgets
3: (0)              ===================
4: (0)              Widgets that are designed to work for any of the GUI backends.
5: (0)              All of these widgets require you to predefine an `~.axes.Axes`
6: (0)              instance and pass that as the first parameter.  Matplotlib doesn't try to
7: (0)              be too smart with respect to layout -- you will have to figure out how
8: (0)              wide and tall you want your Axes to be to accommodate your widget.
9: (0)              """
10: (0)             from contextlib import ExitStack
11: (0)             import copy
12: (0)             import itertools
13: (0)             from numbers import Integral, Number
14: (0)             from cycler import cycler
15: (0)             import numpy as np
16: (0)             import matplotlib as mpl
17: (0)             from . import (_api, _docstring, backend_tools, cbook, collections, colors,
18: (15)                           text as mtext, ticker, transforms)
19: (0)             from .lines import Line2D
20: (0)             from .patches import Rectangle, Ellipse, Polygon
21: (0)             from .transforms import TransformedPatchPath, Affine2D
22: (0)             class LockDraw:
```

```
23: (4)                 """
24: (4)                 Some widgets, like the cursor, draw onto the canvas, and this is not
25: (4)                 desirable under all circumstances, like when the toolbar is in zoom-to-
rect
26: (4)                 mode and drawing a rectangle.  To avoid this, a widget can acquire a
27: (4)                 canvas' lock with ``canvas.widgetlock(widget)`` before drawing on the
28: (4)                 canvas; this will prevent other widgets from doing so at the same time (if
29: (4)                 they also try to acquire the lock first).
30: (4)                 """
31: (4)                 def __init__(self):
32: (8)                     self._owner = None
33: (4)                 def __call__(self, o):
34: (8)                     """Reserve the lock for *o*."""
35: (8)                     if not self.available(o):
36: (12)                        raise ValueError('already locked')
37: (8)                     self._owner = o
38: (4)                 def release(self, o):
39: (8)                     """Release the lock from *o*."""
40: (8)                     if not self.available(o):
41: (12)                        raise ValueError('you do not own this lock')
42: (8)                     self._owner = None
43: (4)                 def available(self, o):
44: (8)                     """Return whether drawing is available to *o*."""
45: (8)                     return not self.locked() or self.isowner(o)
46: (4)                 def isowner(self, o):
47: (8)                     """Return whether *o* owns this lock."""
48: (8)                     return self._owner is o
49: (4)                 def locked(self):
50: (8)                     """Return whether the lock is currently held by an owner."""
51: (8)                     return self._owner is not None
52: (0)             class Widget:
53: (4)                 """
54: (4)                 Abstract base class for GUI neutral widgets.
55: (4)                 """
56: (4)                 drawon = True
57: (4)                 eventson = True
58: (4)                 _active = True
59: (4)                 def set_active(self, active):
60: (8)                     """Set whether the widget is active."""
61: (8)                     self._active = active
62: (4)                 def get_active(self):
63: (8)                     """Get whether the widget is active."""
64: (8)                     return self._active
65: (4)                 active = property(get_active, set_active, doc="Is the widget active?")
66: (4)                 def ignore(self, event):
67: (8)                     """
68: (8)                     Return whether *event* should be ignored.
69: (8)                     This method should be called at the beginning of any event callback.
70: (8)                     """
71: (8)                     return not self.active
72: (0)             class AxesWidget(Widget):
73: (4)                 """
74: (4)                 Widget connected to a single `~matplotlib.axes.Axes`.
75: (4)                 To guarantee that the widget remains responsive and not garbage-collected,
76: (4)                 a reference to the object should be maintained by the user.
77: (4)                 This is necessary because the callback registry
78: (4)                 maintains only weak-refs to the functions, which are member
79: (4)                 functions of the widget.  If there are no references to the widget
80: (4)                 object it may be garbage collected which will disconnect the callbacks.
81: (4)                 Attributes
82: (4)                 ----------
83: (4)                 ax : `~matplotlib.axes.Axes`
84: (8)                     The parent Axes for the widget.
85: (4)                 canvas : `~matplotlib.backend_bases.FigureCanvasBase`
86: (8)                     The parent figure canvas for the widget.
87: (4)                 active : bool
88: (8)                     If False, the widget does not respond to events.
89: (4)                 """
90: (4)                 def __init__(self, ax):
```

```
 91: (8)                     self.ax = ax
 92: (8)                     self._cids = []
 93: (4)                 canvas = property(lambda self: self.ax.figure.canvas)
 94: (4)                 def connect_event(self, event, callback):
 95: (8)                     """
 96: (8)                     Connect a callback function with an event.
 97: (8)                     This should be used in lieu of ``figure.canvas.mpl_connect`` since
this
 98: (8)                     function stores callback ids for later clean up.
 99: (8)                     """
 100: (8)                    cid = self.canvas.mpl_connect(event, callback)
 101: (8)                    self._cids.append(cid)
 102: (4)                def disconnect_events(self):
 103: (8)                    """Disconnect all events created by this widget."""
 104: (8)                    for c in self._cids:
 105: (12)                       self.canvas.mpl_disconnect(c)
 106: (4)                def _get_data_coords(self, event):
 107: (8)                    """Return *event*'s data coordinates in this widget's Axes."""
 108: (8)                    return ((event.xdata, event.ydata) if event.inaxes is self.ax
 109: (16)                           else self.ax.transData.inverted().transform((event.x,
event.y)))
 110: (0)            class Button(AxesWidget):
 111: (4)                """
 112: (4)                A GUI neutral button.
 113: (4)                For the button to remain responsive you must keep a reference to it.
 114: (4)                Call `.on_clicked` to connect to the button.
 115: (4)                Attributes
 116: (4)                ----------
 117: (4)                ax
 118: (8)                    The `~.axes.Axes` the button renders into.
 119: (4)                label
 120: (8)                    A `.Text` instance.
 121: (4)                color
 122: (8)                    The color of the button when not hovering.
 123: (4)                hovercolor
 124: (8)                    The color of the button when hovering.
 125: (4)                """
 126: (4)                def __init__(self, ax, label, image=None,
 127: (17)                           color='0.85', hovercolor='0.95', *, useblit=True):
 128: (8)                    """
 129: (8)                    Parameters
 130: (8)                    ----------
 131: (8)                    ax : `~matplotlib.axes.Axes`
 132: (12)                       The `~.axes.Axes` instance the button will be placed into.
 133: (8)                    label : str
 134: (12)                       The button text.
 135: (8)                    image : array-like or PIL Image
 136: (12)                       The image to place in the button, if not *None*.  The parameter is
 137: (12)                       directly forwarded to `~.axes.Axes.imshow`.
 138: (8)                    color : :mpltype:`color`
 139: (12)                       The color of the button when not activated.
 140: (8)                    hovercolor : :mpltype:`color`
 141: (12)                       The color of the button when the mouse is over it.
 142: (8)                    useblit : bool, default: True
 143: (12)                       Use blitting for faster drawing if supported by the backend.
 144: (12)                       See the tutorial :ref:`blitting` for details.
 145: (12)                       .. versionadded:: 3.7
 146: (8)                    """
 147: (8)                    super().__init__(ax)
 148: (8)                    if image is not None:
 149: (12)                       ax.imshow(image)
 150: (8)                    self.label = ax.text(0.5, 0.5, label,
 151: (29)                                        verticalalignment='center',
 152: (29)                                        horizontalalignment='center',
 153: (29)                                        transform=ax.transAxes)
 154: (8)                    self._useblit = useblit and self.canvas.supports_blit
 155: (8)                    self._observers = cbook.CallbackRegistry(signals=["clicked"])
 156: (8)                    self.connect_event('button_press_event', self._click)
 157: (8)                    self.connect_event('button_release_event', self._release)
```

```
158: (8)                    self.connect_event('motion_notify_event', self._motion)
159: (8)                    ax.set_navigate(False)
160: (8)                    ax.set_facecolor(color)
161: (8)                    ax.set_xticks([])
162: (8)                    ax.set_yticks([])
163: (8)                    self.color = color
164: (8)                    self.hovercolor = hovercolor
165: (4)                def _click(self, event):
166: (8)                    if not self.eventson or self.ignore(event) or not
self.ax.contains(event)[0]:
167: (12)                       return
168: (8)                    if event.canvas.mouse_grabber != self.ax:
169: (12)                       event.canvas.grab_mouse(self.ax)
170: (4)                def _release(self, event):
171: (8)                    if self.ignore(event) or event.canvas.mouse_grabber != self.ax:
172: (12)                       return
173: (8)                    event.canvas.release_mouse(self.ax)
174: (8)                    if self.eventson and self.ax.contains(event)[0]:
175: (12)                       self._observers.process('clicked', event)
176: (4)                def _motion(self, event):
177: (8)                    if self.ignore(event):
178: (12)                       return
179: (8)                    c = self.hovercolor if self.ax.contains(event)[0] else self.color
180: (8)                    if not colors.same_color(c, self.ax.get_facecolor()):
181: (12)                       self.ax.set_facecolor(c)
182: (12)                       if self.drawon:
183: (16)                           if self._useblit:
184: (20)                               self.ax.draw_artist(self.ax)
185: (20)                               self.canvas.blit(self.ax.bbox)
186: (16)                           else:
187: (20)                               self.canvas.draw()
188: (4)                def on_clicked(self, func):
189: (8)                    """
190: (8)                    Connect the callback function *func* to button click events.
191: (8)                    Returns a connection id, which can be used to disconnect the callback.
192: (8)                    """
193: (8)                    return self._observers.connect('clicked', lambda event: func(event))
194: (4)                def disconnect(self, cid):
195: (8)                    """Remove the callback function with connection id *cid*."""
196: (8)                    self._observers.disconnect(cid)
197: (0)            class SliderBase(AxesWidget):
198: (4)                """
199: (4)                The base class for constructing Slider widgets. Not intended for direct
200: (4)                usage.
201: (4)                For the slider to remain responsive you must maintain a reference to it.
202: (4)                """
203: (4)                def __init__(self, ax, orientation, closedmin, closedmax,
204: (17)                            valmin, valmax, valfmt, dragging, valstep):
205: (8)                    if ax.name == '3d':
206: (12)                       raise ValueError('Sliders cannot be added to 3D Axes')
207: (8)                    super().__init__(ax)
208: (8)                    _api.check_in_list(['horizontal', 'vertical'],
orientation=orientation)
209: (8)                    self.orientation = orientation
210: (8)                    self.closedmin = closedmin
211: (8)                    self.closedmax = closedmax
212: (8)                    self.valmin = valmin
213: (8)                    self.valmax = valmax
214: (8)                    self.valstep = valstep
215: (8)                    self.drag_active = False
216: (8)                    self.valfmt = valfmt
217: (8)                    if orientation == "vertical":
218: (12)                       ax.set_ylim((valmin, valmax))
219: (12)                       axis = ax.yaxis
220: (8)                    else:
221: (12)                       ax.set_xlim((valmin, valmax))
222: (12)                       axis = ax.xaxis
223: (8)                    self._fmt = axis.get_major_formatter()
224: (8)                    if not isinstance(self._fmt, ticker.ScalarFormatter):
```

```
225: (12)                        self._fmt = ticker.ScalarFormatter()
226: (12)                        self._fmt.set_axis(axis)
227: (8)                    self._fmt.set_useOffset(False)  # No additive offset.
228: (8)                    self._fmt.set_useMathText(True)  # x sign before multiplicative
offset.
229: (8)                    ax.set_axis_off()
230: (8)                    ax.set_navigate(False)
231: (8)                    self.connect_event("button_press_event", self._update)
232: (8)                    self.connect_event("button_release_event", self._update)
233: (8)                    if dragging:
234: (12)                       self.connect_event("motion_notify_event", self._update)
235: (8)                    self._observers = cbook.CallbackRegistry(signals=["changed"])
236: (4)                def _stepped_value(self, val):
237: (8)                    """Return *val* coerced to closest number in the ``valstep`` grid."""
238: (8)                    if isinstance(self.valstep, Number):
239: (12)                       val = (self.valmin
240: (19)                              + round((val - self.valmin) / self.valstep) * self.valstep)
241: (8)                    elif self.valstep is not None:
242: (12)                       valstep = np.asanyarray(self.valstep)
243: (12)                       if valstep.ndim != 1:
244: (16)                           raise ValueError(
245: (20)                               f"valstep must have 1 dimension but has {valstep.ndim}"
246: (16)                           )
247: (12)                       val = valstep[np.argmin(np.abs(valstep - val))]
248: (8)                    return val
249: (4)                def disconnect(self, cid):
250: (8)                    """
251: (8)                    Remove the observer with connection id *cid*.
252: (8)                    Parameters
253: (8)                    ----------
254: (8)                    cid : int
255: (12)                       Connection id of the observer to be removed.
256: (8)                    """
257: (8)                    self._observers.disconnect(cid)
258: (4)                def reset(self):
259: (8)                    """Reset the slider to the initial value."""
260: (8)                    if np.any(self.val != self.valinit):
261: (12)                       self.set_val(self.valinit)
262: (0)            class Slider(SliderBase):
263: (4)                """
264: (4)                A slider representing a floating point range.
265: (4)                Create a slider from *valmin* to *valmax* in Axes *ax*. For the slider to
266: (4)                remain responsive you must maintain a reference to it. Call
267: (4)                :meth:`on_changed` to connect to the slider event.
268: (4)                Attributes
269: (4)                ----------
270: (4)                val : float
271: (8)                    Slider value.
272: (4)                """
273: (4)                def __init__(self, ax, label, valmin, valmax, *, valinit=0.5, valfmt=None,
274: (17)                            closedmin=True, closedmax=True, slidermin=None,
275: (17)                            slidermax=None, dragging=True, valstep=None,
276: (17)                            orientation='horizontal', initcolor='r',
277: (17)                            track_color='lightgrey', handle_style=None, **kwargs):
278: (8)                    """
279: (8)                    Parameters
280: (8)                    ----------
281: (8)                    ax : Axes
282: (12)                       The Axes to put the slider in.
283: (8)                    label : str
284: (12)                       Slider label.
285: (8)                    valmin : float
286: (12)                       The minimum value of the slider.
287: (8)                    valmax : float
288: (12)                       The maximum value of the slider.
289: (8)                    valinit : float, default: 0.5
290: (12)                       The slider initial position.
291: (8)                    valfmt : str, default: None
292: (12)                       %-format string used to format the slider value.  If None, a
```

```
(12)                        `.ScalarFormatter` is used instead.
(8)                     closedmin : bool, default: True
(12)                        Whether the slider interval is closed on the bottom.
(8)                     closedmax : bool, default: True
(12)                        Whether the slider interval is closed on the top.
(8)                     slidermin : Slider, default: None
(12)                        Do not allow the current slider to have a value less than
(12)                        the value of the Slider *slidermin*.
(8)                     slidermax : Slider, default: None
(12)                        Do not allow the current slider to have a value greater than
(12)                        the value of the Slider *slidermax*.
(8)                     dragging : bool, default: True
(12)                        If True the slider can be dragged by the mouse.
(8)                     valstep : float or array-like, default: None
(12)                        If a float, the slider will snap to multiples of *valstep*.
(12)                        If an array the slider will snap to the values in the array.
(8)                     orientation : {'horizontal', 'vertical'}, default: 'horizontal'
(12)                        The orientation of the slider.
(8)                     initcolor : :mpltype:`color`, default: 'r'
(12)                        The color of the line at the *valinit* position. Set to ``'none'``
(12)                        for no line.
(8)                     track_color : :mpltype:`color`, default: 'lightgrey'
(12)                        The color of the background track. The track is accessible for
(12)                        further styling via the *track* attribute.
(8)                     handle_style : dict
(12)                        Properties of the slider handle. Default values are
(12)                        ========= ===== ======= ========================================
(12)                        Key       Value Default Description
(12)                        ========= ===== ======= ========================================
(12)                        facecolor color 'white' The facecolor of the slider handle.
(12)                        edgecolor color '.75'   The edgecolor of the slider handle.
(12)                        size      int   10      The size of the slider handle in points.
(12)                        ========= ===== ======= ========================================
(12)                        Other values will be transformed as marker{foo} and passed to the
(12)                        `~.Line2D` constructor. e.g. ``handle_style = {'style'='x'}`` will
(12)                        result in ``markerstyle = 'x'``.
(8)                     Notes
(8)                     -----
(8)                     Additional kwargs are passed on to ``self.poly`` which is the
(8)                     `~matplotlib.patches.Rectangle` that draws the slider knob.  See the
(8)                     `.Rectangle` documentation for valid property names (``facecolor``,
(8)                     ``edgecolor``, ``alpha``, etc.).
(8)                     """
(8)                     super().__init__(ax, orientation, closedmin, closedmax,
(25)                                     valmin, valmax, valfmt, dragging, valstep)
(8)                     if slidermin is not None and not hasattr(slidermin, 'val'):
(12)                        raise ValueError(
(16)                            f"Argument slidermin ({type(slidermin)}) has no 'val'")
(8)                     if slidermax is not None and not hasattr(slidermax, 'val'):
(12)                        raise ValueError(
(16)                            f"Argument slidermax ({type(slidermax)}) has no 'val'")
(8)                     self.slidermin = slidermin
(8)                     self.slidermax = slidermax
(8)                     valinit = self._value_in_bounds(valinit)
(8)                     if valinit is None:
(12)                        valinit = valmin
(8)                     self.val = valinit
(8)                     self.valinit = valinit
(8)                     defaults = {'facecolor': 'white', 'edgecolor': '.75', 'size': 10}
(8)                     handle_style = {} if handle_style is None else handle_style
(8)                     marker_props = {
(12)                        f'marker{k}': v for k, v in {**defaults, **handle_style}.items()
(8)                     }
(8)                     if orientation == 'vertical':
(12)                        self.track = Rectangle(
(16)                            (.25, 0), .5, 1,
(16)                            transform=ax.transAxes,
(16)                            facecolor=track_color
(12)                        )
```

```
362: (12)                        ax.add_patch(self.track)
363: (12)                        self.poly = ax.axhspan(valmin, valinit, .25, .75, **kwargs)
364: (12)                        self.hline = ax.axhline(valinit, 0, 1, color=initcolor, lw=1,
365: (36)
clip_path=TransformedPatchPath(self.track))
366: (12)                        handleXY = [[0.5], [valinit]]
367: (8)                    else:
368: (12)                        self.track = Rectangle(
369: (16)                            (0, .25), 1, .5,
370: (16)                            transform=ax.transAxes,
371: (16)                            facecolor=track_color
372: (12)                        )
373: (12)                        ax.add_patch(self.track)
374: (12)                        self.poly = ax.axvspan(valmin, valinit, .25, .75, **kwargs)
375: (12)                        self.vline = ax.axvline(valinit, 0, 1, color=initcolor, lw=1,
376: (36)
clip_path=TransformedPatchPath(self.track))
377: (12)                        handleXY = [[valinit], [0.5]]
378: (8)                    self._handle, = ax.plot(
379: (12)                        *handleXY,
380: (12)                        "o",
381: (12)                        **marker_props,
382: (12)                        clip_on=False
383: (8)                    )
384: (8)                    if orientation == 'vertical':
385: (12)                        self.label = ax.text(0.5, 1.02, label, transform=ax.transAxes,
386: (33)                                             verticalalignment='bottom',
387: (33)                                             horizontalalignment='center')
388: (12)                        self.valtext = ax.text(0.5, -0.02, self._format(valinit),
389: (35)                                               transform=ax.transAxes,
390: (35)                                               verticalalignment='top',
391: (35)                                               horizontalalignment='center')
392: (8)                    else:
393: (12)                        self.label = ax.text(-0.02, 0.5, label, transform=ax.transAxes,
394: (33)                                             verticalalignment='center',
395: (33)                                             horizontalalignment='right')
396: (12)                        self.valtext = ax.text(1.02, 0.5, self._format(valinit),
397: (35)                                               transform=ax.transAxes,
398: (35)                                               verticalalignment='center',
399: (35)                                               horizontalalignment='left')
400: (8)                    self.set_val(valinit)
401: (4)                def _value_in_bounds(self, val):
402: (8)                    """Makes sure *val* is with given bounds."""
403: (8)                    val = self._stepped_value(val)
404: (8)                    if val <= self.valmin:
405: (12)                        if not self.closedmin:
406: (16)                            return
407: (12)                        val = self.valmin
408: (8)                    elif val >= self.valmax:
409: (12)                        if not self.closedmax:
410: (16)                            return
411: (12)                        val = self.valmax
412: (8)                    if self.slidermin is not None and val <= self.slidermin.val:
413: (12)                        if not self.closedmin:
414: (16)                            return
415: (12)                        val = self.slidermin.val
416: (8)                    if self.slidermax is not None and val >= self.slidermax.val:
417: (12)                        if not self.closedmax:
418: (16)                            return
419: (12)                        val = self.slidermax.val
420: (8)                    return val
421: (4)                def _update(self, event):
422: (8)                    """Update the slider position."""
423: (8)                    if self.ignore(event) or event.button != 1:
424: (12)                        return
425: (8)                    if event.name == 'button_press_event' and self.ax.contains(event)[0]:
426: (12)                        self.drag_active = True
427: (12)                        event.canvas.grab_mouse(self.ax)
428: (8)                    if not self.drag_active:
```

```
429: (12)                        return
430: (8)                    if (event.name == 'button_release_event'
431: (14)                          or event.name == 'button_press_event' and not
self.ax.contains(event)[0]):
432: (12)                        self.drag_active = False
433: (12)                        event.canvas.release_mouse(self.ax)
434: (12)                        return
435: (8)                    xdata, ydata = self._get_data_coords(event)
436: (8)                    val = self._value_in_bounds(
437: (12)                        xdata if self.orientation == 'horizontal' else ydata)
438: (8)                    if val not in [None, self.val]:
439: (12)                        self.set_val(val)
440: (4)                def _format(self, val):
441: (8)                    """Pretty-print *val*."""
442: (8)                    if self.valfmt is not None:
443: (12)                        return self.valfmt % val
444: (8)                    else:
445: (12)                        _, s, _ = self._fmt.format_ticks([self.valmin, val, self.valmax])
446: (12)                        return s + self._fmt.get_offset()
447: (4)                def set_val(self, val):
448: (8)                    """
449: (8)                    Set slider value to *val*.
450: (8)                    Parameters
451: (8)                    ----------
452: (8)                    val : float
453: (8)                    """
454: (8)                    if self.orientation == 'vertical':
455: (12)                        self.poly.set_height(val - self.poly.get_y())
456: (12)                        self._handle.set_ydata([val])
457: (8)                    else:
458: (12)                        self.poly.set_width(val - self.poly.get_x())
459: (12)                        self._handle.set_xdata([val])
460: (8)                    self.valtext.set_text(self._format(val))
461: (8)                    if self.drawon:
462: (12)                        self.ax.figure.canvas.draw_idle()
463: (8)                    self.val = val
464: (8)                    if self.eventson:
465: (12)                        self._observers.process('changed', val)
466: (4)                def on_changed(self, func):
467: (8)                    """
468: (8)                    Connect *func* as callback function to changes of the slider value.
469: (8)                    Parameters
470: (8)                    ----------
471: (8)                    func : callable
472: (12)                        Function to call when slider is changed.
473: (12)                        The function must accept a single float as its arguments.
474: (8)                    Returns
475: (8)                    -------
476: (8)                    int
477: (12)                        Connection id (which can be used to disconnect *func*).
478: (8)                    """
479: (8)                    return self._observers.connect('changed', lambda val: func(val))
480: (0)            class RangeSlider(SliderBase):
481: (4)                """
482: (4)                A slider representing a range of floating point values. Defines the min
and
483: (4)                max of the range via the *val* attribute as a tuple of (min, max).
484: (4)                Create a slider that defines a range contained within [*valmin*, *valmax*]
485: (4)                in Axes *ax*. For the slider to remain responsive you must maintain a
486: (4)                reference to it. Call :meth:`on_changed` to connect to the slider event.
487: (4)                Attributes
488: (4)                ----------
489: (4)                val : tuple of float
490: (8)                    Slider value.
491: (4)                """
492: (4)                def __init__(
493: (8)                    self,
494: (8)                    ax,
495: (8)                    label,
```

```
(8)                     valmin,
(8)                     valmax,
(8)                     *,
(8)                     valinit=None,
(8)                     valfmt=None,
(8)                     closedmin=True,
(8)                     closedmax=True,
(8)                     dragging=True,
(8)                     valstep=None,
(8)                     orientation="horizontal",
(8)                     track_color='lightgrey',
(8)                     handle_style=None,
(8)                     **kwargs,
(4)                 ):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     ax : Axes
(12)                        The Axes to put the slider in.
(8)                     label : str
(12)                        Slider label.
(8)                     valmin : float
(12)                        The minimum value of the slider.
(8)                     valmax : float
(12)                        The maximum value of the slider.
(8)                     valinit : tuple of float or None, default: None
(12)                        The initial positions of the slider. If None the initial positions
(12)                        will be at the 25th and 75th percentiles of the range.
(8)                     valfmt : str, default: None
(12)                        %-format string used to format the slider values.  If None, a
(12)                        `.ScalarFormatter` is used instead.
(8)                     closedmin : bool, default: True
(12)                        Whether the slider interval is closed on the bottom.
(8)                     closedmax : bool, default: True
(12)                        Whether the slider interval is closed on the top.
(8)                     dragging : bool, default: True
(12)                        If True the slider can be dragged by the mouse.
(8)                     valstep : float, default: None
(12)                        If given, the slider will snap to multiples of *valstep*.
(8)                     orientation : {'horizontal', 'vertical'}, default: 'horizontal'
(12)                        The orientation of the slider.
(8)                     track_color : :mpltype:`color`, default: 'lightgrey'
(12)                        The color of the background track. The track is accessible for
(12)                        further styling via the *track* attribute.
(8)                     handle_style : dict
(12)                        Properties of the slider handles. Default values are
(12)                        ========= ===== ======= ========================================
(12)                        Key       Value Default Description
(12)                        ========= ===== ======= ========================================
(12)                        facecolor color 'white' The facecolor of the slider handles.
(12)                        edgecolor color '.75'   The edgecolor of the slider handles.
(12)                        size      int   10      The size of the slider handles in points.
(12)                        ========= ===== ======= ========================================
(12)                        Other values will be transformed as marker{foo} and passed to the
(12)                        `~.Line2D` constructor. e.g. ``handle_style = {'style'='x'}`` will
(12)                        result in ``markerstyle = 'x'``.
(8)                     Notes
(8)                     -----
(8)                     Additional kwargs are passed on to ``self.poly`` which is the
(8)                     `~matplotlib.patches.Polygon` that draws the slider knob.  See the
(8)                     `.Polygon` documentation for valid property names (``facecolor``,
(8)                     ``edgecolor``, ``alpha``, etc.).
(8)                     """
(8)                     super().__init__(ax, orientation, closedmin, closedmax,
(25)                                     valmin, valmax, valfmt, dragging, valstep)
(8)                     self.val = (valmin, valmax)
(8)                     if valinit is None:
(12)                        extent = valmax - valmin
(12)                        valinit = np.array([valmin + extent * 0.25,
```

```
(32)                                    valmin + extent * 0.75])
(8)                     else:
(12)                        valinit = self._value_in_bounds(valinit)
(8)                     self.val = valinit
(8)                     self.valinit = valinit
(8)                     defaults = {'facecolor': 'white', 'edgecolor': '.75', 'size': 10}
(8)                     handle_style = {} if handle_style is None else handle_style
(8)                     marker_props = {
(12)                        f'marker{k}': v for k, v in {**defaults, **handle_style}.items()
(8)                     }
(8)                     if orientation == "vertical":
(12)                        self.track = Rectangle(
(16)                            (.25, 0), .5, 2,
(16)                            transform=ax.transAxes,
(16)                            facecolor=track_color
(12)                        )
(12)                        ax.add_patch(self.track)
(12)                        poly_transform = self.ax.get_yaxis_transform(which="grid")
(12)                        handleXY_1 = [.5, valinit[0]]
(12)                        handleXY_2 = [.5, valinit[1]]
(8)                     else:
(12)                        self.track = Rectangle(
(16)                            (0, .25), 1, .5,
(16)                            transform=ax.transAxes,
(16)                            facecolor=track_color
(12)                        )
(12)                        ax.add_patch(self.track)
(12)                        poly_transform = self.ax.get_xaxis_transform(which="grid")
(12)                        handleXY_1 = [valinit[0], .5]
(12)                        handleXY_2 = [valinit[1], .5]
(8)                     self.poly = Polygon(np.zeros([5, 2]), **kwargs)
(8)                     self._update_selection_poly(*valinit)
(8)                     self.poly.set_transform(poly_transform)
(8)                     self.poly.get_path()._interpolation_steps = 100
(8)                     self.ax.add_patch(self.poly)
(8)                     self.ax._request_autoscale_view()
(8)                     self._handles = [
(12)                        ax.plot(
(16)                            *handleXY_1,
(16)                            "o",
(16)                            **marker_props,
(16)                            clip_on=False
(12)                        )[0],
(12)                        ax.plot(
(16)                            *handleXY_2,
(16)                            "o",
(16)                            **marker_props,
(16)                            clip_on=False
(12)                        )[0]
(8)                     ]
(8)                     if orientation == "vertical":
(12)                        self.label = ax.text(
(16)                            0.5,
(16)                            1.02,
(16)                            label,
(16)                            transform=ax.transAxes,
(16)                            verticalalignment="bottom",
(16)                            horizontalalignment="center",
(12)                        )
(12)                        self.valtext = ax.text(
(16)                            0.5,
(16)                            -0.02,
(16)                            self._format(valinit),
(16)                            transform=ax.transAxes,
(16)                            verticalalignment="top",
(16)                            horizontalalignment="center",
(12)                        )
(8)                     else:
(12)                        self.label = ax.text(
```

```
(16)                            -0.02,
(16)                            0.5,
(16)                            label,
(16)                            transform=ax.transAxes,
(16)                            verticalalignment="center",
(16)                            horizontalalignment="right",
(12)                        )
(12)                        self.valtext = ax.text(
(16)                            1.02,
(16)                            0.5,
(16)                            self._format(valinit),
(16)                            transform=ax.transAxes,
(16)                            verticalalignment="center",
(16)                            horizontalalignment="left",
(12)                        )
(8)                     self._active_handle = None
(8)                     self.set_val(valinit)
(4)                 def _update_selection_poly(self, vmin, vmax):
(8)                     """
(8)                     Update the vertices of the *self.poly* slider in-place
(8)                     to cover the data range *vmin*, *vmax*.
(8)                     """
(8)                     verts = self.poly.xy
(8)                     if self.orientation == "vertical":
(12)                        verts[0] = verts[4] = .25, vmin
(12)                        verts[1] = .25, vmax
(12)                        verts[2] = .75, vmax
(12)                        verts[3] = .75, vmin
(8)                     else:
(12)                        verts[0] = verts[4] = vmin, .25
(12)                        verts[1] = vmin, .75
(12)                        verts[2] = vmax, .75
(12)                        verts[3] = vmax, .25
(4)                 def _min_in_bounds(self, min):
(8)                     """Ensure the new min value is between valmin and self.val[1]."""
(8)                     if min <= self.valmin:
(12)                        if not self.closedmin:
(16)                            return self.val[0]
(12)                        min = self.valmin
(8)                     if min > self.val[1]:
(12)                        min = self.val[1]
(8)                     return self._stepped_value(min)
(4)                 def _max_in_bounds(self, max):
(8)                     """Ensure the new max value is between valmax and self.val[0]."""
(8)                     if max >= self.valmax:
(12)                        if not self.closedmax:
(16)                            return self.val[1]
(12)                        max = self.valmax
(8)                     if max <= self.val[0]:
(12)                        max = self.val[0]
(8)                     return self._stepped_value(max)
(4)                 def _value_in_bounds(self, vals):
(8)                     """Clip min, max values to the bounds."""
(8)                     return (self._min_in_bounds(vals[0]), self._max_in_bounds(vals[1]))
(4)                 def _update_val_from_pos(self, pos):
(8)                     """Update the slider value based on a given position."""
(8)                     idx = np.argmin(np.abs(self.val - pos))
(8)                     if idx == 0:
(12)                        val = self._min_in_bounds(pos)
(12)                        self.set_min(val)
(8)                     else:
(12)                        val = self._max_in_bounds(pos)
(12)                        self.set_max(val)
(8)                     if self._active_handle:
(12)                        if self.orientation == "vertical":
(16)                            self._active_handle.set_ydata([val])
(12)                        else:
(16)                            self._active_handle.set_xdata([val])
(4)                 def _update(self, event):
```

```
(8)                     """Update the slider position."""
(8)                     if self.ignore(event) or event.button != 1:
(12)                        return
(8)                     if event.name == "button_press_event" and self.ax.contains(event)[0]:
(12)                        self.drag_active = True
(12)                        event.canvas.grab_mouse(self.ax)
(8)                     if not self.drag_active:
(12)                        return
(8)                     if (event.name == "button_release_event"
(14)                          or event.name == "button_press_event" and not
self.ax.contains(event)[0]):
(12)                        self.drag_active = False
(12)                        event.canvas.release_mouse(self.ax)
(12)                        self._active_handle = None
(12)                        return
(8)                     xdata, ydata = self._get_data_coords(event)
(8)                     handle_index = np.argmin(np.abs(
(12)                        [h.get_xdata()[0] - xdata for h in self._handles]
(12)                        if self.orientation == "horizontal" else
(12)                        [h.get_ydata()[0] - ydata for h in self._handles]))
(8)                     handle = self._handles[handle_index]
(8)                     if handle is not self._active_handle:
(12)                        self._active_handle = handle
(8)                     self._update_val_from_pos(xdata if self.orientation == "horizontal"
else ydata)
(4)                 def _format(self, val):
(8)                     """Pretty-print *val*."""
(8)                     if self.valfmt is not None:
(12)                        return f"({self.valfmt % val[0]}, {self.valfmt % val[1]})"
(8)                     else:
(12)                        _, s1, s2, _ = self._fmt.format_ticks(
(16)                            [self.valmin, *val, self.valmax]
(12)                        )
(12)                        s1 += self._fmt.get_offset()
(12)                        s2 += self._fmt.get_offset()
(12)                        return f"({s1}, {s2})"
(4)                 def set_min(self, min):
(8)                     """
(8)                     Set the lower value of the slider to *min*.
(8)                     Parameters
(8)                     ----------
(8)                     min : float
(8)                     """
(8)                     self.set_val((min, self.val[1]))
(4)                 def set_max(self, max):
(8)                     """
(8)                     Set the lower value of the slider to *max*.
(8)                     Parameters
(8)                     ----------
(8)                     max : float
(8)                     """
(8)                     self.set_val((self.val[0], max))
(4)                 def set_val(self, val):
(8)                     """
(8)                     Set slider value to *val*.
(8)                     Parameters
(8)                     ----------
(8)                     val : tuple or array-like of float
(8)                     """
(8)                     val = np.sort(val)
(8)                     _api.check_shape((2,), val=val)
(8)                     self.val = (self.valmin, self.valmax)
(8)                     vmin, vmax = self._value_in_bounds(val)
(8)                     self._update_selection_poly(vmin, vmax)
(8)                     if self.orientation == "vertical":
(12)                        self._handles[0].set_ydata([vmin])
(12)                        self._handles[1].set_ydata([vmax])
(8)                     else:
(12)                        self._handles[0].set_xdata([vmin])
```

```
(12)                        self._handles[1].set_xdata([vmax])
(8)                     self.valtext.set_text(self._format((vmin, vmax)))
(8)                     if self.drawon:
(12)                        self.ax.figure.canvas.draw_idle()
(8)                     self.val = (vmin, vmax)
(8)                     if self.eventson:
(12)                        self._observers.process("changed", (vmin, vmax))
(4)                 def on_changed(self, func):
(8)                     """
(8)                     Connect *func* as callback function to changes of the slider value.
(8)                     Parameters
(8)                     ----------
(8)                     func : callable
(12)                        Function to call when slider is changed. The function
(12)                        must accept a 2-tuple of floats as its argument.
(8)                     Returns
(8)                     -------
(8)                     int
(12)                        Connection id (which can be used to disconnect *func*).
(8)                     """
(8)                     return self._observers.connect('changed', lambda val: func(val))
(0)             def _expand_text_props(props):
(4)                 props = cbook.normalize_kwargs(props, mtext.Text)
(4)                 return cycler(**props)() if props else itertools.repeat({})
(0)             class CheckButtons(AxesWidget):
(4)                 r"""
(4)                 A GUI neutral set of check buttons.
(4)                 For the check buttons to remain responsive you must keep a
(4)                 reference to this object.
(4)                 Connect to the CheckButtons with the `.on_clicked` method.
(4)                 Attributes
(4)                 ----------
(4)                 ax : `~matplotlib.axes.Axes`
(8)                     The parent Axes for the widget.
(4)                 labels : list of `~matplotlib.text.Text`
(8)                     The text label objects of the check buttons.
(4)                 """
(4)                 def __init__(self, ax, labels, actives=None, *, useblit=True,
(17)                         label_props=None, frame_props=None, check_props=None):
(8)                     """
(8)                     Add check buttons to `~.axes.Axes` instance *ax*.
(8)                     Parameters
(8)                     ----------
(8)                     ax : `~matplotlib.axes.Axes`
(12)                        The parent Axes for the widget.
(8)                     labels : list of str
(12)                        The labels of the check buttons.
(8)                     actives : list of bool, optional
(12)                        The initial check states of the buttons. The list must have the
(12)                        same length as *labels*. If not given, all buttons are unchecked.
(8)                     useblit : bool, default: True
(12)                        Use blitting for faster drawing if supported by the backend.
(12)                        See the tutorial :ref:`blitting` for details.
(12)                        .. versionadded:: 3.7
(8)                     label_props : dict, optional
(12)                        Dictionary of `.Text` properties to be used for the labels.
(12)                        .. versionadded:: 3.7
(8)                     frame_props : dict, optional
(12)                        Dictionary of scatter `.Collection` properties to be used for the
(12)                        check button frame. Defaults (label font size / 2)**2 size, black
(12)                        edgecolor, no facecolor, and 1.0 linewidth.
(12)                        .. versionadded:: 3.7
(8)                     check_props : dict, optional
(12)                        Dictionary of scatter `.Collection` properties to be used for the
(12)                        check button check. Defaults to (label font size / 2)**2 size,
(12)                        black color, and 1.0 linewidth.
(12)                        .. versionadded:: 3.7
(8)                     """
(8)                     super().__init__(ax)
```

```
839: (8)                    _api.check_isinstance((dict, None), label_props=label_props,
840: (30)                                          frame_props=frame_props,
check_props=check_props)
841: (8)                    ax.set_xticks([])
842: (8)                    ax.set_yticks([])
843: (8)                    ax.set_navigate(False)
844: (8)                    if actives is None:
845: (12)                       actives = [False] * len(labels)
846: (8)                    self._useblit = useblit and self.canvas.supports_blit
847: (8)                    self._background = None
848: (8)                    ys = np.linspace(1, 0, len(labels)+2)[1:-1]
849: (8)                    label_props = _expand_text_props(label_props)
850: (8)                    self.labels = [
851: (12)                       ax.text(0.25, y, label, transform=ax.transAxes,
852: (20)                               horizontalalignment="left", verticalalignment="center",
853: (20)                               **props)
854: (12)                       for y, label, props in zip(ys, labels, label_props)]
855: (8)                    text_size = np.array([text.get_fontsize() for text in self.labels]) /
2
856: (8)                    frame_props = {
857: (12)                       's': text_size**2,
858: (12)                       'linewidth': 1,
859: (12)                       **cbook.normalize_kwargs(frame_props, collections.PathCollection),
860: (12)                       'marker': 's',
861: (12)                       'transform': ax.transAxes,
862: (8)                    }
863: (8)                    frame_props.setdefault('facecolor', frame_props.get('color', 'none'))
864: (8)                    frame_props.setdefault('edgecolor', frame_props.pop('color', 'black'))
865: (8)                    self._frames = ax.scatter([0.15] * len(ys), ys, **frame_props)
866: (8)                    check_props = {
867: (12)                       'linewidth': 1,
868: (12)                       's': text_size**2,
869: (12)                       **cbook.normalize_kwargs(check_props, collections.PathCollection),
870: (12)                       'marker': 'x',
871: (12)                       'transform': ax.transAxes,
872: (12)                       'animated': self._useblit,
873: (8)                    }
874: (8)                    check_props.setdefault('facecolor', check_props.pop('color', 'black'))
875: (8)                    self._checks = ax.scatter([0.15] * len(ys), ys, **check_props)
876: (8)                    self._init_status(actives)
877: (8)                    self.connect_event('button_press_event', self._clicked)
878: (8)                    if self._useblit:
879: (12)                       self.connect_event('draw_event', self._clear)
880: (8)                    self._observers = cbook.CallbackRegistry(signals=["clicked"])
881: (4)                def _clear(self, event):
882: (8)                    """Internal event handler to clear the buttons."""
883: (8)                    if self.ignore(event) or self.canvas.is_saving():
884: (12)                       return
885: (8)                    self._background = self.canvas.copy_from_bbox(self.ax.bbox)
886: (8)                    self.ax.draw_artist(self._checks)
887: (4)                def _clicked(self, event):
888: (8)                    if self.ignore(event) or event.button != 1 or not
self.ax.contains(event)[0]:
889: (12)                       return
890: (8)                    idxs = [  # Indices of frames and of texts that contain the event.
891: (12)                       *self._frames.contains(event)[1]["ind"],
892: (12)                       *[i for i, text in enumerate(self.labels) if text.contains(event)
[0]]]
893: (8)                    if idxs:
894: (12)                       coords = self._frames.get_offset_transform().transform(
895: (16)                           self._frames.get_offsets())
896: (12)                       self.set_active(  # Closest index, only looking in idxs.
897: (16)                           idxs[(((event.x, event.y) - coords[idxs]) **
2).sum(-1).argmin()])
898: (4)                def set_label_props(self, props):
899: (8)                    """
900: (8)                    Set properties of the `.Text` labels.
901: (8)                    .. versionadded:: 3.7
902: (8)                    Parameters
```

```
(8)                     ----------
(8)                     props : dict
(12)                        Dictionary of `.Text` properties to be used for the labels.
(8)                     """
(8)                     _api.check_isinstance(dict, props=props)
(8)                     props = _expand_text_props(props)
(8)                     for text, prop in zip(self.labels, props):
(12)                        text.update(prop)
(4)                 def set_frame_props(self, props):
(8)                     """
(8)                     Set properties of the check button frames.
(8)                     .. versionadded:: 3.7
(8)                     Parameters
(8)                     ----------
(8)                     props : dict
(12)                        Dictionary of `.Collection` properties to be used for the check
(12)                        button frames.
(8)                     """
(8)                     _api.check_isinstance(dict, props=props)
(8)                     if 's' in props:  # Keep API consistent with constructor.
(12)                        props['sizes'] = np.broadcast_to(props.pop('s'), len(self.labels))
(8)                     self._frames.update(props)
(4)                 def set_check_props(self, props):
(8)                     """
(8)                     Set properties of the check button checks.
(8)                     .. versionadded:: 3.7
(8)                     Parameters
(8)                     ----------
(8)                     props : dict
(12)                        Dictionary of `.Collection` properties to be used for the check
(12)                        button check.
(8)                     """
(8)                     _api.check_isinstance(dict, props=props)
(8)                     if 's' in props:  # Keep API consistent with constructor.
(12)                        props['sizes'] = np.broadcast_to(props.pop('s'), len(self.labels))
(8)                     actives = self.get_status()
(8)                     self._checks.update(props)
(8)                     self._init_status(actives)
(4)                 def set_active(self, index, state=None):
(8)                     """
(8)                     Modify the state of a check button by index.
(8)                     Callbacks will be triggered if :attr:`eventson` is True.
(8)                     Parameters
(8)                     ----------
(8)                     index : int
(12)                        Index of the check button to toggle.
(8)                     state : bool, optional
(12)                        If a boolean value, set the state explicitly. If no value is
(12)                        provided, the state is toggled.
(8)                     Raises
(8)                     ------
(8)                     ValueError
(12)                        If *index* is invalid.
(8)                     TypeError
(12)                        If *state* is not boolean.
(8)                     """
(8)                     if index not in range(len(self.labels)):
(12)                        raise ValueError(f'Invalid CheckButton index: {index}')
(8)                     _api.check_isinstance((bool, None), state=state)
(8)                     invisible = colors.to_rgba('none')
(8)                     facecolors = self._checks.get_facecolor()
(8)                     if state is None:
(12)                        state = colors.same_color(facecolors[index], invisible)
(8)                     facecolors[index] = self._active_check_colors[index] if state else
invisible
(8)                     self._checks.set_facecolor(facecolors)
(8)                     if self.drawon:
(12)                        if self._useblit:
(16)                            if self._background is not None:
```

```
(20)                                self.canvas.restore_region(self._background)
(16)                            self.ax.draw_artist(self._checks)
(16)                            self.canvas.blit(self.ax.bbox)
(12)                        else:
(16)                            self.canvas.draw()
(8)                     if self.eventson:
(12)                        self._observers.process('clicked', self.labels[index].get_text())
(4)                 def _init_status(self, actives):
(8)                     """
(8)                     Initialize properties to match active status.
(8)                     The user may have passed custom colours in *check_props* to the
(8)                     constructor, or to `.set_check_props`, so we need to modify the
(8)                     visibility after getting whatever the user set.
(8)                     """
(8)                     self._active_check_colors = self._checks.get_facecolor()
(8)                     if len(self._active_check_colors) == 1:
(12)                        self._active_check_colors = np.repeat(self._active_check_colors,
(50)                                                              len(actives), axis=0)
(8)                     self._checks.set_facecolor(
(12)                        [ec if active else "none"
(13)                         for ec, active in zip(self._active_check_colors, actives)])
(4)                 def clear(self):
(8)                     """Uncheck all checkboxes."""
(8)                     self._checks.set_facecolor(['none'] * len(self._active_check_colors))
(8)                     if hasattr(self, '_lines'):
(12)                        for l1, l2 in self._lines:
(16)                            l1.set_visible(False)
(16)                            l2.set_visible(False)
(8)                     if self.drawon:
(12)                        self.canvas.draw()
(8)                     if self.eventson:
(12)                        self._observers.process('clicked', None)
(4)                 def get_status(self):
(8)                     """
(8)                     Return a list of the status (True/False) of all of the check buttons.
(8)                     """
(8)                     return [not colors.same_color(color, colors.to_rgba("none"))
(16)                            for color in self._checks.get_facecolors()]
(4)                 def get_checked_labels(self):
(8)                     """Return a list of labels currently checked by user."""
(8)                     return [l.get_text() for l, box_checked in
(16)                            zip(self.labels, self.get_status())
(16)                            if box_checked]
(4)                 def on_clicked(self, func):
(8)                     """
(8)                     Connect the callback function *func* to button click events.
(8)                     Parameters
(8)                     ----------
(8)                     func : callable
(12)                        When the button is clicked, call *func* with button label.
(12)                        When all buttons are cleared, call *func* with None.
(12)                        The callback func must have the signature::
(16)                            def func(label: str | None) -> Any
(12)                        Return values may exist, but are ignored.
(8)                     Returns
(8)                     -------
(8)                     A connection id, which can be used to disconnect the callback.
(8)                     """
(8)                     return self._observers.connect('clicked', lambda text: func(text))
(4)                 def disconnect(self, cid):
(8)                     """Remove the observer with connection id *cid*."""
(8)                     self._observers.disconnect(cid)
(0)             class TextBox(AxesWidget):
(4)                 """
(4)                 A GUI neutral text input box.
(4)                 For the text box to remain responsive you must keep a reference to it.
(4)                 Call `.on_text_change` to be updated whenever the text changes.
(4)                 Call `.on_submit` to be updated whenever the user hits enter or
(4)                 leaves the text entry field.
```

```
(4)                 Attributes
(4)                 ----------
(4)                 ax : `~matplotlib.axes.Axes`
(8)                     The parent Axes for the widget.
(4)                 label : `~matplotlib.text.Text`
(4)                 color : :mpltype:`color`
(8)                     The color of the text box when not hovering.
(4)                 hovercolor : :mpltype:`color`
(8)                     The color of the text box when hovering.
(4)                 """
(4)                 def __init__(self, ax, label, initial='', *,
(17)                         color='.95', hovercolor='1', label_pad=.01,
(17)                         textalignment="left"):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     ax : `~matplotlib.axes.Axes`
(12)                        The `~.axes.Axes` instance the button will be placed into.
(8)                     label : str
(12)                        Label for this text box.
(8)                     initial : str
(12)                        Initial value in the text box.
(8)                     color : :mpltype:`color`
(12)                        The color of the box.
(8)                     hovercolor : :mpltype:`color`
(12)                        The color of the box when the mouse is over it.
(8)                     label_pad : float
(12)                        The distance between the label and the right side of the textbox.
(8)                     textalignment : {'left', 'center', 'right'}
(12)                        The horizontal location of the text.
(8)                     """
(8)                     super().__init__(ax)
(8)                     self._text_position = _api.check_getitem(
(12)                        {"left": 0.05, "center": 0.5, "right": 0.95},
(12)                        textalignment=textalignment)
(8)                     self.label = ax.text(
(12)                        -label_pad, 0.5, label, transform=ax.transAxes,
(12)                        verticalalignment='center', horizontalalignment='right')
(8)                     self.text_disp = self.ax.text(
(12)                        self._text_position, 0.5, initial, transform=self.ax.transAxes,
(12)                        verticalalignment='center', horizontalalignment=textalignment,
(12)                        parse_math=False)
(8)                     self._observers = cbook.CallbackRegistry(signals=["change", "submit"])
(8)                     ax.set(
(12)                        xlim=(0, 1), ylim=(0, 1),  # s.t. cursor appears from first click.
(12)                        navigate=False, facecolor=color,
(12)                        xticks=[], yticks=[])
(8)                     self.cursor_index = 0
(8)                     self.cursor = ax.vlines(0, 0, 0, visible=False, color="k", lw=1,
(32)                                    transform=mpl.transforms.IdentityTransform())
(8)                     self.connect_event('button_press_event', self._click)
(8)                     self.connect_event('button_release_event', self._release)
(8)                     self.connect_event('motion_notify_event', self._motion)
(8)                     self.connect_event('key_press_event', self._keypress)
(8)                     self.connect_event('resize_event', self._resize)
(8)                     self.color = color
(8)                     self.hovercolor = hovercolor
(8)                     self.capturekeystrokes = False
(4)                 @property
(4)                 def text(self):
(8)                     return self.text_disp.get_text()
(4)                 def _rendercursor(self):
(8)                     if self.ax.figure._get_renderer() is None:
(12)                        self.ax.figure.canvas.draw()
(8)                     text = self.text_disp.get_text()  # Save value before overwriting it.
(8)                     widthtext = text[:self.cursor_index]
(8)                     bb_text = self.text_disp.get_window_extent()
(8)                     self.text_disp.set_text(widthtext or ",")
(8)                     bb_widthtext = self.text_disp.get_window_extent()
```

```
1109: (8)                    if bb_text.y0 == bb_text.y1:  # Restoring the height if no text.
1110: (12)                       bb_text.y0 -= bb_widthtext.height / 2
1111: (12)                       bb_text.y1 += bb_widthtext.height / 2
1112: (8)                    elif not widthtext:  # Keep width to 0.
1113: (12)                       bb_text.x1 = bb_text.x0
1114: (8)                    else:  # Move the cursor using width of bb_widthtext.
1115: (12)                       bb_text.x1 = bb_text.x0 + bb_widthtext.width
1116: (8)                    self.cursor.set(
1117: (12)                       segments=[[(bb_text.x1, bb_text.y0), (bb_text.x1, bb_text.y1)]],
1118: (12)                       visible=True)
1119: (8)                    self.text_disp.set_text(text)
1120: (8)                    self.ax.figure.canvas.draw()
1121: (4)                def _release(self, event):
1122: (8)                    if self.ignore(event):
1123: (12)                       return
1124: (8)                    if event.canvas.mouse_grabber != self.ax:
1125: (12)                       return
1126: (8)                    event.canvas.release_mouse(self.ax)
1127: (4)                def _keypress(self, event):
1128: (8)                    if self.ignore(event):
1129: (12)                       return
1130: (8)                    if self.capturekeystrokes:
1131: (12)                       key = event.key
1132: (12)                       text = self.text
1133: (12)                       if len(key) == 1:
1134: (16)                           text = (text[:self.cursor_index] + key +
1135: (24)                                   text[self.cursor_index:])
1136: (16)                           self.cursor_index += 1
1137: (12)                       elif key == "right":
1138: (16)                           if self.cursor_index != len(text):
1139: (20)                               self.cursor_index += 1
1140: (12)                       elif key == "left":
1141: (16)                           if self.cursor_index != 0:
1142: (20)                               self.cursor_index -= 1
1143: (12)                       elif key == "home":
1144: (16)                           self.cursor_index = 0
1145: (12)                       elif key == "end":
1146: (16)                           self.cursor_index = len(text)
1147: (12)                       elif key == "backspace":
1148: (16)                           if self.cursor_index != 0:
1149: (20)                               text = (text[:self.cursor_index - 1] +
1150: (28)                                       text[self.cursor_index:])
1151: (20)                               self.cursor_index -= 1
1152: (12)                       elif key == "delete":
1153: (16)                           if self.cursor_index != len(self.text):
1154: (20)                               text = (text[:self.cursor_index] +
1155: (28)                                       text[self.cursor_index + 1:])
1156: (12)                       self.text_disp.set_text(text)
1157: (12)                       self._rendercursor()
1158: (12)                       if self.eventson:
1159: (16)                           self._observers.process('change', self.text)
1160: (16)                           if key in ["enter", "return"]:
1161: (20)                               self._observers.process('submit', self.text)
1162: (4)                def set_val(self, val):
1163: (8)                    newval = str(val)
1164: (8)                    if self.text == newval:
1165: (12)                       return
1166: (8)                    self.text_disp.set_text(newval)
1167: (8)                    self._rendercursor()
1168: (8)                    if self.eventson:
1169: (12)                       self._observers.process('change', self.text)
1170: (12)                       self._observers.process('submit', self.text)
1171: (4)                def begin_typing(self):
1172: (8)                    self.capturekeystrokes = True
1173: (8)                    stack = ExitStack()  # Register cleanup actions when user stops
typing.
1174: (8)                    self._on_stop_typing = stack.close
1175: (8)                    toolmanager = getattr(
1176: (12)                       self.ax.figure.canvas.manager, "toolmanager", None)
```

```
(8)                 if toolmanager is not None:
(12)                    toolmanager.keypresslock(self)
(12)                    stack.callback(toolmanager.keypresslock.release, self)
(8)                 else:
(12)                    with _api.suppress_matplotlib_deprecation_warning():
(16)                        stack.enter_context(mpl.rc_context(
(20)                            {k: [] for k in mpl.rcParams if k.startswith("keymap.")}))
(4)             def stop_typing(self):
(8)                 if self.capturekeystrokes:
(12)                    self._on_stop_typing()
(12)                    self._on_stop_typing = None
(12)                    notifysubmit = True
(8)                 else:
(12)                    notifysubmit = False
(8)                 self.capturekeystrokes = False
(8)                 self.cursor.set_visible(False)
(8)                 self.ax.figure.canvas.draw()
(8)                 if notifysubmit and self.eventson:
(12)                    self._observers.process('submit', self.text)
(4)             def _click(self, event):
(8)                 if self.ignore(event):
(12)                    return
(8)                 if not self.ax.contains(event)[0]:
(12)                    self.stop_typing()
(12)                    return
(8)                 if not self.eventson:
(12)                    return
(8)                 if event.canvas.mouse_grabber != self.ax:
(12)                    event.canvas.grab_mouse(self.ax)
(8)                 if not self.capturekeystrokes:
(12)                    self.begin_typing()
(8)                 self.cursor_index = self.text_disp._char_index_at(event.x)
(8)                 self._rendercursor()
(4)             def _resize(self, event):
(8)                 self.stop_typing()
(4)             def _motion(self, event):
(8)                 if self.ignore(event):
(12)                    return
(8)                 c = self.hovercolor if self.ax.contains(event)[0] else self.color
(8)                 if not colors.same_color(c, self.ax.get_facecolor()):
(12)                    self.ax.set_facecolor(c)
(12)                    if self.drawon:
(16)                        self.ax.figure.canvas.draw()
(4)             def on_text_change(self, func):
(8)                 """
(8)                 When the text changes, call this *func* with event.
(8)                 A connection id is returned which can be used to disconnect.
(8)                 """
(8)                 return self._observers.connect('change', lambda text: func(text))
(4)             def on_submit(self, func):
(8)                 """
(8)                 When the user hits enter or leaves the submission box, call this
(8)                 *func* with event.
(8)                 A connection id is returned which can be used to disconnect.
(8)                 """
(8)                 return self._observers.connect('submit', lambda text: func(text))
(4)             def disconnect(self, cid):
(8)                 """Remove the observer with connection id *cid*."""
(8)                 self._observers.disconnect(cid)
(0)         class RadioButtons(AxesWidget):
(4)             """
(4)             A GUI neutral radio button.
(4)             For the buttons to remain responsive you must keep a reference to this
(4)             object.
(4)             Connect to the RadioButtons with the `.on_clicked` method.
(4)             Attributes
(4)             ----------
(4)             ax : `~matplotlib.axes.Axes`
(8)                 The parent Axes for the widget.
```

```
    activecolor : :mpltype:`color`
        The color of the selected button.
    labels : list of `.Text`
        The button labels.
    value_selected : str
        The label text of the currently selected button.
    index_selected : int
        The index of the selected button.
    """
    def __init__(self, ax, labels, active=0, activecolor=None, *,
                 useblit=True, label_props=None, radio_props=None):
        """
        Add radio buttons to an `~.axes.Axes`.
        Parameters
        ----------
        ax : `~matplotlib.axes.Axes`
            The Axes to add the buttons to.
        labels : list of str
            The button labels.
        active : int
            The index of the initially selected button.
        activecolor : :mpltype:`color`
            The color of the selected button. The default is ``'blue'`` if not
            specified here or in *radio_props*.
        useblit : bool, default: True
            Use blitting for faster drawing if supported by the backend.
            See the tutorial :ref:`blitting` for details.
            .. versionadded:: 3.7
        label_props : dict or list of dict, optional
            Dictionary of `.Text` properties to be used for the labels.
            .. versionadded:: 3.7
        radio_props : dict, optional
            Dictionary of scatter `.Collection` properties to be used for the
            radio buttons. Defaults to (label font size / 2)**2 size, black
            edgecolor, and *activecolor* facecolor (when active).
            .. note::
                If a facecolor is supplied in *radio_props*, it will override
                *activecolor*. This may be used to provide an active color per
                button.
            .. versionadded:: 3.7
        """
        super().__init__(ax)
        _api.check_isinstance((dict, None), label_props=label_props,
                              radio_props=radio_props)
        radio_props = cbook.normalize_kwargs(radio_props,
                                             collections.PathCollection)
        if activecolor is not None:
            if 'facecolor' in radio_props:
                _api.warn_external(
                    'Both the *activecolor* parameter and the *facecolor* '
                    'key in the *radio_props* parameter has been specified. '
                    '*activecolor* will be ignored.')
        else:
            activecolor = 'blue'  # Default.
        self._activecolor = activecolor
        self._initial_active = active
        self.value_selected = labels[active]
        self.index_selected = active
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_navigate(False)
        ys = np.linspace(1, 0, len(labels) + 2)[1:-1]
        self._useblit = useblit and self.canvas.supports_blit
        self._background = None
        label_props = _expand_text_props(label_props)
        self.labels = [
            ax.text(0.25, y, label, transform=ax.transAxes,
                    horizontalalignment="left", verticalalignment="center",
                    **props)
```

```
1315: (12)                      for y, label, props in zip(ys, labels, label_props)]
1316: (8)                   text_size = np.array([text.get_fontsize() for text in self.labels]) /
2
1317: (8)                   radio_props = {
1318: (12)                      's': text_size**2,
1319: (12)                      **radio_props,
1320: (12)                      'marker': 'o',
1321: (12)                      'transform': ax.transAxes,
1322: (12)                      'animated': self._useblit,
1323: (8)                   }
1324: (8)                   radio_props.setdefault('edgecolor', radio_props.get('color', 'black'))
1325: (8)                   radio_props.setdefault('facecolor',
1326: (31)                                          radio_props.pop('color', activecolor))
1327: (8)                   self._buttons = ax.scatter([.15] * len(ys), ys, **radio_props)
1328: (8)                   self._active_colors = self._buttons.get_facecolor()
1329: (8)                   if len(self._active_colors) == 1:
1330: (12)                      self._active_colors = np.repeat(self._active_colors, len(labels),
1331: (44)                                                      axis=0)
1332: (8)                   self._buttons.set_facecolor(
1333: (12)                      [activecolor if i == active else "none"
1334: (13)                       for i, activecolor in enumerate(self._active_colors)])
1335: (8)                   self.connect_event('button_press_event', self._clicked)
1336: (8)                   if self._useblit:
1337: (12)                      self.connect_event('draw_event', self._clear)
1338: (8)                   self._observers = cbook.CallbackRegistry(signals=["clicked"])
1339: (4)               def _clear(self, event):
1340: (8)                   """Internal event handler to clear the buttons."""
1341: (8)                   if self.ignore(event) or self.canvas.is_saving():
1342: (12)                      return
1343: (8)                   self._background = self.canvas.copy_from_bbox(self.ax.bbox)
1344: (8)                   self.ax.draw_artist(self._buttons)
1345: (4)               def _clicked(self, event):
1346: (8)                   if self.ignore(event) or event.button != 1 or not
self.ax.contains(event)[0]:
1347: (12)                      return
1348: (8)                   idxs = [  # Indices of buttons and of texts that contain the event.
1349: (12)                      *self._buttons.contains(event)[1]["ind"],
1350: (12)                      *[i for i, text in enumerate(self.labels) if text.contains(event)
[0]]]
1351: (8)                   if idxs:
1352: (12)                      coords = self._buttons.get_offset_transform().transform(
1353: (16)                          self._buttons.get_offsets())
1354: (12)                      self.set_active(  # Closest index, only looking in idxs.
1355: (16)                          idxs[(((event.x, event.y) - coords[idxs]) **
2).sum(-1).argmin()])
1356: (4)               def set_label_props(self, props):
1357: (8)                   """
1358: (8)                   Set properties of the `.Text` labels.
1359: (8)                   .. versionadded:: 3.7
1360: (8)                   Parameters
1361: (8)                   ----------
1362: (8)                   props : dict
1363: (12)                      Dictionary of `.Text` properties to be used for the labels.
1364: (8)                   """
1365: (8)                   _api.check_isinstance(dict, props=props)
1366: (8)                   props = _expand_text_props(props)
1367: (8)                   for text, prop in zip(self.labels, props):
1368: (12)                      text.update(prop)
1369: (4)               def set_radio_props(self, props):
1370: (8)                   """
1371: (8)                   Set properties of the `.Text` labels.
1372: (8)                   .. versionadded:: 3.7
1373: (8)                   Parameters
1374: (8)                   ----------
1375: (8)                   props : dict
1376: (12)                      Dictionary of `.Collection` properties to be used for the radio
1377: (12)                      buttons.
1378: (8)                   """
1379: (8)                   _api.check_isinstance(dict, props=props)
```

```
1380: (8)                    if 's' in props:  # Keep API consistent with constructor.
1381: (12)                       props['sizes'] = np.broadcast_to(props.pop('s'), len(self.labels))
1382: (8)                    self._buttons.update(props)
1383: (8)                    self._active_colors = self._buttons.get_facecolor()
1384: (8)                    if len(self._active_colors) == 1:
1385: (12)                       self._active_colors = np.repeat(self._active_colors,
1386: (44)                                                       len(self.labels), axis=0)
1387: (8)                    self._buttons.set_facecolor(
1388: (12)                       [activecolor if text.get_text() == self.value_selected else "none"
1389: (13)                        for text, activecolor in zip(self.labels, self._active_colors)])
1390: (4)                @property
1391: (4)                def activecolor(self):
1392: (8)                    return self._activecolor
1393: (4)                @activecolor.setter
1394: (4)                def activecolor(self, activecolor):
1395: (8)                    colors._check_color_like(activecolor=activecolor)
1396: (8)                    self._activecolor = activecolor
1397: (8)                    self.set_radio_props({'facecolor': activecolor})
1398: (4)                def set_active(self, index):
1399: (8)                    """
1400: (8)                    Select button with number *index*.
1401: (8)                    Callbacks will be triggered if :attr:`eventson` is True.
1402: (8)                    Parameters
1403: (8)                    ----------
1404: (8)                    index : int
1405: (12)                       The index of the button to activate.
1406: (8)                    Raises
1407: (8)                    ------
1408: (8)                    ValueError
1409: (12)                       If the index is invalid.
1410: (8)                    """
1411: (8)                    if index not in range(len(self.labels)):
1412: (12)                       raise ValueError(f'Invalid RadioButton index: {index}')
1413: (8)                    self.value_selected = self.labels[index].get_text()
1414: (8)                    self.index_selected = index
1415: (8)                    button_facecolors = self._buttons.get_facecolor()
1416: (8)                    button_facecolors[:] = colors.to_rgba("none")
1417: (8)                    button_facecolors[index] = colors.to_rgba(self._active_colors[index])
1418: (8)                    self._buttons.set_facecolor(button_facecolors)
1419: (8)                    if self.drawon:
1420: (12)                       if self._useblit:
1421: (16)                           if self._background is not None:
1422: (20)                               self.canvas.restore_region(self._background)
1423: (16)                           self.ax.draw_artist(self._buttons)
1424: (16)                           self.canvas.blit(self.ax.bbox)
1425: (12)                       else:
1426: (16)                           self.canvas.draw()
1427: (8)                    if self.eventson:
1428: (12)                       self._observers.process('clicked', self.labels[index].get_text())
1429: (4)                def clear(self):
1430: (8)                    """Reset the active button to the initially active one."""
1431: (8)                    self.set_active(self._initial_active)
1432: (4)                def on_clicked(self, func):
1433: (8)                    """
1434: (8)                    Connect the callback function *func* to button click events.
1435: (8)                    Parameters
1436: (8)                    ----------
1437: (8)                    func : callable
1438: (12)                       When the button is clicked, call *func* with button label.
1439: (12)                       When all buttons are cleared, call *func* with None.
1440: (12)                       The callback func must have the signature::
1441: (16)                           def func(label: str | None) -> Any
1442: (12)                       Return values may exist, but are ignored.
1443: (8)                    Returns
1444: (8)                    -------
1445: (8)                    A connection id, which can be used to disconnect the callback.
1446: (8)                    """
1447: (8)                    return self._observers.connect('clicked', func)
1448: (4)                def disconnect(self, cid):
```

```
1449: (8)                    """Remove the observer with connection id *cid*."""
1450: (8)                    self._observers.disconnect(cid)
1451: (0)            class SubplotTool(Widget):
1452: (4)                """
1453: (4)                A tool to adjust the subplot params of a `.Figure`.
1454: (4)                """
1455: (4)                def __init__(self, targetfig, toolfig):
1456: (8)                    """
1457: (8)                    Parameters
1458: (8)                    ----------
1459: (8)                    targetfig : `~matplotlib.figure.Figure`
1460: (12)                       The figure instance to adjust.
1461: (8)                    toolfig : `~matplotlib.figure.Figure`
1462: (12)                       The figure instance to embed the subplot tool into.
1463: (8)                    """
1464: (8)                    self.figure = toolfig
1465: (8)                    self.targetfig = targetfig
1466: (8)                    toolfig.subplots_adjust(left=0.2, right=0.9)
1467: (8)                    toolfig.suptitle("Click on slider to adjust subplot param")
1468: (8)                    self._sliders = []
1469: (8)                    names = ["left", "bottom", "right", "top", "wspace", "hspace"]
1470: (8)                    for name, ax in zip(names, toolfig.subplots(len(names) + 1)):
1471: (12)                       ax.set_navigate(False)
1472: (12)                       slider = Slider(ax, name, 0, 1,
1473: (28)                                       valinit=getattr(targetfig.subplotpars, name))
1474: (12)                       slider.on_changed(self._on_slider_changed)
1475: (12)                       self._sliders.append(slider)
1476: (8)                    toolfig.axes[-1].remove()
1477: (8)                    (self.sliderleft, self.sliderbottom, self.sliderright, self.slidertop,
1478: (9)                     self.sliderwspace, self.sliderhspace) = self._sliders
1479: (8)                    for slider in [self.sliderleft, self.sliderbottom,
1480: (23)                                  self.sliderwspace, self.sliderhspace]:
1481: (12)                       slider.closedmax = False
1482: (8)                    for slider in [self.sliderright, self.slidertop]:
1483: (12)                       slider.closedmin = False
1484: (8)                    self.sliderleft.slidermax = self.sliderright
1485: (8)                    self.sliderright.slidermin = self.sliderleft
1486: (8)                    self.sliderbottom.slidermax = self.slidertop
1487: (8)                    self.slidertop.slidermin = self.sliderbottom
1488: (8)                    bax = toolfig.add_axes([0.8, 0.05, 0.15, 0.075])
1489: (8)                    self.buttonreset = Button(bax, 'Reset')
1490: (8)                    self.buttonreset.on_clicked(self._on_reset)
1491: (4)                def _on_slider_changed(self, _):
1492: (8)                    self.targetfig.subplots_adjust(
1493: (12)                       **{slider.label.get_text(): slider.val
1494: (15)                          for slider in self._sliders})
1495: (8)                    if self.drawon:
1496: (12)                       self.targetfig.canvas.draw()
1497: (4)                def _on_reset(self, event):
1498: (8)                    with ExitStack() as stack:
1499: (12)                       stack.enter_context(cbook._setattr_cm(self, drawon=False))
1500: (12)                       for slider in self._sliders:
1501: (16)                           stack.enter_context(
1502: (20)                               cbook._setattr_cm(slider, drawon=False, eventson=False))
1503: (12)                       for slider in self._sliders:
1504: (16)                           slider.reset()
1505: (8)                    if self.drawon:
1506: (12)                       event.canvas.draw()  # Redraw the subplottool canvas.
1507: (8)                    self._on_slider_changed(None)  # Apply changes to the target window.
1508: (0)            class Cursor(AxesWidget):
1509: (4)                """
1510: (4)                A crosshair cursor that spans the Axes and moves with mouse cursor.
1511: (4)                For the cursor to remain responsive you must keep a reference to it.
1512: (4)                Parameters
1513: (4)                ----------
1514: (4)                ax : `~matplotlib.axes.Axes`
1515: (8)                    The `~.axes.Axes` to attach the cursor to.
1516: (4)                horizOn : bool, default: True
1517: (8)                    Whether to draw the horizontal line.
```

```
(4)                 vertOn : bool, default: True
(8)                     Whether to draw the vertical line.
(4)                 useblit : bool, default: False
(8)                     Use blitting for faster drawing if supported by the backend.
(8)                     See the tutorial :ref:`blitting` for details.
(4)                 Other Parameters
(4)                 ----------------
(4)                 **lineprops
(8)                     `.Line2D` properties that control the appearance of the lines.
(8)                     See also `~.Axes.axhline`.
(4)                 Examples
(4)                 --------
(4)                 See :doc:`/gallery/widgets/cursor`.
(4)                 """
(4)                 def __init__(self, ax, *, horizOn=True, vertOn=True, useblit=False,
(17)                             **lineprops):
(8)                     super().__init__(ax)
(8)                     self.connect_event('motion_notify_event', self.onmove)
(8)                     self.connect_event('draw_event', self.clear)
(8)                     self.visible = True
(8)                     self.horizOn = horizOn
(8)                     self.vertOn = vertOn
(8)                     self.useblit = useblit and self.canvas.supports_blit
(8)                     if self.useblit:
(12)                        lineprops['animated'] = True
(8)                     self.lineh = ax.axhline(ax.get_ybound()[0], visible=False,
**lineprops)
(8)                     self.linev = ax.axvline(ax.get_xbound()[0], visible=False,
**lineprops)
(8)                     self.background = None
(8)                     self.needclear = False
(4)                 def clear(self, event):
(8)                     """Internal event handler to clear the cursor."""
(8)                     if self.ignore(event) or self.canvas.is_saving():
(12)                        return
(8)                     if self.useblit:
(12)                        self.background = self.canvas.copy_from_bbox(self.ax.bbox)
(4)                 def onmove(self, event):
(8)                     """Internal event handler to draw the cursor when the mouse moves."""
(8)                     if self.ignore(event):
(12)                        return
(8)                     if not self.canvas.widgetlock.available(self):
(12)                        return
(8)                     if not self.ax.contains(event)[0]:
(12)                        self.linev.set_visible(False)
(12)                        self.lineh.set_visible(False)
(12)                        if self.needclear:
(16)                            self.canvas.draw()
(16)                            self.needclear = False
(12)                        return
(8)                     self.needclear = True
(8)                     xdata, ydata = self._get_data_coords(event)
(8)                     self.linev.set_xdata((xdata, xdata))
(8)                     self.linev.set_visible(self.visible and self.vertOn)
(8)                     self.lineh.set_ydata((ydata, ydata))
(8)                     self.lineh.set_visible(self.visible and self.horizOn)
(8)                     if not (self.visible and (self.vertOn or self.horizOn)):
(12)                        return
(8)                     if self.useblit:
(12)                        if self.background is not None:
(16)                            self.canvas.restore_region(self.background)
(12)                        self.ax.draw_artist(self.linev)
(12)                        self.ax.draw_artist(self.lineh)
(12)                        self.canvas.blit(self.ax.bbox)
(8)                     else:
(12)                        self.canvas.draw_idle()
(0)             class MultiCursor(Widget):
(4)                 """
(4)                 Provide a vertical (default) and/or horizontal line cursor shared between
```

```
(4)                multiple Axes.
(4)                For the cursor to remain responsive you must keep a reference to it.
(4)                Parameters
(4)                ----------
(4)                canvas : object
(8)                    This parameter is entirely unused and only kept for back-compatibility.
(4)                axes : list of `~matplotlib.axes.Axes`
(8)                    The `~.axes.Axes` to attach the cursor to.
(4)                useblit : bool, default: True
(8)                    Use blitting for faster drawing if supported by the backend.
(8)                    See the tutorial :ref:`blitting`
(8)                    for details.
(4)                horizOn : bool, default: False
(8)                    Whether to draw the horizontal line.
(4)                vertOn : bool, default: True
(8)                    Whether to draw the vertical line.
(4)                Other Parameters
(4)                ----------------
(4)                **lineprops
(8)                    `.Line2D` properties that control the appearance of the lines.
(8)                    See also `~.Axes.axhline`.
(4)                Examples
(4)                --------
(4)                See :doc:`/gallery/widgets/multicursor`.
(4)                """
(4)                def __init__(self, canvas, axes, *, useblit=True, horizOn=False, vertOn=True,
(17)                            **lineprops):
(8)                    self._canvas = canvas
(8)                    self.axes = axes
(8)                    self.horizOn = horizOn
(8)                    self.vertOn = vertOn
(8)                    self._canvas_infos = {
(12)                       ax.figure.canvas: {"cids": [], "background": None} for ax in axes}
(8)                    xmin, xmax = axes[-1].get_xlim()
(8)                    ymin, ymax = axes[-1].get_ylim()
(8)                    xmid = 0.5 * (xmin + xmax)
(8)                    ymid = 0.5 * (ymin + ymax)
(8)                    self.visible = True
(8)                    self.useblit = (
(12)                       useblit
(12)                       and all(canvas.supports_blit for canvas in self._canvas_infos))
(8)                    if self.useblit:
(12)                       lineprops['animated'] = True
(8)                    self.vlines = [ax.axvline(xmid, visible=False, **lineprops)
(23)                                  for ax in axes]
(8)                    self.hlines = [ax.axhline(ymid, visible=False, **lineprops)
(23)                                  for ax in axes]
(8)                    self.connect()
(4)                def connect(self):
(8)                    """Connect events."""
(8)                    for canvas, info in self._canvas_infos.items():
(12)                       info["cids"] = [
(16)                           canvas.mpl_connect('motion_notify_event', self.onmove),
(16)                           canvas.mpl_connect('draw_event', self.clear),
(12)                       ]
(4)                def disconnect(self):
(8)                    """Disconnect events."""
(8)                    for canvas, info in self._canvas_infos.items():
(12)                       for cid in info["cids"]:
(16)                           canvas.mpl_disconnect(cid)
(12)                       info["cids"].clear()
(4)                def clear(self, event):
(8)                    """Clear the cursor."""
(8)                    if self.ignore(event):
(12)                       return
(8)                    if self.useblit:
(12)                       for canvas, info in self._canvas_infos.items():
```

```
1652: (16)                          if canvas is not canvas.figure.canvas:
1653: (20)                              continue
1654: (16)                          info["background"] = canvas.copy_from_bbox(canvas.figure.bbox)
1655: (4)               def onmove(self, event):
1656: (8)                   axs = [ax for ax in self.axes if ax.contains(event)[0]]
1657: (8)                   if self.ignore(event) or not axs or not
event.canvas.widgetlock.available(self):
1658: (12)                      return
1659: (8)                   ax = cbook._topmost_artist(axs)
1660: (8)                   xdata, ydata = ((event.xdata, event.ydata) if event.inaxes is ax
1661: (24)                                  else ax.transData.inverted().transform((event.x,
event.y)))
1662: (8)                   for line in self.vlines:
1663: (12)                      line.set_xdata((xdata, xdata))
1664: (12)                      line.set_visible(self.visible and self.vertOn)
1665: (8)                   for line in self.hlines:
1666: (12)                      line.set_ydata((ydata, ydata))
1667: (12)                      line.set_visible(self.visible and self.horizOn)
1668: (8)                   if not (self.visible and (self.vertOn or self.horizOn)):
1669: (12)                      return
1670: (8)                   if self.useblit:
1671: (12)                      for canvas, info in self._canvas_infos.items():
1672: (16)                          if info["background"]:
1673: (20)                              canvas.restore_region(info["background"])
1674: (12)                      if self.vertOn:
1675: (16)                          for ax, line in zip(self.axes, self.vlines):
1676: (20)                              ax.draw_artist(line)
1677: (12)                      if self.horizOn:
1678: (16)                          for ax, line in zip(self.axes, self.hlines):
1679: (20)                              ax.draw_artist(line)
1680: (12)                      for canvas in self._canvas_infos:
1681: (16)                          canvas.blit()
1682: (8)                   else:
1683: (12)                      for canvas in self._canvas_infos:
1684: (16)                          canvas.draw_idle()
1685: (0)           class _SelectorWidget(AxesWidget):
1686: (4)               def __init__(self, ax, onselect, useblit=False, button=None,
1687: (17)                           state_modifier_keys=None, use_data_coordinates=False):
1688: (8)                   super().__init__(ax)
1689: (8)                   self._visible = True
1690: (8)                   self.onselect = onselect
1691: (8)                   self.useblit = useblit and self.canvas.supports_blit
1692: (8)                   self.connect_default_events()
1693: (8)                   self._state_modifier_keys = dict(move=' ', clear='escape',
1694: (41)                                                   square='shift', center='control',
1695: (41)                                                   rotate='r')
1696: (8)                   self._state_modifier_keys.update(state_modifier_keys or {})
1697: (8)                   self._use_data_coordinates = use_data_coordinates
1698: (8)                   self.background = None
1699: (8)                   if isinstance(button, Integral):
1700: (12)                      self.validButtons = [button]
1701: (8)                   else:
1702: (12)                      self.validButtons = button
1703: (8)                   self._selection_completed = False
1704: (8)                   self._eventpress = None
1705: (8)                   self._eventrelease = None
1706: (8)                   self._prev_event = None
1707: (8)                   self._state = set()
1708: (4)               def set_active(self, active):
1709: (8)                   super().set_active(active)
1710: (8)                   if active:
1711: (12)                      self.update_background(None)
1712: (4)               def _get_animated_artists(self):
1713: (8)                   """
1714: (8)                   Convenience method to get all animated artists of the figure
containing
1715: (8)                   this widget, excluding those already present in self.artists.
1716: (8)                   The returned tuple is not sorted by 'z_order': z_order sorting is
1717: (8)                   valid only when considering all artists and not only a subset of all
```

```
(8)                 artists.
(8)                 """
(8)                 return tuple(a for ax_ in self.ax.get_figure().get_axes()
(21)                             for a in ax_.get_children()
(21)                             if a.get_animated() and a not in self.artists)
(4)             def update_background(self, event):
(8)                 """Force an update of the background."""
(8)                 if not self.useblit:
(12)                    return
(8)                 artists = sorted(self.artists + self._get_animated_artists(),
(25)                                 key=lambda a: a.get_zorder())
(8)                 needs_redraw = any(artist.get_visible() for artist in artists)
(8)                 with ExitStack() as stack:
(12)                    if needs_redraw:
(16)                        for artist in artists:
(20)                            stack.enter_context(artist._cm_set(visible=False))
(16)                        self.canvas.draw()
(12)                    self.background = self.canvas.copy_from_bbox(self.ax.bbox)
(8)                 if needs_redraw:
(12)                    for artist in artists:
(16)                        self.ax.draw_artist(artist)
(4)             def connect_default_events(self):
(8)                 """Connect the major canvas events to methods."""
(8)                 self.connect_event('motion_notify_event', self.onmove)
(8)                 self.connect_event('button_press_event', self.press)
(8)                 self.connect_event('button_release_event', self.release)
(8)                 self.connect_event('draw_event', self.update_background)
(8)                 self.connect_event('key_press_event', self.on_key_press)
(8)                 self.connect_event('key_release_event', self.on_key_release)
(8)                 self.connect_event('scroll_event', self.on_scroll)
(4)             def ignore(self, event):
(8)                 if not self.active or not self.ax.get_visible():
(12)                    return True
(8)                 if not self.canvas.widgetlock.available(self):
(12)                    return True
(8)                 if not hasattr(event, 'button'):
(12)                    event.button = None
(8)                 if (self.validButtons is not None
(16)                        and event.button not in self.validButtons):
(12)                    return True
(8)                 if self._eventpress is None:
(12)                    return not self.ax.contains(event)[0]
(8)                 if event.button == self._eventpress.button:
(12)                    return False
(8)                 return (not self.ax.contains(event)[0] or
(16)                        event.button != self._eventpress.button)
(4)             def update(self):
(8)                 """Draw using blit() or draw_idle(), depending on ``self.useblit``."""
(8)                 if (not self.ax.get_visible() or
(16)                        self.ax.figure._get_renderer() is None):
(12)                    return
(8)                 if self.useblit:
(12)                    if self.background is not None:
(16)                        self.canvas.restore_region(self.background)
(12)                    else:
(16)                        self.update_background(None)
(12)                    artists = sorted(self.artists + self._get_animated_artists(),
(29)                                     key=lambda a: a.get_zorder())
(12)                    for artist in artists:
(16)                        self.ax.draw_artist(artist)
(12)                    self.canvas.blit(self.ax.bbox)
(8)                 else:
(12)                    self.canvas.draw_idle()
(4)             def _get_data(self, event):
(8)                 """Get the xdata and ydata for event, with limits."""
(8)                 if event.xdata is None:
(12)                    return None, None
(8)                 xdata, ydata = self._get_data_coords(event)
(8)                 xdata = np.clip(xdata, *self.ax.get_xbound())
```

```
1787: (8)                    ydata = np.clip(ydata, *self.ax.get_ybound())
1788: (8)                    return xdata, ydata
1789: (4)                def _clean_event(self, event):
1790: (8)                    """
1791: (8)                    Preprocess an event:
1792: (8)                    - Replace *event* by the previous event if *event* has no ``xdata``.
1793: (8)                    - Get ``xdata`` and ``ydata`` from this widget's Axes, and clip them
to the axes
1794: (10)                     limits.
1795: (8)                    - Update the previous event.
1796: (8)                    """
1797: (8)                    if event.xdata is None:
1798: (12)                       event = self._prev_event
1799: (8)                    else:
1800: (12)                       event = copy.copy(event)
1801: (8)                    event.xdata, event.ydata = self._get_data(event)
1802: (8)                    self._prev_event = event
1803: (8)                    return event
1804: (4)                def press(self, event):
1805: (8)                    """Button press handler and validator."""
1806: (8)                    if not self.ignore(event):
1807: (12)                       event = self._clean_event(event)
1808: (12)                       self._eventpress = event
1809: (12)                       self._prev_event = event
1810: (12)                       key = event.key or ''
1811: (12)                       key = key.replace('ctrl', 'control')
1812: (12)                       if key == self._state_modifier_keys['move']:
1813: (16)                           self._state.add('move')
1814: (12)                       self._press(event)
1815: (12)                       return True
1816: (8)                    return False
1817: (4)                def _press(self, event):
1818: (8)                    """Button press event handler."""
1819: (4)                def release(self, event):
1820: (8)                    """Button release event handler and validator."""
1821: (8)                    if not self.ignore(event) and self._eventpress:
1822: (12)                       event = self._clean_event(event)
1823: (12)                       self._eventrelease = event
1824: (12)                       self._release(event)
1825: (12)                       self._eventpress = None
1826: (12)                       self._eventrelease = None
1827: (12)                       self._state.discard('move')
1828: (12)                       return True
1829: (8)                    return False
1830: (4)                def _release(self, event):
1831: (8)                    """Button release event handler."""
1832: (4)                def onmove(self, event):
1833: (8)                    """Cursor move event handler and validator."""
1834: (8)                    if not self.ignore(event) and self._eventpress:
1835: (12)                       event = self._clean_event(event)
1836: (12)                       self._onmove(event)
1837: (12)                       return True
1838: (8)                    return False
1839: (4)                def _onmove(self, event):
1840: (8)                    """Cursor move event handler."""
1841: (4)                def on_scroll(self, event):
1842: (8)                    """Mouse scroll event handler and validator."""
1843: (8)                    if not self.ignore(event):
1844: (12)                       self._on_scroll(event)
1845: (4)                def _on_scroll(self, event):
1846: (8)                    """Mouse scroll event handler."""
1847: (4)                def on_key_press(self, event):
1848: (8)                    """Key press event handler and validator for all selection widgets."""
1849: (8)                    if self.active:
1850: (12)                       key = event.key or ''
1851: (12)                       key = key.replace('ctrl', 'control')
1852: (12)                       if key == self._state_modifier_keys['clear']:
1853: (16)                           self.clear()
1854: (16)                           return
```

```
(12)                        for (state, modifier) in self._state_modifier_keys.items():
(16)                            if modifier in key.split('+'):
(20)                                if state == 'rotate':
(24)                                    if state in self._state:
(28)                                        self._state.discard(state)
(24)                                    else:
(28)                                        self._state.add(state)
(20)                                else:
(24)                                    self._state.add(state)
(12)                        self._on_key_press(event)
(4)                 def _on_key_press(self, event):
(8)                     """Key press event handler - for widget-specific key press actions."""
(4)                 def on_key_release(self, event):
(8)                     """Key release event handler and validator."""
(8)                     if self.active:
(12)                        key = event.key or ''
(12)                        for (state, modifier) in self._state_modifier_keys.items():
(16)                            if modifier in key.split('+') and state != 'rotate':
(20)                                self._state.discard(state)
(12)                        self._on_key_release(event)
(4)                 def _on_key_release(self, event):
(8)                     """Key release event handler."""
(4)                 def set_visible(self, visible):
(8)                     """Set the visibility of the selector artists."""
(8)                     self._visible = visible
(8)                     for artist in self.artists:
(12)                        artist.set_visible(visible)
(4)                 def get_visible(self):
(8)                     """Get the visibility of the selector artists."""
(8)                     return self._visible
(4)                 @property
(4)                 def visible(self):
(8)                     _api.warn_deprecated("3.8", alternative="get_visible")
(8)                     return self.get_visible()
(4)                 def clear(self):
(8)                     """Clear the selection and set the selector ready to make a new
one."""
(8)                     self._clear_without_update()
(8)                     self.update()
(4)                 def _clear_without_update(self):
(8)                     self._selection_completed = False
(8)                     self.set_visible(False)
(4)                 @property
(4)                 def artists(self):
(8)                     """Tuple of the artists of the selector."""
(8)                     handles_artists = getattr(self, '_handles_artists', ())
(8)                     return (self._selection_artist,) + handles_artists
(4)                 def set_props(self, **props):
(8)                     """
(8)                     Set the properties of the selector artist.
(8)                     See the *props* argument in the selector docstring to know which
properties are
(8)                     supported.
(8)                     """
(8)                     artist = self._selection_artist
(8)                     props = cbook.normalize_kwargs(props, artist)
(8)                     artist.set(**props)
(8)                     if self.useblit:
(12)                        self.update()
(4)                 def set_handle_props(self, **handle_props):
(8)                     """
(8)                     Set the properties of the handles selector artist. See the
(8)                     `handle_props` argument in the selector docstring to know which
(8)                     properties are supported.
(8)                     """
(8)                     if not hasattr(self, '_handles_artists'):
(12)                        raise NotImplementedError("This selector doesn't have handles.")
(8)                     artist = self._handles_artists[0]
(8)                     handle_props = cbook.normalize_kwargs(handle_props, artist)
```

```
(8)                 for handle in self._handles_artists:
(12)                    handle.set(**handle_props)
(8)                 if self.useblit:
(12)                    self.update()
(8)                 self._handle_props.update(handle_props)
(4)             def _validate_state(self, state):
(8)                 supported_state = [
(12)                    key for key, value in self._state_modifier_keys.items()
(12)                    if key != 'clear' and value != 'not-applicable'
(12)                    ]
(8)                 _api.check_in_list(supported_state, state=state)
(4)             def add_state(self, state):
(8)                 """
(8)                 Add a state to define the widget's behavior. See the
(8)                 `state_modifier_keys` parameters for details.
(8)                 Parameters
(8)                 ----------
(8)                 state : str
(12)                    Must be a supported state of the selector. See the
(12)                    `state_modifier_keys` parameters for details.
(8)                 Raises
(8)                 ------
(8)                 ValueError
(12)                    When the state is not supported by the selector.
(8)                 """
(8)                 self._validate_state(state)
(8)                 self._state.add(state)
(4)             def remove_state(self, state):
(8)                 """
(8)                 Remove a state to define the widget's behavior. See the
(8)                 `state_modifier_keys` parameters for details.
(8)                 Parameters
(8)                 ----------
(8)                 state : str
(12)                    Must be a supported state of the selector. See the
(12)                    `state_modifier_keys` parameters for details.
(8)                 Raises
(8)                 ------
(8)                 ValueError
(12)                    When the state is not supported by the selector.
(8)                 """
(8)                 self._validate_state(state)
(8)                 self._state.remove(state)
(0)         class SpanSelector(_SelectorWidget):
(4)             """
(4)             Visually select a min/max range on a single axis and call a function with
(4)             those values.
(4)             To guarantee that the selector remains responsive, keep a reference to it.
(4)             In order to turn off the SpanSelector, set ``span_selector.active`` to
(4)             False.  To turn it back on, set it to True.
(4)             Press and release events triggered at the same coordinates outside the
(4)             selection will clear the selector, except when
(4)             ``ignore_event_outside=True``.
(4)             Parameters
(4)             ----------
(4)             ax : `~matplotlib.axes.Axes`
(4)             onselect : callable with signature ``func(min: float, max: float)``
(8)                 A callback function that is called after a release event and the
(8)                 selection is created, changed or removed.
(4)             direction : {"horizontal", "vertical"}
(8)                 The direction along which to draw the span selector.
(4)             minspan : float, default: 0
(8)                 If selection is less than or equal to *minspan*, the selection is
(8)                 removed (when already existing) or cancelled.
(4)             useblit : bool, default: False
(8)                 If True, use the backend-dependent blitting features for faster
(8)                 canvas updates. See the tutorial :ref:`blitting` for details.
(4)             props : dict, default: {'facecolor': 'red', 'alpha': 0.5}
(8)                 Dictionary of `.Patch` properties.
```

```
(4)                onmove_callback : callable with signature ``func(min: float, max:
float)``, optional
(8)                    Called on mouse move while the span is being selected.
(4)                interactive : bool, default: False
(8)                    Whether to draw a set of handles that allow interaction with the
(8)                    widget after it is drawn.
(4)                button : `.MouseButton` or list of `.MouseButton`, default: all buttons
(8)                    The mouse buttons which activate the span selector.
(4)                handle_props : dict, default: None
(8)                    Properties of the handle lines at the edges of the span. Only used
(8)                    when *interactive* is True. See `.Line2D` for valid properties.
(4)                grab_range : float, default: 10
(8)                    Distance in pixels within which the interactive tool handles can be
activated.
(4)                state_modifier_keys : dict, optional
(8)                    Keyboard modifiers which affect the widget's behavior.  Values
(8)                    amend the defaults, which are:
(8)                    - "clear": Clear the current shape, default: "escape".
(4)                drag_from_anywhere : bool, default: False
(8)                    If `True`, the widget can be moved by clicking anywhere within its
bounds.
(4)                ignore_event_outside : bool, default: False
(8)                    If `True`, the event triggered outside the span selector will be
ignored.
(4)                snap_values : 1D array-like, optional
(8)                    Snap the selector edges to the given values.
(4)                Examples
(4)                --------
(4)                >>> import matplotlib.pyplot as plt
(4)                >>> import matplotlib.widgets as mwidgets
(4)                >>> fig, ax = plt.subplots()
(4)                >>> ax.plot([1, 2, 3], [10, 50, 100])
(4)                >>> def onselect(vmin, vmax):
(4)                ...     print(vmin, vmax)
(4)                >>> span = mwidgets.SpanSelector(ax, onselect, 'horizontal',
(4)                ...                               props=dict(facecolor='blue', alpha=0.5))
(4)                >>> fig.show()
(4)                See also: :doc:`/gallery/widgets/span_selector`
(4)                """
(4)                def __init__(self, ax, onselect, direction, *, minspan=0, useblit=False,
(17)                            props=None, onmove_callback=None, interactive=False,
(17)                            button=None, handle_props=None, grab_range=10,
(17)                            state_modifier_keys=None, drag_from_anywhere=False,
(17)                            ignore_event_outside=False, snap_values=None):
(8)                    if state_modifier_keys is None:
(12)                       state_modifier_keys = dict(clear='escape',
(39)                                                  square='not-applicable',
(39)                                                  center='not-applicable',
(39)                                                  rotate='not-applicable')
(8)                    super().__init__(ax, onselect, useblit=useblit, button=button,
(25)                                    state_modifier_keys=state_modifier_keys)
(8)                    if props is None:
(12)                       props = dict(facecolor='red', alpha=0.5)
(8)                    props['animated'] = self.useblit
(8)                    self.direction = direction
(8)                    self._extents_on_press = None
(8)                    self.snap_values = snap_values
(8)                    self.onmove_callback = onmove_callback
(8)                    self.minspan = minspan
(8)                    self.grab_range = grab_range
(8)                    self._interactive = interactive
(8)                    self._edge_handles = None
(8)                    self.drag_from_anywhere = drag_from_anywhere
(8)                    self.ignore_event_outside = ignore_event_outside
(8)                    self.new_axes(ax, _props=props, _init=True)
(8)                    self._handle_props = {
(12)                       'color': props.get('facecolor', 'r'),
(12)                       **cbook.normalize_kwargs(handle_props, Line2D)}
(8)                    if self._interactive:
```

```
2056: (12)                        self._edge_order = ['min', 'max']
2057: (12)                        self._setup_edge_handles(self._handle_props)
2058: (8)                    self._active_handle = None
2059: (4)                def new_axes(self, ax, *, _props=None, _init=False):
2060: (8)                    """Set SpanSelector to operate on a new Axes."""
2061: (8)                    reconnect = False
2062: (8)                    if _init or self.canvas is not ax.figure.canvas:
2063: (12)                        if self.canvas is not None:
2064: (16)                            self.disconnect_events()
2065: (12)                        reconnect = True
2066: (8)                    self.ax = ax
2067: (8)                    if reconnect:
2068: (12)                        self.connect_default_events()
2069: (8)                    self._selection_completed = False
2070: (8)                    if self.direction == 'horizontal':
2071: (12)                        trans = ax.get_xaxis_transform()
2072: (12)                        w, h = 0, 1
2073: (8)                    else:
2074: (12)                        trans = ax.get_yaxis_transform()
2075: (12)                        w, h = 1, 0
2076: (8)                    rect_artist = Rectangle((0, 0), w, h, transform=trans, visible=False)
2077: (8)                    if _props is not None:
2078: (12)                        rect_artist.update(_props)
2079: (8)                    elif self._selection_artist is not None:
2080: (12)                        rect_artist.update_from(self._selection_artist)
2081: (8)                    self.ax.add_patch(rect_artist)
2082: (8)                    self._selection_artist = rect_artist
2083: (4)                def _setup_edge_handles(self, props):
2084: (8)                    if self.direction == 'horizontal':
2085: (12)                        positions = self.ax.get_xbound()
2086: (8)                    else:
2087: (12)                        positions = self.ax.get_ybound()
2088: (8)                    self._edge_handles = ToolLineHandles(self.ax, positions,
2089: (45)                                                         direction=self.direction,
2090: (45)                                                         line_props=props,
2091: (45)                                                         useblit=self.useblit)
2092: (4)                @property
2093: (4)                def _handles_artists(self):
2094: (8)                    if self._edge_handles is not None:
2095: (12)                        return self._edge_handles.artists
2096: (8)                    else:
2097: (12)                        return ()
2098: (4)                def _set_cursor(self, enabled):
2099: (8)                    """Update the canvas cursor based on direction of the selector."""
2100: (8)                    if enabled:
2101: (12)                        cursor = (backend_tools.Cursors.RESIZE_HORIZONTAL
2102: (22)                                  if self.direction == 'horizontal' else
2103: (22)                                  backend_tools.Cursors.RESIZE_VERTICAL)
2104: (8)                    else:
2105: (12)                        cursor = backend_tools.Cursors.POINTER
2106: (8)                    self.ax.figure.canvas.set_cursor(cursor)
2107: (4)                def connect_default_events(self):
2108: (8)                    super().connect_default_events()
2109: (8)                    if getattr(self, '_interactive', False):
2110: (12)                        self.connect_event('motion_notify_event', self._hover)
2111: (4)                def _press(self, event):
2112: (8)                    """Button press event handler."""
2113: (8)                    self._set_cursor(True)
2114: (8)                    if self._interactive and self._selection_artist.get_visible():
2115: (12)                        self._set_active_handle(event)
2116: (8)                    else:
2117: (12)                        self._active_handle = None
2118: (8)                    if self._active_handle is None or not self._interactive:
2119: (12)                        self.update()
2120: (8)                    xdata, ydata = self._get_data_coords(event)
2121: (8)                    v = xdata if self.direction == 'horizontal' else ydata
2122: (8)                    if self._active_handle is None and not self.ignore_event_outside:
2123: (12)                        self._visible = False
2124: (12)                        self._set_extents((v, v))
```

```
(12)                        self._visible = True
(8)                     else:
(12)                        self.set_visible(True)
(8)                     return False
(4)                 @property
(4)                 def direction(self):
(8)                     """Direction of the span selector: 'vertical' or 'horizontal'."""
(8)                     return self._direction
(4)                 @direction.setter
(4)                 def direction(self, direction):
(8)                     """Set the direction of the span selector."""
(8)                     _api.check_in_list(['horizontal', 'vertical'], direction=direction)
(8)                     if hasattr(self, '_direction') and direction != self._direction:
(12)                        self._selection_artist.remove()
(12)                        if self._interactive:
(16)                            self._edge_handles.remove()
(12)                        self._direction = direction
(12)                        self.new_axes(self.ax)
(12)                        if self._interactive:
(16)                            self._setup_edge_handles(self._handle_props)
(8)                     else:
(12)                        self._direction = direction
(4)                 def _release(self, event):
(8)                     """Button release event handler."""
(8)                     self._set_cursor(False)
(8)                     if not self._interactive:
(12)                        self._selection_artist.set_visible(False)
(8)                     if (self._active_handle is None and self._selection_completed and
(16)                            self.ignore_event_outside):
(12)                        return
(8)                     vmin, vmax = self.extents
(8)                     span = vmax - vmin
(8)                     if span <= self.minspan:
(12)                        self.set_visible(False)
(12)                        if self._selection_completed:
(16)                            self.onselect(vmin, vmax)
(12)                        self._selection_completed = False
(8)                     else:
(12)                        self.onselect(vmin, vmax)
(12)                        self._selection_completed = True
(8)                     self.update()
(8)                     self._active_handle = None
(8)                     return False
(4)                 def _hover(self, event):
(8)                     """Update the canvas cursor if it's over a handle."""
(8)                     if self.ignore(event):
(12)                        return
(8)                     if self._active_handle is not None or not self._selection_completed:
(12)                        return
(8)                     _, e_dist = self._edge_handles.closest(event.x, event.y)
(8)                     self._set_cursor(e_dist <= self.grab_range)
(4)                 def _onmove(self, event):
(8)                     """Motion notify event handler."""
(8)                     xdata, ydata = self._get_data_coords(event)
(8)                     if self.direction == 'horizontal':
(12)                        v = xdata
(12)                        vpress = self._eventpress.xdata
(8)                     else:
(12)                        v = ydata
(12)                        vpress = self._eventpress.ydata
(8)                     if self._active_handle == 'C' and self._extents_on_press is not None:
(12)                        vmin, vmax = self._extents_on_press
(12)                        dv = v - vpress
(12)                        vmin += dv
(12)                        vmax += dv
(8)                     elif self._active_handle and self._active_handle != 'C':
(12)                        vmin, vmax = self._extents_on_press
(12)                        if self._active_handle == 'min':
(16)                            vmin = v
```

```
(12)                        else:
(16)                            vmax = v
(8)                     else:
(12)                        if self.ignore_event_outside and self._selection_completed:
(16)                            return
(12)                        vmin, vmax = vpress, v
(12)                        if vmin > vmax:
(16)                            vmin, vmax = vmax, vmin
(8)                     self._set_extents((vmin, vmax))
(8)                     if self.onmove_callback is not None:
(12)                        self.onmove_callback(vmin, vmax)
(8)                     return False
(4)                 def _draw_shape(self, vmin, vmax):
(8)                     if vmin > vmax:
(12)                        vmin, vmax = vmax, vmin
(8)                     if self.direction == 'horizontal':
(12)                        self._selection_artist.set_x(vmin)
(12)                        self._selection_artist.set_width(vmax - vmin)
(8)                     else:
(12)                        self._selection_artist.set_y(vmin)
(12)                        self._selection_artist.set_height(vmax - vmin)
(4)                 def _set_active_handle(self, event):
(8)                     """Set active handle based on the location of the mouse event."""
(8)                     e_idx, e_dist = self._edge_handles.closest(event.x, event.y)
(8)                     if 'move' in self._state:
(12)                        self._active_handle = 'C'
(8)                     elif e_dist > self.grab_range:
(12)                        self._active_handle = None
(12)                        if self.drag_from_anywhere and self._contains(event):
(16)                            self._active_handle = 'C'
(16)                            self._extents_on_press = self.extents
(12)                        else:
(16)                            self._active_handle = None
(16)                            return
(8)                     else:
(12)                        self._active_handle = self._edge_order[e_idx]
(8)                     self._extents_on_press = self.extents
(4)                 def _contains(self, event):
(8)                     """Return True if event is within the patch."""
(8)                     return self._selection_artist.contains(event, radius=0)[0]
(4)                 @staticmethod
(4)                 def _snap(values, snap_values):
(8)                     """Snap values to a given array values (snap_values)."""
(8)                     eps = np.min(np.abs(np.diff(snap_values))) * 1e-12
(8)                     return tuple(
(12)                        snap_values[np.abs(snap_values - v + np.sign(v) * eps).argmin()]
(12)                        for v in values)
(4)                 @property
(4)                 def extents(self):
(8)                     """
(8)                     (float, float)
(12)                        The values, in data coordinates, for the start and end points of
the current
(12)                        selection. If there is no selection then the start and end values
will be
(12)                        the same.
(8)                     """
(8)                     if self.direction == 'horizontal':
(12)                        vmin = self._selection_artist.get_x()
(12)                        vmax = vmin + self._selection_artist.get_width()
(8)                     else:
(12)                        vmin = self._selection_artist.get_y()
(12)                        vmax = vmin + self._selection_artist.get_height()
(8)                     return vmin, vmax
(4)                 @extents.setter
(4)                 def extents(self, extents):
(8)                     self._set_extents(extents)
(8)                     self._selection_completed = True
(4)                 def _set_extents(self, extents):
```

```
2261: (8)                   if self.snap_values is not None:
2262: (12)                      extents = tuple(self._snap(extents, self.snap_values))
2263: (8)                   self._draw_shape(*extents)
2264: (8)                   if self._interactive:
2265: (12)                      self._edge_handles.set_data(self.extents)
2266: (8)                   self.set_visible(self._visible)
2267: (8)                   self.update()
2268: (0)           class ToolLineHandles:
2269: (4)               """
2270: (4)               Control handles for canvas tools.
2271: (4)               Parameters
2272: (4)               ----------
2273: (4)               ax : `~matplotlib.axes.Axes`
2274: (8)                   Matplotlib Axes where tool handles are displayed.
2275: (4)               positions : 1D array
2276: (8)                   Positions of handles in data coordinates.
2277: (4)               direction : {"horizontal", "vertical"}
2278: (8)                   Direction of handles, either 'vertical' or 'horizontal'
2279: (4)               line_props : dict, optional
2280: (8)                   Additional line properties. See `.Line2D`.
2281: (4)               useblit : bool, default: True
2282: (8)                   Whether to use blitting for faster drawing (if supported by the
2283: (8)                   backend). See the tutorial :ref:`blitting`
2284: (8)                   for details.
2285: (4)               """
2286: (4)               def __init__(self, ax, positions, direction, *, line_props=None,
2287: (17)                           useblit=True):
2288: (8)                   self.ax = ax
2289: (8)                   _api.check_in_list(['horizontal', 'vertical'], direction=direction)
2290: (8)                   self._direction = direction
2291: (8)                   line_props = {
2292: (12)                      **(line_props if line_props is not None else {}),
2293: (12)                      'visible': False,
2294: (12)                      'animated': useblit,
2295: (8)                   }
2296: (8)                   line_fun = ax.axvline if self.direction == 'horizontal' else
ax.axhline
2297: (8)                   self._artists = [line_fun(p, **line_props) for p in positions]
2298: (4)               @property
2299: (4)               def artists(self):
2300: (8)                   return tuple(self._artists)
2301: (4)               @property
2302: (4)               def positions(self):
2303: (8)                   """Positions of the handle in data coordinates."""
2304: (8)                   method = 'get_xdata' if self.direction == 'horizontal' else
'get_ydata'
2305: (8)                   return [getattr(line, method)()[0] for line in self.artists]
2306: (4)               @property
2307: (4)               def direction(self):
2308: (8)                   """Direction of the handle: 'vertical' or 'horizontal'."""
2309: (8)                   return self._direction
2310: (4)               def set_data(self, positions):
2311: (8)                   """
2312: (8)                   Set x- or y-positions of handles, depending on if the lines are
2313: (8)                   vertical or horizontal.
2314: (8)                   Parameters
2315: (8)                   ----------
2316: (8)                   positions : tuple of length 2
2317: (12)                      Set the positions of the handle in data coordinates
2318: (8)                   """
2319: (8)                   method = 'set_xdata' if self.direction == 'horizontal' else
'set_ydata'
2320: (8)                   for line, p in zip(self.artists, positions):
2321: (12)                      getattr(line, method)([p, p])
2322: (4)               def set_visible(self, value):
2323: (8)                   """Set the visibility state of the handles artist."""
2324: (8)                   for artist in self.artists:
2325: (12)                      artist.set_visible(value)
2326: (4)               def set_animated(self, value):
```

```
(8)                 """Set the animated state of the handles artist."""
(8)                 for artist in self.artists:
(12)                    artist.set_animated(value)
(4)             def remove(self):
(8)                 """Remove the handles artist from the figure."""
(8)                 for artist in self._artists:
(12)                    artist.remove()
(4)             def closest(self, x, y):
(8)                 """
(8)                 Return index and pixel distance to closest handle.
(8)                 Parameters
(8)                 ----------
(8)                 x, y : float
(12)                    x, y position from which the distance will be calculated to
(12)                    determinate the closest handle
(8)                 Returns
(8)                 -------
(8)                 index, distance : index of the handle and its distance from
(12)                    position x, y
(8)                 """
(8)                 if self.direction == 'horizontal':
(12)                    p_pts = np.array([
(16)                        self.ax.transData.transform((p, 0))[0] for p in self.positions
(16)                        ])
(12)                    dist = abs(p_pts - x)
(8)                 else:
(12)                    p_pts = np.array([
(16)                        self.ax.transData.transform((0, p))[1] for p in self.positions
(16)                        ])
(12)                    dist = abs(p_pts - y)
(8)                 index = np.argmin(dist)
(8)                 return index, dist[index]
(0)         class ToolHandles:
(4)             """
(4)             Control handles for canvas tools.
(4)             Parameters
(4)             ----------
(4)             ax : `~matplotlib.axes.Axes`
(8)                 Matplotlib Axes where tool handles are displayed.
(4)             x, y : 1D arrays
(8)                 Coordinates of control handles.
(4)             marker : str, default: 'o'
(8)                 Shape of marker used to display handle. See `~.pyplot.plot`.
(4)             marker_props : dict, optional
(8)                 Additional marker properties. See `.Line2D`.
(4)             useblit : bool, default: True
(8)                 Whether to use blitting for faster drawing (if supported by the
(8)                 backend). See the tutorial :ref:`blitting`
(8)                 for details.
(4)             """
(4)             def __init__(self, ax, x, y, *, marker='o', marker_props=None,
useblit=True):
(8)                 self.ax = ax
(8)                 props = {'marker': marker, 'markersize': 7, 'markerfacecolor': 'w',
(17)                         'linestyle': 'none', 'alpha': 0.5, 'visible': False,
(17)                         'label': '_nolegend_',
(17)                         **cbook.normalize_kwargs(marker_props, Line2D._alias_map)}
(8)                 self._markers = Line2D(x, y, animated=useblit, **props)
(8)                 self.ax.add_line(self._markers)
(4)             @property
(4)             def x(self):
(8)                 return self._markers.get_xdata()
(4)             @property
(4)             def y(self):
(8)                 return self._markers.get_ydata()
(4)             @property
(4)             def artists(self):
(8)                 return (self._markers, )
(4)             def set_data(self, pts, y=None):
```

```
(8)                 """Set x and y positions of handles."""
(8)                 if y is not None:
(12)                    x = pts
(12)                    pts = np.array([x, y])
(8)                 self._markers.set_data(pts)
(4)             def set_visible(self, val):
(8)                 self._markers.set_visible(val)
(4)             def set_animated(self, val):
(8)                 self._markers.set_animated(val)
(4)             def closest(self, x, y):
(8)                 """Return index and pixel distance to closest index."""
(8)                 pts = np.column_stack([self.x, self.y])
(8)                 pts = self.ax.transData.transform(pts)
(8)                 diff = pts - [x, y]
(8)                 dist = np.hypot(*diff.T)
(8)                 min_index = np.argmin(dist)
(8)                 return min_index, dist[min_index]
(0)         _RECTANGLESELECTOR_PARAMETERS_DOCSTRING = \
(4)             r"""
(4)             Parameters
(4)             ----------
(4)             ax : `~matplotlib.axes.Axes`
(8)                 The parent Axes for the widget.
(4)             onselect : function
(8)                 A callback function that is called after a release event and the
(8)                 selection is created, changed or removed.
(8)                 It must have the signature::
(12)                    def onselect(eclick: MouseEvent, erelease: MouseEvent)
(8)                 where *eclick* and *erelease* are the mouse click and release
(8)                 `.MouseEvent`\s that start and complete the selection.
(4)             minspanx : float, default: 0
(8)                 Selections with an x-span less than or equal to *minspanx* are removed
(8)                 (when already existing) or cancelled.
(4)             minspany : float, default: 0
(8)                 Selections with an y-span less than or equal to *minspanx* are removed
(8)                 (when already existing) or cancelled.
(4)             useblit : bool, default: False
(8)                 Whether to use blitting for faster drawing (if supported by the
(8)                 backend). See the tutorial :ref:`blitting`
(8)                 for details.
(4)             props : dict, optional
(8)                 Properties with which the __ARTIST_NAME__ is drawn. See
(8)                 `.Patch` for valid properties.
(8)                 Default:
(8)                 ``dict(facecolor='red', edgecolor='black', alpha=0.2, fill=True)``
(4)             spancoords : {"data", "pixels"}, default: "data"
(8)                 Whether to interpret *minspanx* and *minspany* in data or in pixel
(8)                 coordinates.
(4)             button : `.MouseButton`, list of `.MouseButton`, default: all buttons
(8)                 Button(s) that trigger rectangle selection.
(4)             grab_range : float, default: 10
(8)                 Distance in pixels within which the interactive tool handles can be
(8)                 activated.
(4)             handle_props : dict, optional
(8)                 Properties with which the interactive handles (marker artists) are
(8)                 drawn. See the marker arguments in `.Line2D` for valid
(8)                 properties.  Default values are defined in ``mpl.rcParams`` except for
(8)                 the default value of ``markeredgecolor`` which will be the same as the
(8)                 ``edgecolor`` property in *props*.
(4)             interactive : bool, default: False
(8)                 Whether to draw a set of handles that allow interaction with the
(8)                 widget after it is drawn.
(4)             state_modifier_keys : dict, optional
(8)                 Keyboard modifiers which affect the widget's behavior.  Values
(8)                 amend the defaults, which are:
(8)                 - "move": Move the existing shape, default: no modifier.
(8)                 - "clear": Clear the current shape, default: "escape".
(8)                 - "square": Make the shape square, default: "shift".
(8)                 - "center": change the shape around its center, default: "ctrl".
```

```
(8)                 - "rotate": Rotate the shape around its center between -45° and 45°,
(10)                  default: "r".
(8)                 "square" and "center" can be combined. The square shape can be defined
(8)                 in data or display coordinates as determined by the
(8)                 ``use_data_coordinates`` argument specified when creating the
selector.
(4)             drag_from_anywhere : bool, default: False
(8)                 If `True`, the widget can be moved by clicking anywhere within
(8)                 its bounds.
(4)             ignore_event_outside : bool, default: False
(8)                 If `True`, the event triggered outside the span selector will be
(8)                 ignored.
(4)             use_data_coordinates : bool, default: False
(8)                 If `True`, the "square" shape of the selector is defined in
(8)                 data coordinates instead of display coordinates.
(4)             """
(0)         @_docstring.Substitution(_RECTANGLESELECTOR_PARAMETERS_DOCSTRING.replace(
(4)             '__ARTIST_NAME__', 'rectangle'))
(0)         class RectangleSelector(_SelectorWidget):
(4)             """
(4)             Select a rectangular region of an Axes.
(4)             For the cursor to remain responsive you must keep a reference to it.
(4)             Press and release events triggered at the same coordinates outside the
(4)             selection will clear the selector, except when
(4)             ``ignore_event_outside=True``.
(4)             %s
(4)             Examples
(4)             --------
(4)             >>> import matplotlib.pyplot as plt
(4)             >>> import matplotlib.widgets as mwidgets
(4)             >>> fig, ax = plt.subplots()
(4)             >>> ax.plot([1, 2, 3], [10, 50, 100])
(4)             >>> def onselect(eclick, erelease):
(4)             ...     print(eclick.xdata, eclick.ydata)
(4)             ...     print(erelease.xdata, erelease.ydata)
(4)             >>> props = dict(facecolor='blue', alpha=0.5)
(4)             >>> rect = mwidgets.RectangleSelector(ax, onselect, interactive=True,
(4)             ...                                   props=props)
(4)             >>> fig.show()
(4)             >>> rect.add_state('square')
(4)             See also: :doc:`/gallery/widgets/rectangle_selector`
(4)             """
(4)             def __init__(self, ax, onselect, *, minspanx=0, minspany=0, useblit=False,
(17)                         props=None, spancoords='data', button=None, grab_range=10,
(17)                         handle_props=None, interactive=False,
(17)                         state_modifier_keys=None, drag_from_anywhere=False,
(17)                         ignore_event_outside=False, use_data_coordinates=False):
(8)                 super().__init__(ax, onselect, useblit=useblit, button=button,
(25)                                 state_modifier_keys=state_modifier_keys,
(25)                                 use_data_coordinates=use_data_coordinates)
(8)                 self._interactive = interactive
(8)                 self.drag_from_anywhere = drag_from_anywhere
(8)                 self.ignore_event_outside = ignore_event_outside
(8)                 self._rotation = 0.0
(8)                 self._aspect_ratio_correction = 1.0
(8)                 self._allow_creation = True
(8)                 if props is None:
(12)                    props = dict(facecolor='red', edgecolor='black',
(25)                                 alpha=0.2, fill=True)
(8)                 props = {**props, 'animated': self.useblit}
(8)                 self._visible = props.pop('visible', self._visible)
(8)                 to_draw = self._init_shape(**props)
(8)                 self.ax.add_patch(to_draw)
(8)                 self._selection_artist = to_draw
(8)                 self._set_aspect_ratio_correction()
(8)                 self.minspanx = minspanx
(8)                 self.minspany = minspany
(8)                 _api.check_in_list(['data', 'pixels'], spancoords=spancoords)
(8)                 self.spancoords = spancoords
```

```
2532: (8)                    self.grab_range = grab_range
2533: (8)                    if self._interactive:
2534: (12)                       self._handle_props = {
2535: (16)                           'markeredgecolor': (props or {}).get('edgecolor', 'black'),
2536: (16)                           **cbook.normalize_kwargs(handle_props, Line2D)}
2537: (12)                       self._corner_order = ['SW', 'SE', 'NE', 'NW']
2538: (12)                       xc, yc = self.corners
2539: (12)                       self._corner_handles = ToolHandles(self.ax, xc, yc,
2540: (47)
marker_props=self._handle_props,
2541: (47)                                                          useblit=self.useblit)
2542: (12)                       self._edge_order = ['W', 'S', 'E', 'N']
2543: (12)                       xe, ye = self.edge_centers
2544: (12)                       self._edge_handles = ToolHandles(self.ax, xe, ye, marker='s',
2545: (45)                                                        marker_props=self._handle_props,
2546: (45)                                                        useblit=self.useblit)
2547: (12)                       xc, yc = self.center
2548: (12)                       self._center_handle = ToolHandles(self.ax, [xc], [yc], marker='s',
2549: (46)                                                         marker_props=self._handle_props,
2550: (46)                                                         useblit=self.useblit)
2551: (12)                       self._active_handle = None
2552: (8)                    self._extents_on_press = None
2553: (4)                @property
2554: (4)                def _handles_artists(self):
2555: (8)                    return (*self._center_handle.artists, *self._corner_handles.artists,
2556: (16)                           *self._edge_handles.artists)
2557: (4)                def _init_shape(self, **props):
2558: (8)                    return Rectangle((0, 0), 0, 1, visible=False,
2559: (25)                                     rotation_point='center', **props)
2560: (4)                def _press(self, event):
2561: (8)                    """Button press event handler."""
2562: (8)                    if self._interactive and self._selection_artist.get_visible():
2563: (12)                       self._set_active_handle(event)
2564: (8)                    else:
2565: (12)                       self._active_handle = None
2566: (8)                    if ((self._active_handle is None or not self._interactive) and
2567: (16)                           self._allow_creation):
2568: (12)                       self.update()
2569: (8)                    if (self._active_handle is None and not self.ignore_event_outside and
2570: (16)                           self._allow_creation):
2571: (12)                       x, y = self._get_data_coords(event)
2572: (12)                       self._visible = False
2573: (12)                       self.extents = x, x, y, y
2574: (12)                       self._visible = True
2575: (8)                    else:
2576: (12)                       self.set_visible(True)
2577: (8)                    self._extents_on_press = self.extents
2578: (8)                    self._rotation_on_press = self._rotation
2579: (8)                    self._set_aspect_ratio_correction()
2580: (8)                    return False
2581: (4)                def _release(self, event):
2582: (8)                    """Button release event handler."""
2583: (8)                    if not self._interactive:
2584: (12)                       self._selection_artist.set_visible(False)
2585: (8)                    if (self._active_handle is None and self._selection_completed and
2586: (16)                           self.ignore_event_outside):
2587: (12)                       return
2588: (8)                    x0, x1, y0, y1 = self.extents
2589: (8)                    self._eventpress.xdata = x0
2590: (8)                    self._eventpress.ydata = y0
2591: (8)                    xy0 = self.ax.transData.transform([x0, y0])
2592: (8)                    self._eventpress.x, self._eventpress.y = xy0
2593: (8)                    self._eventrelease.xdata = x1
2594: (8)                    self._eventrelease.ydata = y1
2595: (8)                    xy1 = self.ax.transData.transform([x1, y1])
2596: (8)                    self._eventrelease.x, self._eventrelease.y = xy1
2597: (8)                    if self.spancoords == 'data':
2598: (12)                       spanx = abs(self._eventpress.xdata - self._eventrelease.xdata)
2599: (12)                       spany = abs(self._eventpress.ydata - self._eventrelease.ydata)
```

```
2600: (8)                   elif self.spancoords == 'pixels':
2601: (12)                      spanx = abs(self._eventpress.x - self._eventrelease.x)
2602: (12)                      spany = abs(self._eventpress.y - self._eventrelease.y)
2603: (8)                   else:
2604: (12)                      _api.check_in_list(['data', 'pixels'],
2605: (31)                                         spancoords=self.spancoords)
2606: (8)                   if spanx <= self.minspanx or spany <= self.minspany:
2607: (12)                      if self._selection_completed:
2608: (16)                          self.onselect(self._eventpress, self._eventrelease)
2609: (12)                      self._clear_without_update()
2610: (8)                   else:
2611: (12)                      self.onselect(self._eventpress, self._eventrelease)
2612: (12)                      self._selection_completed = True
2613: (8)                   self.update()
2614: (8)                   self._active_handle = None
2615: (8)                   self._extents_on_press = None
2616: (8)                   return False
2617: (4)               def _onmove(self, event):
2618: (8)                   """
2619: (8)                   Motion notify event handler.
2620: (8)                   This can do one of four things:
2621: (8)                   - Translate
2622: (8)                   - Rotate
2623: (8)                   - Re-size
2624: (8)                   - Continue the creation of a new shape
2625: (8)                   """
2626: (8)                   eventpress = self._eventpress
2627: (8)                   state = self._state
2628: (8)                   rotate = 'rotate' in state and self._active_handle in
self._corner_order
2629: (8)                   move = self._active_handle == 'C'
2630: (8)                   resize = self._active_handle and not move
2631: (8)                   xdata, ydata = self._get_data_coords(event)
2632: (8)                   if resize:
2633: (12)                      inv_tr = self._get_rotation_transform().inverted()
2634: (12)                      xdata, ydata = inv_tr.transform([xdata, ydata])
2635: (12)                      eventpress.xdata, eventpress.ydata = inv_tr.transform(
2636: (16)                          (eventpress.xdata, eventpress.ydata))
2637: (8)                   dx = xdata - eventpress.xdata
2638: (8)                   dy = ydata - eventpress.ydata
2639: (8)                   refmax = None
2640: (8)                   if self._use_data_coordinates:
2641: (12)                      refx, refy = dx, dy
2642: (8)                   else:
2643: (12)                      refx = event.x - eventpress.x
2644: (12)                      refy = event.y - eventpress.y
2645: (8)                   x0, x1, y0, y1 = self._extents_on_press
2646: (8)                   if rotate:
2647: (12)                      a = (eventpress.xdata, eventpress.ydata)
2648: (12)                      b = self.center
2649: (12)                      c = (xdata, ydata)
2650: (12)                      angle = (np.arctan2(c[1]-b[1], c[0]-b[0]) -
2651: (21)                               np.arctan2(a[1]-b[1], a[0]-b[0]))
2652: (12)                      self.rotation = np.rad2deg(self._rotation_on_press + angle)
2653: (8)                   elif resize:
2654: (12)                      size_on_press = [x1 - x0, y1 - y0]
2655: (12)                      center = (x0 + size_on_press[0] / 2, y0 + size_on_press[1] / 2)
2656: (12)                      if 'center' in state:
2657: (16)                          if 'square' in state:
2658: (20)                              if self._active_handle in self._corner_order:
2659: (24)                                  refmax = max(refx, refy, key=abs)
2660: (20)                              if self._active_handle in ['E', 'W'] or refmax == refx:
2661: (24)                                  hw = xdata - center[0]
2662: (24)                                  hh = hw / self._aspect_ratio_correction
2663: (20)                              else:
2664: (24)                                  hh = ydata - center[1]
2665: (24)                                  hw = hh * self._aspect_ratio_correction
2666: (16)                          else:
2667: (20)                              hw = size_on_press[0] / 2
```

```
2668: (20)                                hh = size_on_press[1] / 2
2669: (20)                                if self._active_handle in ['E', 'W'] + self._corner_order:
2670: (24)                                    hw = abs(xdata - center[0])
2671: (20)                                if self._active_handle in ['N', 'S'] + self._corner_order:
2672: (24)                                    hh = abs(ydata - center[1])
2673: (16)                            x0, x1, y0, y1 = (center[0] - hw, center[0] + hw,
2674: (34)                                              center[1] - hh, center[1] + hh)
2675: (12)                        else:
2676: (16)                            if 'W' in self._active_handle:
2677: (20)                                x0 = x1
2678: (16)                            if 'S' in self._active_handle:
2679: (20)                                y0 = y1
2680: (16)                            if self._active_handle in ['E', 'W'] + self._corner_order:
2681: (20)                                x1 = xdata
2682: (16)                            if self._active_handle in ['N', 'S'] + self._corner_order:
2683: (20)                                y1 = ydata
2684: (16)                            if 'square' in state:
2685: (20)                                if self._active_handle in self._corner_order:
2686: (24)                                    refmax = max(refx, refy, key=abs)
2687: (20)                                if self._active_handle in ['E', 'W'] or refmax == refx:
2688: (24)                                    sign = np.sign(ydata - y0)
2689: (24)                                    y1 = y0 + sign * abs(x1 - x0) /
self._aspect_ratio_correction
2690: (20)                                else:
2691: (24)                                    sign = np.sign(xdata - x0)
2692: (24)                                    x1 = x0 + sign * abs(y1 - y0) *
self._aspect_ratio_correction
2693: (8)                    elif move:
2694: (12)                        x0, x1, y0, y1 = self._extents_on_press
2695: (12)                        dx = xdata - eventpress.xdata
2696: (12)                        dy = ydata - eventpress.ydata
2697: (12)                        x0 += dx
2698: (12)                        x1 += dx
2699: (12)                        y0 += dy
2700: (12)                        y1 += dy
2701: (8)                    else:
2702: (12)                        self._rotation = 0
2703: (12)                        if ((self.ignore_event_outside and self._selection_completed) or
2704: (20)                                not self._allow_creation):
2705: (16)                            return
2706: (12)                        center = [eventpress.xdata, eventpress.ydata]
2707: (12)                        dx = (xdata - center[0]) / 2
2708: (12)                        dy = (ydata - center[1]) / 2
2709: (12)                        if 'square' in state:
2710: (16)                            refmax = max(refx, refy, key=abs)
2711: (16)                            if refmax == refx:
2712: (20)                                dy = np.sign(dy) * abs(dx) / self._aspect_ratio_correction
2713: (16)                            else:
2714: (20)                                dx = np.sign(dx) * abs(dy) * self._aspect_ratio_correction
2715: (12)                        if 'center' in state:
2716: (16)                            dx *= 2
2717: (16)                            dy *= 2
2718: (12)                        else:
2719: (16)                            center[0] += dx
2720: (16)                            center[1] += dy
2721: (12)                        x0, x1, y0, y1 = (center[0] - dx, center[0] + dx,
2722: (30)                                          center[1] - dy, center[1] + dy)
2723: (8)                    self.extents = x0, x1, y0, y1
2724: (4)                @property
2725: (4)                def _rect_bbox(self):
2726: (8)                    return self._selection_artist.get_bbox().bounds
2727: (4)                def _set_aspect_ratio_correction(self):
2728: (8)                    aspect_ratio = self.ax._get_aspect_ratio()
2729: (8)                    self._selection_artist._aspect_ratio_correction = aspect_ratio
2730: (8)                    if self._use_data_coordinates:
2731: (12)                       self._aspect_ratio_correction = 1
2732: (8)                    else:
2733: (12)                       self._aspect_ratio_correction = aspect_ratio
2734: (4)                def _get_rotation_transform(self):
```

```
(8)                 aspect_ratio = self.ax._get_aspect_ratio()
(8)                 return Affine2D().translate(-self.center[0], -self.center[1]) \
(16)                        .scale(1, aspect_ratio) \
(16)                        .rotate(self._rotation) \
(16)                        .scale(1, 1 / aspect_ratio) \
(16)                        .translate(*self.center)
(4)             @property
(4)             def corners(self):
(8)                 """
(8)                 Corners of rectangle in data coordinates from lower left,
(8)                 moving clockwise.
(8)                 """
(8)                 x0, y0, width, height = self._rect_bbox
(8)                 xc = x0, x0 + width, x0 + width, x0
(8)                 yc = y0, y0, y0 + height, y0 + height
(8)                 transform = self._get_rotation_transform()
(8)                 coords = transform.transform(np.array([xc, yc]).T).T
(8)                 return coords[0], coords[1]
(4)             @property
(4)             def edge_centers(self):
(8)                 """
(8)                 Midpoint of rectangle edges in data coordinates from left,
(8)                 moving anti-clockwise.
(8)                 """
(8)                 x0, y0, width, height = self._rect_bbox
(8)                 w = width / 2.
(8)                 h = height / 2.
(8)                 xe = x0, x0 + w, x0 + width, x0 + w
(8)                 ye = y0 + h, y0, y0 + h, y0 + height
(8)                 transform = self._get_rotation_transform()
(8)                 coords = transform.transform(np.array([xe, ye]).T).T
(8)                 return coords[0], coords[1]
(4)             @property
(4)             def center(self):
(8)                 """Center of rectangle in data coordinates."""
(8)                 x0, y0, width, height = self._rect_bbox
(8)                 return x0 + width / 2., y0 + height / 2.
(4)             @property
(4)             def extents(self):
(8)                 """
(8)                 Return (xmin, xmax, ymin, ymax) in data coordinates as defined by the
(8)                 bounding box before rotation.
(8)                 """
(8)                 x0, y0, width, height = self._rect_bbox
(8)                 xmin, xmax = sorted([x0, x0 + width])
(8)                 ymin, ymax = sorted([y0, y0 + height])
(8)                 return xmin, xmax, ymin, ymax
(4)             @extents.setter
(4)             def extents(self, extents):
(8)                 self._draw_shape(extents)
(8)                 if self._interactive:
(12)                    self._corner_handles.set_data(*self.corners)
(12)                    self._edge_handles.set_data(*self.edge_centers)
(12)                    x, y = self.center
(12)                    self._center_handle.set_data([x], [y])
(8)                 self.set_visible(self._visible)
(8)                 self.update()
(4)             @property
(4)             def rotation(self):
(8)                 """
(8)                 Rotation in degree in interval [-45°, 45°]. The rotation is limited in
(8)                 range to keep the implementation simple.
(8)                 """
(8)                 return np.rad2deg(self._rotation)
(4)             @rotation.setter
(4)             def rotation(self, value):
(8)                 if -45 <= value and value <= 45:
(12)                    self._rotation = np.deg2rad(value)
(12)                    self.extents = self.extents
```

```
(4)     def _draw_shape(self, extents):
(8)         x0, x1, y0, y1 = extents
(8)         xmin, xmax = sorted([x0, x1])
(8)         ymin, ymax = sorted([y0, y1])
(8)         xlim = sorted(self.ax.get_xlim())
(8)         ylim = sorted(self.ax.get_ylim())
(8)         xmin = max(xlim[0], xmin)
(8)         ymin = max(ylim[0], ymin)
(8)         xmax = min(xmax, xlim[1])
(8)         ymax = min(ymax, ylim[1])
(8)         self._selection_artist.set_x(xmin)
(8)         self._selection_artist.set_y(ymin)
(8)         self._selection_artist.set_width(xmax - xmin)
(8)         self._selection_artist.set_height(ymax - ymin)
(8)         self._selection_artist.set_angle(self.rotation)
(4)     def _set_active_handle(self, event):
(8)         """Set active handle based on the location of the mouse event."""
(8)         c_idx, c_dist = self._corner_handles.closest(event.x, event.y)
(8)         e_idx, e_dist = self._edge_handles.closest(event.x, event.y)
(8)         m_idx, m_dist = self._center_handle.closest(event.x, event.y)
(8)         if 'move' in self._state:
(12)            self._active_handle = 'C'
(8)         elif m_dist < self.grab_range * 2:
(12)            self._active_handle = 'C'
(8)         elif c_dist > self.grab_range and e_dist > self.grab_range:
(12)            if self.drag_from_anywhere and self._contains(event):
(16)                self._active_handle = 'C'
(12)            else:
(16)                self._active_handle = None
(16)                return
(8)         elif c_dist < e_dist:
(12)            self._active_handle = self._corner_order[c_idx]
(8)         else:
(12)            self._active_handle = self._edge_order[e_idx]
(4)     def _contains(self, event):
(8)         """Return True if event is within the patch."""
(8)         return self._selection_artist.contains(event, radius=0)[0]
(4)     @property
(4)     def geometry(self):
(8)         """
(8)         Return an array of shape (2, 5) containing the
(8)         x (``RectangleSelector.geometry[1, :]``) and
(8)         y (``RectangleSelector.geometry[0, :]``) data coordinates of the four
(8)         corners of the rectangle starting and ending in the top left corner.
(8)         """
(8)         if hasattr(self._selection_artist, 'get_verts'):
(12)            xfm = self.ax.transData.inverted()
(12)            y, x = xfm.transform(self._selection_artist.get_verts()).T
(12)            return np.array([x, y])
(8)         else:
(12)            return np.array(self._selection_artist.get_data())
(0) @_docstring.Substitution(_RECTANGLESELECTOR_PARAMETERS_DOCSTRING.replace(
(4)     '__ARTIST_NAME__', 'ellipse'))
(0) class EllipseSelector(RectangleSelector):
(4)     """
(4)     Select an elliptical region of an Axes.
(4)     For the cursor to remain responsive you must keep a reference to it.
(4)     Press and release events triggered at the same coordinates outside the
(4)     selection will clear the selector, except when
(4)     ``ignore_event_outside=True``.
(4)     %s
(4)     Examples
(4)     --------
(4)     :doc:`/gallery/widgets/rectangle_selector`
(4)     """
(4)     def _init_shape(self, **props):
(8)         return Ellipse((0, 0), 0, 1, visible=False, **props)
(4)     def _draw_shape(self, extents):
(8)         x0, x1, y0, y1 = extents
```

```
2873: (8)                   xmin, xmax = sorted([x0, x1])
2874: (8)                   ymin, ymax = sorted([y0, y1])
2875: (8)                   center = [x0 + (x1 - x0) / 2., y0 + (y1 - y0) / 2.]
2876: (8)                   a = (xmax - xmin) / 2.
2877: (8)                   b = (ymax - ymin) / 2.
2878: (8)                   self._selection_artist.center = center
2879: (8)                   self._selection_artist.width = 2 * a
2880: (8)                   self._selection_artist.height = 2 * b
2881: (8)                   self._selection_artist.angle = self.rotation
2882: (4)               @property
2883: (4)               def _rect_bbox(self):
2884: (8)                   x, y = self._selection_artist.center
2885: (8)                   width = self._selection_artist.width
2886: (8)                   height = self._selection_artist.height
2887: (8)                   return x - width / 2., y - height / 2., width, height
2888: (0)           class LassoSelector(_SelectorWidget):
2889: (4)               """
2890: (4)               Selection curve of an arbitrary shape.
2891: (4)               For the selector to remain responsive you must keep a reference to it.
2892: (4)               The selected path can be used in conjunction with `~.Path.contains_point`
2893: (4)               to select data points from an image.
2894: (4)               In contrast to `Lasso`, `LassoSelector` is written with an interface
2895: (4)               similar to `RectangleSelector` and `SpanSelector`, and will continue to
2896: (4)               interact with the Axes until disconnected.
2897: (4)               Example usage::
2898: (8)                   ax = plt.subplot()
2899: (8)                   ax.plot(x, y)
2900: (8)                   def onselect(verts):
2901: (12)                      print(verts)
2902: (8)                   lasso = LassoSelector(ax, onselect)
2903: (4)               Parameters
2904: (4)               ----------
2905: (4)               ax : `~matplotlib.axes.Axes`
2906: (8)                   The parent Axes for the widget.
2907: (4)               onselect : function
2908: (8)                   Whenever the lasso is released, the *onselect* function is called and
2909: (8)                   passed the vertices of the selected path.
2910: (4)               useblit : bool, default: True
2911: (8)                   Whether to use blitting for faster drawing (if supported by the
2912: (8)                   backend). See the tutorial :ref:`blitting`
2913: (8)                   for details.
2914: (4)               props : dict, optional
2915: (8)                   Properties with which the line is drawn, see `.Line2D`
2916: (8)                   for valid properties. Default values are defined in ``mpl.rcParams``.
2917: (4)               button : `.MouseButton` or list of `.MouseButton`, optional
2918: (8)                   The mouse buttons used for rectangle selection.  Default is ``None``,
2919: (8)                   which corresponds to all buttons.
2920: (4)               """
2921: (4)               def __init__(self, ax, onselect, *, useblit=True, props=None,
button=None):
2922: (8)                   super().__init__(ax, onselect, useblit=useblit, button=button)
2923: (8)                   self.verts = None
2924: (8)                   props = {
2925: (12)                      **(props if props is not None else {}),
2926: (12)                      'animated': self.useblit, 'visible': False,
2927: (8)                   }
2928: (8)                   line = Line2D([], [], **props)
2929: (8)                   self.ax.add_line(line)
2930: (8)                   self._selection_artist = line
2931: (4)               def _press(self, event):
2932: (8)                   self.verts = [self._get_data(event)]
2933: (8)                   self._selection_artist.set_visible(True)
2934: (4)               def _release(self, event):
2935: (8)                   if self.verts is not None:
2936: (12)                      self.verts.append(self._get_data(event))
2937: (12)                      self.onselect(self.verts)
2938: (8)                   self._selection_artist.set_data([[], []])
2939: (8)                   self._selection_artist.set_visible(False)
2940: (8)                   self.verts = None
```

```
    def _onmove(self, event):
        if self.verts is None:
            return
        self.verts.append(self._get_data(event))
        self._selection_artist.set_data(list(zip(*self.verts)))
        self.update()
class PolygonSelector(_SelectorWidget):
    """
    Select a polygon region of an Axes.
    Place vertices with each mouse click, and make the selection by completing
    the polygon (clicking on the first vertex). Once drawn individual vertices
    can be moved by clicking and dragging with the left mouse button, or
    removed by clicking the right mouse button.
    In addition, the following modifier keys can be used:
    - Hold *ctrl* and click and drag a vertex to reposition it before the
      polygon has been completed.
    - Hold the *shift* key and click and drag anywhere in the Axes to move
      all vertices.
    - Press the *esc* key to start a new polygon.
    For the selector to remain responsive you must keep a reference to it.
    Parameters
    ----------
    ax : `~matplotlib.axes.Axes`
        The parent Axes for the widget.
    onselect : function
        When a polygon is completed or modified after completion,
        the *onselect* function is called and passed a list of the vertices as
        ``(xdata, ydata)`` tuples.
    useblit : bool, default: False
        Whether to use blitting for faster drawing (if supported by the
        backend). See the tutorial :ref:`blitting`
        for details.
    props : dict, optional
        Properties with which the line is drawn, see `.Line2D` for valid
properties.
        Default::
            dict(color='k', linestyle='-', linewidth=2, alpha=0.5)
    handle_props : dict, optional
        Artist properties for the markers drawn at the vertices of the
polygon.
        See the marker arguments in `.Line2D` for valid
        properties.  Default values are defined in ``mpl.rcParams`` except for
        the default value of ``markeredgecolor`` which will be the same as the
        ``color`` property in *props*.
    grab_range : float, default: 10
        A vertex is selected (to complete the polygon or to move a vertex) if
        the mouse click is within *grab_range* pixels of the vertex.
    draw_bounding_box : bool, optional
        If `True`, a bounding box will be drawn around the polygon selector
        once it is complete. This box can be used to move and resize the
        selector.
    box_handle_props : dict, optional
        Properties to set for the box handles. See the documentation for the
        *handle_props* argument to `RectangleSelector` for more info.
    box_props : dict, optional
        Properties to set for the box. See the documentation for the *props*
        argument to `RectangleSelector` for more info.
    Examples
    --------
    :doc:`/gallery/widgets/polygon_selector_simple`
    :doc:`/gallery/widgets/polygon_selector_demo`
    Notes
    -----
    If only one point remains after removing points, the selector reverts to
an
    incomplete state and you can start drawing a new polygon from the existing
    point.
    """
    def __init__(self, ax, onselect, *, useblit=False,
```

```
(17)                             props=None, handle_props=None, grab_range=10,
(17)                             draw_bounding_box=False, box_handle_props=None,
(17)                             box_props=None):
(8)                     state_modifier_keys = dict(clear='escape', move_vertex='control',
(35)                                                move_all='shift', move='not-applicable',
(35)                                                square='not-applicable',
(35)                                                center='not-applicable',
(35)                                                rotate='not-applicable')
(8)                     super().__init__(ax, onselect, useblit=useblit,
(25)                                     state_modifier_keys=state_modifier_keys)
(8)                     self._xys = [(0, 0)]
(8)                     if props is None:
(12)                        props = dict(color='k', linestyle='-', linewidth=2, alpha=0.5)
(8)                     props = {**props, 'animated': self.useblit}
(8)                     self._selection_artist = line = Line2D([], [], **props)
(8)                     self.ax.add_line(line)
(8)                     if handle_props is None:
(12)                        handle_props = dict(markeredgecolor='k',
(32)                                            markerfacecolor=props.get('color', 'k'))
(8)                     self._handle_props = handle_props
(8)                     self._polygon_handles = ToolHandles(self.ax, [], [],
(44)                                                        useblit=self.useblit,
(44)                                                        marker_props=self._handle_props)
(8)                     self._active_handle_idx = -1
(8)                     self.grab_range = grab_range
(8)                     self.set_visible(True)
(8)                     self._draw_box = draw_bounding_box
(8)                     self._box = None
(8)                     if box_handle_props is None:
(12)                        box_handle_props = {}
(8)                     self._box_handle_props = self._handle_props.update(box_handle_props)
(8)                     self._box_props = box_props
(4)                 def _get_bbox(self):
(8)                     return self._selection_artist.get_bbox()
(4)                 def _add_box(self):
(8)                     self._box = RectangleSelector(self.ax,
(38)                                                  onselect=lambda *args, **kwargs: None,
(38)                                                  useblit=self.useblit,
(38)                                                  grab_range=self.grab_range,
(38)                                                  handle_props=self._box_handle_props,
(38)                                                  props=self._box_props,
(38)                                                  interactive=True)
(8)                     self._box._state_modifier_keys.pop('rotate')
(8)                     self._box.connect_event('motion_notify_event', self._scale_polygon)
(8)                     self._update_box()
(8)                     self._box._allow_creation = False
(8)                     self._box._selection_completed = True
(8)                     self._draw_polygon()
(4)                 def _remove_box(self):
(8)                     if self._box is not None:
(12)                        self._box.set_visible(False)
(12)                        self._box = None
(4)                 def _update_box(self):
(8)                     if self._box is not None:
(12)                        bbox = self._get_bbox()
(12)                        self._box.extents = [bbox.x0, bbox.x1, bbox.y0, bbox.y1]
(12)                        self._old_box_extents = self._box.extents
(4)                 def _scale_polygon(self, event):
(8)                     """
(8)                     Scale the polygon selector points when the bounding box is moved or
(8)                     scaled.
(8)                     This is set as a callback on the bounding box RectangleSelector.
(8)                     """
(8)                     if not self._selection_completed:
(12)                        return
(8)                     if self._old_box_extents == self._box.extents:
(12)                        return
(8)                     x1, y1, w1, h1 = self._box._rect_bbox
(8)                     old_bbox = self._get_bbox()
```

```
(8)                 t = (transforms.Affine2D()
(13)                     .translate(-old_bbox.x0, -old_bbox.y0)
(13)                     .scale(1 / old_bbox.width, 1 / old_bbox.height)
(13)                     .scale(w1, h1)
(13)                     .translate(x1, y1))
(8)                 new_verts = [(x, y) for x, y in t.transform(np.array(self.verts))]
(8)                 self._xys = [*new_verts, new_verts[0]]
(8)                 self._draw_polygon()
(8)                 self._old_box_extents = self._box.extents
(4)             @property
(4)             def _handles_artists(self):
(8)                 return self._polygon_handles.artists
(4)             def _remove_vertex(self, i):
(8)                 """Remove vertex with index i."""
(8)                 if (len(self._xys) > 2 and
(16)                        self._selection_completed and
(16)                        i in (0, len(self._xys) - 1)):
(12)                    self._xys.pop(0)
(12)                    self._xys.pop(-1)
(12)                    self._xys.append(self._xys[0])
(8)                 else:
(12)                    self._xys.pop(i)
(8)                 if len(self._xys) <= 2:
(12)                    self._selection_completed = False
(12)                    self._remove_box()
(4)             def _press(self, event):
(8)                 """Button press event handler."""
(8)                 if ((self._selection_completed or 'move_vertex' in self._state)
(16)                        and len(self._xys) > 0):
(12)                    h_idx, h_dist = self._polygon_handles.closest(event.x, event.y)
(12)                    if h_dist < self.grab_range:
(16)                        self._active_handle_idx = h_idx
(8)                 self._xys_at_press = self._xys.copy()
(4)             def _release(self, event):
(8)                 """Button release event handler."""
(8)                 if self._active_handle_idx >= 0:
(12)                    if event.button == 3:
(16)                        self._remove_vertex(self._active_handle_idx)
(16)                        self._draw_polygon()
(12)                    self._active_handle_idx = -1
(8)                 elif len(self._xys) > 3 and self._xys[-1] == self._xys[0]:
(12)                    self._selection_completed = True
(12)                    if self._draw_box and self._box is None:
(16)                        self._add_box()
(8)                 elif (not self._selection_completed
(14)                      and 'move_all' not in self._state
(14)                      and 'move_vertex' not in self._state):
(12)                    self._xys.insert(-1, self._get_data_coords(event))
(8)                 if self._selection_completed:
(12)                    self.onselect(self.verts)
(4)             def onmove(self, event):
(8)                 """Cursor move event handler and validator."""
(8)                 if not self.ignore(event):
(12)                    event = self._clean_event(event)
(12)                    self._onmove(event)
(12)                    return True
(8)                 return False
(4)             def _onmove(self, event):
(8)                 """Cursor move event handler."""
(8)                 if self._active_handle_idx >= 0:
(12)                    idx = self._active_handle_idx
(12)                    self._xys[idx] = self._get_data_coords(event)
(12)                    if idx == 0 and self._selection_completed:
(16)                        self._xys[-1] = self._get_data_coords(event)
(8)                 elif 'move_all' in self._state and self._eventpress:
(12)                    xdata, ydata = self._get_data_coords(event)
(12)                    dx = xdata - self._eventpress.xdata
(12)                    dy = ydata - self._eventpress.ydata
(12)                    for k in range(len(self._xys)):
```

```
(16)                            x_at_press, y_at_press = self._xys_at_press[k]
(16)                            self._xys[k] = x_at_press + dx, y_at_press + dy
(8)                     elif (self._selection_completed
(14)                          or 'move_vertex' in self._state or 'move_all' in self._state):
(12)                        return
(8)                     else:
(12)                        x0, y0 = \
(16)                            self._selection_artist.get_transform().transform(self._xys[0])
(12)                        v0_dist = np.hypot(x0 - event.x, y0 - event.y)
(12)                        if len(self._xys) > 3 and v0_dist < self.grab_range:
(16)                            self._xys[-1] = self._xys[0]
(12)                        else:
(16)                            self._xys[-1] = self._get_data_coords(event)
(8)                     self._draw_polygon()
(4)                 def _on_key_press(self, event):
(8)                     """Key press event handler."""
(8)                     if (not self._selection_completed
(16)                            and ('move_vertex' in self._state or
(21)                                 'move_all' in self._state)):
(12)                        self._xys.pop()
(12)                        self._draw_polygon()
(4)                 def _on_key_release(self, event):
(8)                     """Key release event handler."""
(8)                     if (not self._selection_completed
(16)                            and
(16)                            (event.key == self._state_modifier_keys.get('move_vertex')
(17)                             or event.key == self._state_modifier_keys.get('move_all'))):
(12)                        self._xys.append(self._get_data_coords(event))
(12)                        self._draw_polygon()
(8)                     elif event.key == self._state_modifier_keys.get('clear'):
(12)                        event = self._clean_event(event)
(12)                        self._xys = [self._get_data_coords(event)]
(12)                        self._selection_completed = False
(12)                        self._remove_box()
(12)                        self.set_visible(True)
(4)                 def _draw_polygon_without_update(self):
(8)                     """Redraw the polygon based on new vertex positions, no update()."""
(8)                     xs, ys = zip(*self._xys) if self._xys else ([], [])
(8)                     self._selection_artist.set_data(xs, ys)
(8)                     self._update_box()
(8)                     if (self._selection_completed
(16)                            or (len(self._xys) > 3
(20)                                and self._xys[-1] == self._xys[0])):
(12)                        self._polygon_handles.set_data(xs[:-1], ys[:-1])
(8)                     else:
(12)                        self._polygon_handles.set_data(xs, ys)
(4)                 def _draw_polygon(self):
(8)                     """Redraw the polygon based on the new vertex positions."""
(8)                     self._draw_polygon_without_update()
(8)                     self.update()
(4)                 @property
(4)                 def verts(self):
(8)                     """The polygon vertices, as a list of ``(x, y)`` pairs."""
(8)                     return self._xys[:-1]
(4)                 @verts.setter
(4)                 def verts(self, xys):
(8)                     """
(8)                     Set the polygon vertices.
(8)                     This will remove any preexisting vertices, creating a complete polygon
(8)                     with the new vertices.
(8)                     """
(8)                     self._xys = [*xys, xys[0]]
(8)                     self._selection_completed = True
(8)                     self.set_visible(True)
(8)                     if self._draw_box and self._box is None:
(12)                        self._add_box()
(8)                     self._draw_polygon()
(4)                 def _clear_without_update(self):
(8)                     self._selection_completed = False
```

```
(8)                 self._xys = [(0, 0)]
(8)                 self._draw_polygon_without_update()
(0)         class Lasso(AxesWidget):
(4)             """
(4)             Selection curve of an arbitrary shape.
(4)             The selected path can be used in conjunction with
(4)             `~matplotlib.path.Path.contains_point` to select data points from an image.
(4)             Unlike `LassoSelector`, this must be initialized with a starting
(4)             point *xy*, and the `Lasso` events are destroyed upon release.
(4)             Parameters
(4)             ----------
(4)             ax : `~matplotlib.axes.Axes`
(8)                 The parent Axes for the widget.
(4)             xy : (float, float)
(8)                 Coordinates of the start of the lasso.
(4)             callback : callable
(8)                 Whenever the lasso is released, the *callback* function is called and
(8)                 passed the vertices of the selected path.
(4)             useblit : bool, default: True
(8)                 Whether to use blitting for faster drawing (if supported by the
(8)                 backend). See the tutorial :ref:`blitting`
(8)                 for details.
(4)             props: dict, optional
(8)                 Lasso line properties. See `.Line2D` for valid properties.
(8)                 Default *props* are::
(12)                    {'linestyle' : '-', 'color' : 'black', 'lw' : 2}
(8)                 .. versionadded:: 3.9
(4)             """
(4)             def __init__(self, ax, xy, callback, *, useblit=True, props=None):
(8)                 super().__init__(ax)
(8)                 self.useblit = useblit and self.canvas.supports_blit
(8)                 if self.useblit:
(12)                    self.background = self.canvas.copy_from_bbox(self.ax.bbox)
(8)                 style = {'linestyle': '-', 'color': 'black', 'lw': 2}
(8)                 if props is not None:
(12)                    style.update(props)
(8)                 x, y = xy
(8)                 self.verts = [(x, y)]
(8)                 self.line = Line2D([x], [y], **style)
(8)                 self.ax.add_line(self.line)
(8)                 self.callback = callback
(8)                 self.connect_event('button_release_event', self.onrelease)
(8)                 self.connect_event('motion_notify_event', self.onmove)
(4)             def onrelease(self, event):
(8)                 if self.ignore(event):
(12)                    return
(8)                 if self.verts is not None:
(12)                    self.verts.append(self._get_data_coords(event))
(12)                    if len(self.verts) > 2:
(16)                        self.callback(self.verts)
(12)                    self.line.remove()
(8)                 self.verts = None
(8)                 self.disconnect_events()
(4)             def onmove(self, event):
(8)                 if (self.ignore(event)
(16)                        or self.verts is None
(16)                        or event.button != 1
(16)                        or not self.ax.contains(event)[0]):
(12)                    return
(8)                 self.verts.append(self._get_data_coords(event))
(8)                 self.line.set_data(list(zip(*self.verts)))
(8)                 if self.useblit:
(12)                    self.canvas.restore_region(self.background)
(12)                    self.ax.draw_artist(self.line)
(12)                    self.canvas.blit(self.ax.bbox)
(8)                 else:
(12)                    self.canvas.draw_idle()
```

```
----------------------------------------

File 72 - __init__.py:

1: (0)              """
2: (0)              Helper functions for managing the Matplotlib API.
3: (0)              This documentation is only relevant for Matplotlib developers, not for users.
4: (0)              .. warning::
5: (4)                  This module and its submodules are for internal use only.  Do not use them
6: (4)                  in your own code.  We may change the API at any time with no warning.
7: (0)              """
8: (0)              import functools
9: (0)              import itertools
10: (0)             import re
11: (0)             import sys
12: (0)             import warnings
13: (0)             from .deprecation import (  # noqa: F401
14: (4)                 deprecated, warn_deprecated,
15: (4)                 rename_parameter, delete_parameter, make_keyword_only,
16: (4)                 deprecate_method_override, deprecate_privatize_attribute,
17: (4)                 suppress_matplotlib_deprecation_warning,
18: (4)                 MatplotlibDeprecationWarning)
19: (0)             class classproperty:
20: (4)                 """
21: (4)                 Like `property`, but also triggers on access via the class, and it is the
22: (4)                 *class* that's passed as argument.
23: (4)                 Examples
24: (4)                 --------
25: (4)                 ::
26: (8)                     class C:
27: (12)                        @classproperty
28: (12)                        def foo(cls):
29: (16)                            return cls.__name__
30: (8)                     assert C.foo == "C"
31: (4)                 """
32: (4)                 def __init__(self, fget, fset=None, fdel=None, doc=None):
33: (8)                     self._fget = fget
34: (8)                     if fset is not None or fdel is not None:
35: (12)                        raise ValueError('classproperty only implements fget.')
36: (8)                     self.fset = fset
37: (8)                     self.fdel = fdel
38: (8)                     self._doc = doc
39: (4)                 def __get__(self, instance, owner):
40: (8)                     return self._fget(owner)
41: (4)                 @property
42: (4)                 def fget(self):
43: (8)                     return self._fget
44: (0)             def check_isinstance(types, /, **kwargs):
45: (4)                 """
46: (4)                 For each *key, value* pair in *kwargs*, check that *value* is an instance
47: (4)                 of one of *types*; if not, raise an appropriate TypeError.
48: (4)                 As a special case, a ``None`` entry in *types* is treated as NoneType.
49: (4)                 Examples
50: (4)                 --------
51: (4)                 >>> _api.check_isinstance((SomeClass, None), arg=arg)
52: (4)                 """
53: (4)                 none_type = type(None)
54: (4)                 types = ((types,) if isinstance(types, type) else
55: (13)                         (none_type,) if types is None else
56: (13)                         tuple(none_type if tp is None else tp for tp in types))
57: (4)                 def type_name(tp):
58: (8)                     return ("None" if tp is none_type
59: (16)                            else tp.__qualname__ if tp.__module__ == "builtins"
60: (16)                            else f"{tp.__module__}.{tp.__qualname__}")
61: (4)                 for k, v in kwargs.items():
62: (8)                     if not isinstance(v, types):
63: (12)                        names = [*map(type_name, types)]
64: (12)                        if "None" in names:  # Move it to the end for better wording.
65: (16)                            names.remove("None")
```

```
 66: (16)                            names.append("None")
 67: (12)                        raise TypeError(
 68: (16)                            "{!r} must be an instance of {}, not a {}".format(
 69: (20)                                k,
 70: (20)                                ", ".join(names[:-1]) + " or " + names[-1]
 71: (20)                                if len(names) > 1 else names[0],
 72: (20)                                type_name(type(v))))
 73: (0)             def check_in_list(values, /, *, _print_supported_values=True, **kwargs):
 74: (4)                 """
 75: (4)                 For each *key, value* pair in *kwargs*, check that *value* is in *values*;
 76: (4)                 if not, raise an appropriate ValueError.
 77: (4)                 Parameters
 78: (4)                 ----------
 79: (4)                 values : iterable
 80: (8)                     Sequence of values to check on.
 81: (4)                 _print_supported_values : bool, default: True
 82: (8)                     Whether to print *values* when raising ValueError.
 83: (4)                 **kwargs : dict
 84: (8)                     *key, value* pairs as keyword arguments to find in *values*.
 85: (4)                 Raises
 86: (4)                 ------
 87: (4)                 ValueError
 88: (8)                     If any *value* in *kwargs* is not found in *values*.
 89: (4)                 Examples
 90: (4)                 --------
 91: (4)                 >>> _api.check_in_list(["foo", "bar"], arg=arg, other_arg=other_arg)
 92: (4)                 """
 93: (4)                 if not kwargs:
 94: (8)                     raise TypeError("No argument to check!")
 95: (4)                 for key, val in kwargs.items():
 96: (8)                     if val not in values:
 97: (12)                        msg = f"{val!r} is not a valid value for {key}"
 98: (12)                        if _print_supported_values:
 99: (16)                            msg += f"; supported values are {', '.join(map(repr,
values))}"
100: (12)                        raise ValueError(msg)
101: (0)             def check_shape(shape, /, **kwargs):
102: (4)                 """
103: (4)                 For each *key, value* pair in *kwargs*, check that *value* has the shape
*shape*;
104: (4)                 if not, raise an appropriate ValueError.
105: (4)                 *None* in the shape is treated as a "free" size that can have any length.
106: (4)                 e.g. (None, 2) -> (N, 2)
107: (4)                 The values checked must be numpy arrays.
108: (4)                 Examples
109: (4)                 --------
110: (4)                 To check for (N, 2) shaped arrays
111: (4)                 >>> _api.check_shape((None, 2), arg=arg, other_arg=other_arg)
112: (4)                 """
113: (4)                 for k, v in kwargs.items():
114: (8)                     data_shape = v.shape
115: (8)                     if (len(data_shape) != len(shape)
116: (16)                            or any(s != t and t is not None for s, t in zip(data_shape,
shape))):
117: (12)                        dim_labels = iter(itertools.chain(
118: (16)                            'NMLKJIH',
119: (16)                            (f"D{i}" for i in itertools.count())))
120: (12)                        text_shape = ", ".join([str(n) if n is not None else
next(dim_labels)
121: (36)                                                for n in shape[::-1]][::-1])
122: (12)                        if len(shape) == 1:
123: (16)                            text_shape += ","
124: (12)                        raise ValueError(
125: (16)                            f"{k!r} must be {len(shape)}D with shape ({text_shape}), "
126: (16)                            f"but your input has shape {v.shape}"
127: (12)                        )
128: (0)             def check_getitem(mapping, /, **kwargs):
129: (4)                 """
130: (4)                 *kwargs* must consist of a single *key, value* pair.  If *key* is in
```

```
131: (4)                *mapping*, return ``mapping[value]``; else, raise an appropriate
132: (4)                ValueError.
133: (4)                Examples
134: (4)                --------
135: (4)                >>> _api.check_getitem({"foo": "bar"}, arg=arg)
136: (4)                """
137: (4)                if len(kwargs) != 1:
138: (8)                    raise ValueError("check_getitem takes a single keyword argument")
139: (4)                (k, v), = kwargs.items()
140: (4)                try:
141: (8)                    return mapping[v]
142: (4)                except KeyError:
143: (8)                    raise ValueError(
144: (12)                       f"{v!r} is not a valid value for {k}; supported values are "
145: (12)                       f"{', '.join(map(repr, mapping))}") from None
146: (0)            def caching_module_getattr(cls):
147: (4)                """
148: (4)                Helper decorator for implementing module-level ``__getattr__`` as a class.
149: (4)                This decorator must be used at the module toplevel as follows::
150: (8)                    @caching_module_getattr
151: (8)                    class __getattr__:  # The class *must* be named ``__getattr__``.
152: (12)                       @property  # Only properties are taken into account.
153: (12)                       def name(self): ...
154: (4)                The ``__getattr__`` class will be replaced by a ``__getattr__``
155: (4)                function such that trying to access ``name`` on the module will
156: (4)                resolve the corresponding property (which may be decorated e.g. with
157: (4)                ``_api.deprecated`` for deprecating module globals).  The properties are
158: (4)                all implicitly cached.  Moreover, a suitable AttributeError is generated
159: (4)                and raised if no property with the given name exists.
160: (4)                """
161: (4)                assert cls.__name__ == "__getattr__"
162: (4)                props = {name: prop for name, prop in vars(cls).items()
163: (13)                         if isinstance(prop, property)}
164: (4)                instance = cls()
165: (4)                @functools.cache
166: (4)                def __getattr__(name):
167: (8)                    if name in props:
168: (12)                       return props[name].__get__(instance)
169: (8)                    raise AttributeError(
170: (12)                       f"module {cls.__module__!r} has no attribute {name!r}")
171: (4)                return __getattr__
172: (0)            def define_aliases(alias_d, cls=None):
173: (4)                """
174: (4)                Class decorator for defining property aliases.
175: (4)                Use as ::
176: (8)                    @_api.define_aliases({"property": ["alias", ...], ...})
177: (8)                    class C: ...
178: (4)                For each property, if the corresponding ``get_property`` is defined in the
179: (4)                class so far, an alias named ``get_alias`` will be defined; the same will
180: (4)                be done for setters.  If neither the getter nor the setter exists, an
181: (4)                exception will be raised.
182: (4)                The alias map is stored as the ``_alias_map`` attribute on the class and
183: (4)                can be used by `.normalize_kwargs` (which assumes that higher priority
184: (4)                aliases come last).
185: (4)                """
186: (4)                if cls is None:  # Return the actual class decorator.
187: (8)                    return functools.partial(define_aliases, alias_d)
188: (4)                def make_alias(name):  # Enforce a closure over *name*.
189: (8)                    @functools.wraps(getattr(cls, name))
190: (8)                    def method(self, *args, **kwargs):
191: (12)                       return getattr(self, name)(*args, **kwargs)
192: (8)                    return method
193: (4)                for prop, aliases in alias_d.items():
194: (8)                    exists = False
195: (8)                    for prefix in ["get_", "set_"]:
196: (12)                       if prefix + prop in vars(cls):
197: (16)                           exists = True
198: (16)                           for alias in aliases:
199: (20)                               method = make_alias(prefix + prop)
```

```
200: (20)                                method.__name__ = prefix + alias
201: (20)                                method.__doc__ = f"Alias for `{prefix + prop}`."
202: (20)                                setattr(cls, prefix + alias, method)
203: (8)                    if not exists:
204: (12)                       raise ValueError(
205: (16)                           f"Neither getter nor setter exists for {prop!r}")
206: (4)                def get_aliased_and_aliases(d):
207: (8)                    return {*d, *(alias for aliases in d.values() for alias in aliases)}
208: (4)                preexisting_aliases = getattr(cls, "_alias_map", {})
209: (4)                conflicting = (get_aliased_and_aliases(preexisting_aliases)
210: (19)                              & get_aliased_and_aliases(alias_d))
211: (4)                if conflicting:
212: (8)                    raise NotImplementedError(
213: (12)                       f"Parent class already defines conflicting aliases:
{conflicting}")
214: (4)                cls._alias_map = {**preexisting_aliases, **alias_d}
215: (4)                return cls
216: (0)            def select_matching_signature(funcs, *args, **kwargs):
217: (4)                """
218: (4)                Select and call the function that accepts ``*args, **kwargs``.
219: (4)                *funcs* is a list of functions which should not raise any exception (other
220: (4)                than `TypeError` if the arguments passed do not match their signature).
221: (4)                `select_matching_signature` tries to call each of the functions in *funcs*
222: (4)                with ``*args, **kwargs`` (in the order in which they are given).  Calls
223: (4)                that fail with a `TypeError` are silently skipped.  As soon as a call
224: (4)                succeeds, `select_matching_signature` returns its return value.  If no
225: (4)                function accepts ``*args, **kwargs``, then the `TypeError` raised by the
226: (4)                last failing call is re-raised.
227: (4)                Callers should normally make sure that any ``*args, **kwargs`` can only
228: (4)                bind a single *func* (to avoid any ambiguity), although this is not
checked
229: (4)                by `select_matching_signature`.
230: (4)                Notes
231: (4)                -----
232: (4)                `select_matching_signature` is intended to help implementing
233: (4)                signature-overloaded functions.  In general, such functions should be
234: (4)                avoided, except for back-compatibility concerns.  A typical use pattern is
235: (4)                ::
236: (8)                    def my_func(*args, **kwargs):
237: (12)                       params = select_matching_signature(
238: (16)                           [lambda old1, old2: locals(), lambda new: locals()],
239: (16)                           *args, **kwargs)
240: (12)                       if "old1" in params:
241: (16)                           warn_deprecated(...)
242: (16)                           old1, old2 = params.values()  # note that locals() is ordered.
243: (12)                       else:
244: (16)                           new, = params.values()
245: (4)                which allows *my_func* to be called either with two parameters (*old1* and
246: (4)                *old2*) or a single one (*new*).  Note that the new signature is given
247: (4)                last, so that callers get a `TypeError` corresponding to the new signature
248: (4)                if the arguments they passed in do not match any signature.
249: (4)                """
250: (4)                for i, func in enumerate(funcs):
251: (8)                    try:
252: (12)                       return func(*args, **kwargs)
253: (8)                    except TypeError:
254: (12)                       if i == len(funcs) - 1:
255: (16)                           raise
256: (0)            def nargs_error(name, takes, given):
257: (4)                """Generate a TypeError to be raised by function calls with wrong
arity."""
258: (4)                return TypeError(f"{name}() takes {takes} positional arguments but "
259: (21)                                 f"{given} were given")
260: (0)            def kwarg_error(name, kw):
261: (4)                """
262: (4)                Generate a TypeError to be raised by function calls with wrong kwarg.
263: (4)                Parameters
264: (4)                ----------
265: (4)                name : str
```

```
266: (8)                    The name of the calling function.
267: (4)                kw : str or Iterable[str]
268: (8)                    Either the invalid keyword argument name, or an iterable yielding
269: (8)                    invalid keyword arguments (e.g., a ``kwargs`` dict).
270: (4)                """
271: (4)                if not isinstance(kw, str):
272: (8)                    kw = next(iter(kw))
273: (4)                return TypeError(f"{name}() got an unexpected keyword argument '{kw}'")
274: (0)            def recursive_subclasses(cls):
275: (4)                """Yield *cls* and direct and indirect subclasses of *cls*."""
276: (4)                yield cls
277: (4)                for subcls in cls.__subclasses__():
278: (8)                    yield from recursive_subclasses(subcls)
279: (0)            def warn_external(message, category=None):
280: (4)                """
281: (4)                `warnings.warn` wrapper that sets *stacklevel* to "outside Matplotlib".
282: (4)                The original emitter of the warning can be obtained by patching this
283: (4)                function back to `warnings.warn`, i.e. ``_api.warn_external =
284: (4)                warnings.warn`` (or ``functools.partial(warnings.warn, stacklevel=2)``,
285: (4)                etc.).
286: (4)                """
287: (4)                frame = sys._getframe()
288: (4)                for stacklevel in itertools.count(1):
289: (8)                    if frame is None:
290: (12)                       break
291: (8)                    if not re.match(r"\A(matplotlib|mpl_toolkits)(\Z|\.(?!tests\.))",
292: (24)                                   frame.f_globals.get("__name__", "")):
293: (12)                       break
294: (8)                    frame = frame.f_back
295: (4)                del frame
296: (4)                warnings.warn(message, category, stacklevel)
```

File 73 - deprecation.py:

```
1: (0)              """
2: (0)              Helper functions for deprecating parts of the Matplotlib API.
3: (0)              This documentation is only relevant for Matplotlib developers, not for users.
4: (0)              .. warning::
5: (4)                  This module is for internal use only.  Do not use it in your own code.
6: (4)                  We may change the API at any time with no warning.
7: (0)              """
8: (0)              import contextlib
9: (0)              import functools
10: (0)             import inspect
11: (0)             import math
12: (0)             import warnings
13: (0)             class MatplotlibDeprecationWarning(DeprecationWarning):
14: (4)                 """A class for issuing deprecation warnings for Matplotlib users."""
15: (0)             def _generate_deprecation_warning(
16: (8)                     since, message='', name='', alternative='', pending=False,
obj_type='',
17: (8)                     addendum='', *, removal=''):
18: (4)                 if pending:
19: (8)                     if removal:
20: (12)                        raise ValueError(
21: (16)                            "A pending deprecation cannot have a scheduled removal")
22: (4)                 else:
23: (8)                     if not removal:
24: (12)                        macro, meso, *_ = since.split('.')
25: (12)                        removal = f'{macro}.{int(meso) + 2}'
26: (8)                     removal = f"in {removal}"
27: (4)                 if not message:
28: (8)                     message = (
29: (12)                        ("The %(name)s %(obj_type)s" if obj_type else "%(name)s")
30: (12)                        + (" will be deprecated in a future version"
31: (15)                           if pending else
32: (15)                           " was deprecated in Matplotlib %(since)s and will be removed %
```

```
(removal)s"
33: (15)                            )
34: (12)                        + "."
35: (12)                        + (" Use %(alternative)s instead." if alternative else "")
36: (12)                        + (" %(addendum)s" if addendum else ""))
37: (4)                 warning_cls = (PendingDeprecationWarning if pending
38: (19)                               else MatplotlibDeprecationWarning)
39: (4)                 return warning_cls(message % dict(
40: (8)                     func=name, name=name, obj_type=obj_type, since=since, removal=removal,
41: (8)                     alternative=alternative, addendum=addendum))
42: (0)             def warn_deprecated(
43: (8)                     since, *, message='', name='', alternative='', pending=False,
44: (8)                     obj_type='', addendum='', removal=''):
45: (4)                 """
46: (4)                 Display a standardized deprecation.
47: (4)                 Parameters
48: (4)                 ----------
49: (4)                 since : str
50: (8)                     The release at which this API became deprecated.
51: (4)                 message : str, optional
52: (8)                     Override the default deprecation message.  The ``%(since)s``,
53: (8)                     ``%(name)s``, ``%(alternative)s``, ``%(obj_type)s``, ``%(addendum)s``,
54: (8)                     and ``%(removal)s`` format specifiers will be replaced by the values
55: (8)                     of the respective arguments passed to this function.
56: (4)                 name : str, optional
57: (8)                     The name of the deprecated object.
58: (4)                 alternative : str, optional
59: (8)                     An alternative API that the user may use in place of the deprecated
60: (8)                     API.  The deprecation warning will tell the user about this
alternative
61: (8)                     if provided.
62: (4)                 pending : bool, optional
63: (8)                     If True, uses a PendingDeprecationWarning instead of a
64: (8)                     DeprecationWarning.  Cannot be used together with *removal*.
65: (4)                 obj_type : str, optional
66: (8)                     The object type being deprecated.
67: (4)                 addendum : str, optional
68: (8)                     Additional text appended directly to the final message.
69: (4)                 removal : str, optional
70: (8)                     The expected removal version.  With the default (an empty string), a
71: (8)                     removal version is automatically computed from *since*.  Set to other
72: (8)                     Falsy values to not schedule a removal date.  Cannot be used together
73: (8)                     with *pending*.
74: (4)                 Examples
75: (4)                 --------
76: (4)                 ::
77: (8)                     warn_deprecated('1.4.0', name='matplotlib.name_of_module',
78: (24)                                    obj_type='module')
79: (4)                 """
80: (4)                 warning = _generate_deprecation_warning(
81: (8)                     since, message, name, alternative, pending, obj_type, addendum,
82: (8)                     removal=removal)
83: (4)                 from . import warn_external
84: (4)                 warn_external(warning, category=MatplotlibDeprecationWarning)
85: (0)             def deprecated(since, *, message='', name='', alternative='', pending=False,
86: (15)                           obj_type=None, addendum='', removal=''):
87: (4)                 """
88: (4)                 Decorator to mark a function, a class, or a property as deprecated.
89: (4)                 When deprecating a classmethod, a staticmethod, or a property, the
90: (4)                 ``@deprecated`` decorator should go *under* ``@classmethod`` and
91: (4)                 ``@staticmethod`` (i.e., `deprecated` should directly decorate the
92: (4)                 underlying callable), but *over* ``@property``.
93: (4)                 When deprecating a class ``C`` intended to be used as a base class in a
94: (4)                 multiple inheritance hierarchy, ``C`` *must* define an ``__init__`` method
95: (4)                 (if ``C`` instead inherited its ``__init__`` from its own base class, then
96: (4)                 ``@deprecated`` would mess up ``__init__`` inheritance when installing its
97: (4)                 own (deprecation-emitting) ``C.__init__``).
98: (4)                 Parameters are the same as for `warn_deprecated`, except that *obj_type*
99: (4)                 defaults to 'class' if decorating a class, 'attribute' if decorating a
```

```
(4)                 property, and 'function' otherwise.
(4)                 Examples
(4)                 --------
(4)                 ::
(8)                     @deprecated('1.4.0')
(8)                     def the_function_to_deprecate():
(12)                        pass
(4)                 """
(4)                 def deprecate(obj, message=message, name=name, alternative=alternative,
(18)                              pending=pending, obj_type=obj_type, addendum=addendum):
(8)                     from matplotlib._api import classproperty
(8)                     if isinstance(obj, type):
(12)                        if obj_type is None:
(16)                            obj_type = "class"
(12)                        func = obj.__init__
(12)                        name = name or obj.__name__
(12)                        old_doc = obj.__doc__
(12)                        def finalize(wrapper, new_doc):
(16)                            try:
(20)                                obj.__doc__ = new_doc
(16)                            except AttributeError:  # Can't set on some extension objects.
(20)                                pass
(16)                            obj.__init__ = functools.wraps(obj.__init__)(wrapper)
(16)                            return obj
(8)                     elif isinstance(obj, (property, classproperty)):
(12)                        if obj_type is None:
(16)                            obj_type = "attribute"
(12)                        func = None
(12)                        name = name or obj.fget.__name__
(12)                        old_doc = obj.__doc__
(12)                        class _deprecated_property(type(obj)):
(16)                            def __get__(self, instance, owner=None):
(20)                                if instance is not None or owner is not None \
(28)                                        and isinstance(self, classproperty):
(24)                                    emit_warning()
(20)                                return super().__get__(instance, owner)
(16)                            def __set__(self, instance, value):
(20)                                if instance is not None:
(24)                                    emit_warning()
(20)                                return super().__set__(instance, value)
(16)                            def __delete__(self, instance):
(20)                                if instance is not None:
(24)                                    emit_warning()
(20)                                return super().__delete__(instance)
(16)                            def __set_name__(self, owner, set_name):
(20)                                nonlocal name
(20)                                if name == "<lambda>":
(24)                                    name = set_name
(12)                        def finalize(_, new_doc):
(16)                            return _deprecated_property(
(20)                                fget=obj.fget, fset=obj.fset, fdel=obj.fdel, doc=new_doc)
(8)                     else:
(12)                        if obj_type is None:
(16)                            obj_type = "function"
(12)                        func = obj
(12)                        name = name or obj.__name__
(12)                        old_doc = func.__doc__
(12)                        def finalize(wrapper, new_doc):
(16)                            wrapper = functools.wraps(func)(wrapper)
(16)                            wrapper.__doc__ = new_doc
(16)                            return wrapper
(8)                     def emit_warning():
(12)                        warn_deprecated(
(16)                            since, message=message, name=name, alternative=alternative,
(16)                            pending=pending, obj_type=obj_type, addendum=addendum,
(16)                            removal=removal)
(8)                     def wrapper(*args, **kwargs):
(12)                        emit_warning()
(12)                        return func(*args, **kwargs)
```

```
169: (8)                    old_doc = inspect.cleandoc(old_doc or '').strip('\n')
170: (8)                    notes_header = '\nNotes\n-----'
171: (8)                    second_arg = ' '.join([t.strip() for t in
172: (31)                                         (message, f"Use {alternative} instead."
173: (32)                                          if alternative else "", addendum) if t])
174: (8)                    new_doc = (f"[*Deprecated*] {old_doc}\n"
175: (19)                               f"{notes_header if notes_header not in old_doc else ''}\n"
176: (19)                               f".. deprecated:: {since}\n"
177: (19)                               f"   {second_arg}")
178: (8)                    if not old_doc:
179: (12)                       new_doc += r'\ '
180: (8)                    return finalize(wrapper, new_doc)
181: (4)                return deprecate
182: (0)            class deprecate_privatize_attribute:
183: (4)                """
184: (4)                Helper to deprecate public access to an attribute (or method).
185: (4)                This helper should only be used at class scope, as follows::
186: (8)                    class Foo:
187: (12)                       attr = _deprecate_privatize_attribute(*args, **kwargs)
188: (4)                where *all* parameters are forwarded to `deprecated`.  This form makes
189: (4)                ``attr`` a property which forwards read and write access to ``self._attr``
190: (4)                (same name but with a leading underscore), with a deprecation warning.
191: (4)                Note that the attribute name is derived from *the name this helper is
192: (4)                assigned to*.  This helper also works for deprecating methods.
193: (4)                """
194: (4)                def __init__(self, *args, **kwargs):
195: (8)                    self.deprecator = deprecated(*args, **kwargs)
196: (4)                def __set_name__(self, owner, name):
197: (8)                    setattr(owner, name, self.deprecator(
198: (12)                       property(lambda self: getattr(self, f"_{name}"),
199: (21)                                lambda self, value: setattr(self, f"_{name}", value)),
200: (12)                       name=name))
201: (0)            DECORATORS = {}
202: (0)            def rename_parameter(since, old, new, func=None):
203: (4)                """
204: (4)                Decorator indicating that parameter *old* of *func* is renamed to *new*.
205: (4)                The actual implementation of *func* should use *new*, not *old*.  If *old*
206: (4)                is passed to *func*, a DeprecationWarning is emitted, and its value is
207: (4)                used, even if *new* is also passed by keyword (this is to simplify pyplot
208: (4)                wrapper functions, which always pass *new* explicitly to the Axes method).
209: (4)                If *new* is also passed but positionally, a TypeError will be raised by
the
210: (4)                underlying function during argument binding.
211: (4)                Examples
212: (4)                --------
213: (4)                ::
214: (8)                    @_api.rename_parameter("3.1", "bad_name", "good_name")
215: (8)                    def func(good_name): ...
216: (4)                """
217: (4)                decorator = functools.partial(rename_parameter, since, old, new)
218: (4)                if func is None:
219: (8)                    return decorator
220: (4)                signature = inspect.signature(func)
221: (4)                assert old not in signature.parameters, (
222: (8)                    f"Matplotlib internal error: {old!r} cannot be a parameter for "
223: (8)                    f"{func.__name__}()")
224: (4)                assert new in signature.parameters, (
225: (8)                    f"Matplotlib internal error: {new!r} must be a parameter for "
226: (8)                    f"{func.__name__}()")
227: (4)                @functools.wraps(func)
228: (4)                def wrapper(*args, **kwargs):
229: (8)                    if old in kwargs:
230: (12)                       warn_deprecated(
231: (16)                           since, message=f"The {old!r} parameter of {func.__name__}() "
232: (16)                           f"has been renamed {new!r} since Matplotlib {since}; support "
233: (16)                           f"for the old name will be dropped %(removal)s.")
234: (12)                       kwargs[new] = kwargs.pop(old)
235: (8)                    return func(*args, **kwargs)
236: (4)                DECORATORS[wrapper] = decorator
```

```
(4)                 return wrapper
(0)             class _deprecated_parameter_class:
(4)                 def __repr__(self):
(8)                     return "<deprecated parameter>"
(0)             _deprecated_parameter = _deprecated_parameter_class()
(0)             def delete_parameter(since, name, func=None, **kwargs):
(4)                 """
(4)                 Decorator indicating that parameter *name* of *func* is being deprecated.
(4)                 The actual implementation of *func* should keep the *name* parameter in
its
(4)                 signature, or accept a ``**kwargs`` argument (through which *name* would
be
(4)                 passed).
(4)                 Parameters that come after the deprecated parameter effectively become
(4)                 keyword-only (as they cannot be passed positionally without triggering the
(4)                 DeprecationWarning on the deprecated parameter), and should be marked as
(4)                 such after the deprecation period has passed and the deprecated parameter
(4)                 is removed.
(4)                 Parameters other than *since*, *name*, and *func* are keyword-only and
(4)                 forwarded to `.warn_deprecated`.
(4)                 Examples
(4)                 --------
(4)                 ::
(8)                     @_api.delete_parameter("3.1", "unused")
(8)                     def func(used_arg, other_arg, unused, more_args): ...
(4)                 """
(4)                 decorator = functools.partial(delete_parameter, since, name, **kwargs)
(4)                 if func is None:
(8)                     return decorator
(4)                 signature = inspect.signature(func)
(4)                 kwargs_name = next((param.name for param in signature.parameters.values()
(24)                                    if param.kind == inspect.Parameter.VAR_KEYWORD), None)
(4)                 if name in signature.parameters:
(8)                     kind = signature.parameters[name].kind
(8)                     is_varargs = kind is inspect.Parameter.VAR_POSITIONAL
(8)                     is_varkwargs = kind is inspect.Parameter.VAR_KEYWORD
(8)                     if not is_varargs and not is_varkwargs:
(12)                        name_idx = (
(16)                            math.inf if kind is inspect.Parameter.KEYWORD_ONLY
(16)                            else [*signature.parameters].index(name))
(12)                        func.__signature__ = signature = signature.replace(parameters=[
(16)                            param.replace(default=_deprecated_parameter)
(16)                            if param.name == name else param
(16)                            for param in signature.parameters.values()])
(8)                     else:
(12)                        name_idx = -1  # Deprecated parameter can always have been passed.
(4)                 else:
(8)                     is_varargs = is_varkwargs = False
(8)                     name_idx = math.inf
(8)                     assert kwargs_name, (
(12)                        f"Matplotlib internal error: {name!r} must be a parameter for "
(12)                        f"{func.__name__}()")
(4)                 addendum = kwargs.pop('addendum', None)
(4)                 @functools.wraps(func)
(4)                 def wrapper(*inner_args, **inner_kwargs):
(8)                     if len(inner_args) <= name_idx and name not in inner_kwargs:
(12)                        return func(*inner_args, **inner_kwargs)
(8)                     arguments = signature.bind(*inner_args, **inner_kwargs).arguments
(8)                     if is_varargs and arguments.get(name):
(12)                        warn_deprecated(
(16)                            since, message=f"Additional positional arguments to "
(16)                            f"{func.__name__}() are deprecated since %(since)s and "
(16)                            f"support for them will be removed %(removal)s.")
(8)                     elif is_varkwargs and arguments.get(name):
(12)                        warn_deprecated(
(16)                            since, message=f"Additional keyword arguments to "
(16)                            f"{func.__name__}() are deprecated since %(since)s and "
(16)                            f"support for them will be removed %(removal)s.")
(8)                     elif any(name in d and d[name] != _deprecated_parameter
```

```
304: (17)                            for d in [arguments, arguments.get(kwargs_name, {})]):
305: (12)                        deprecation_addendum = (
306: (16)                            f"If any parameter follows {name!r}, they should be passed as
"
307: (16)                            f"keyword, not positionally.")
308: (12)                        warn_deprecated(
309: (16)                            since,
310: (16)                            name=repr(name),
311: (16)                            obj_type=f"parameter of {func.__name__}()",
312: (16)                            addendum=(addendum + " " + deprecation_addendum) if addendum
313: (25)                                     else deprecation_addendum,
314: (16)                            **kwargs)
315: (8)                    return func(*inner_args, **inner_kwargs)
316: (4)                DECORATORS[wrapper] = decorator
317: (4)                return wrapper
318: (0)            def make_keyword_only(since, name, func=None):
319: (4)                """
320: (4)                Decorator indicating that passing parameter *name* (or any of the
following
321: (4)                ones) positionally to *func* is being deprecated.
322: (4)                When used on a method that has a pyplot wrapper, this should be the
323: (4)                outermost decorator, so that :file:`boilerplate.py` can access the
original
324: (4)                signature.
325: (4)                """
326: (4)                decorator = functools.partial(make_keyword_only, since, name)
327: (4)                if func is None:
328: (8)                    return decorator
329: (4)                signature = inspect.signature(func)
330: (4)                POK = inspect.Parameter.POSITIONAL_OR_KEYWORD
331: (4)                KWO = inspect.Parameter.KEYWORD_ONLY
332: (4)                assert (name in signature.parameters
333: (12)                        and signature.parameters[name].kind == POK), (
334: (8)                    f"Matplotlib internal error: {name!r} must be a positional-or-keyword
"
335: (8)                    f"parameter for {func.__name__}()")
336: (4)                names = [*signature.parameters]
337: (4)                name_idx = names.index(name)
338: (4)                kwonly = [name for name in names[name_idx:]
339: (14)                         if signature.parameters[name].kind == POK]
340: (4)                @functools.wraps(func)
341: (4)                def wrapper(*args, **kwargs):
342: (8)                    if len(args) > name_idx:
343: (12)                       warn_deprecated(
344: (16)                           since, message="Passing the %(name)s %(obj_type)s "
345: (16)                           "positionally is deprecated since Matplotlib %(since)s; the "
346: (16)                           "parameter will become keyword-only %(removal)s.",
347: (16)                           name=name, obj_type=f"parameter of {func.__name__}()")
348: (8)                    return func(*args, **kwargs)
349: (4)                wrapper.__signature__ = signature.replace(parameters=[
350: (8)                    param.replace(kind=KWO) if param.name in kwonly else param
351: (8)                    for param in signature.parameters.values()])
352: (4)                DECORATORS[wrapper] = decorator
353: (4)                return wrapper
354: (0)            def deprecate_method_override(method, obj, *, allow_empty=False, **kwargs):
355: (4)                """
356: (4)                Return ``obj.method`` with a deprecation if it was overridden, else None.
357: (4)                Parameters
358: (4)                ----------
359: (4)                method
360: (8)                    An unbound method, i.e. an expression of the form
361: (8)                    ``Class.method_name``.  Remember that within the body of a method, one
362: (8)                    can always use ``__class__`` to refer to the class that is currently
363: (8)                    being defined.
364: (4)                obj
365: (8)                    Either an object of the class where *method* is defined, or a subclass
366: (8)                    of that class.
367: (4)                allow_empty : bool, default: False
368: (8)                    Whether to allow overrides by "empty" methods without emitting a
```

```
369: (8)                    warning.
370: (4)                **kwargs
371: (8)                    Additional parameters passed to `warn_deprecated` to generate the
372: (8)                    deprecation warning; must at least include the "since" key.
373: (4)                """
374: (4)                def empty(): pass
375: (4)                def empty_with_docstring(): """doc"""
376: (4)                name = method.__name__
377: (4)                bound_child = getattr(obj, name)
378: (4)                bound_base = (
379: (8)                    method  # If obj is a class, then we need to use unbound methods.
380: (8)                    if isinstance(bound_child, type(empty)) and isinstance(obj, type)
381: (8)                    else method.__get__(obj))
382: (4)                if (bound_child != bound_base
383: (12)                       and (not allow_empty
384: (17)                            or (getattr(getattr(bound_child, "__code__", None),
385: (29)                                        "co_code", None)
386: (21)                                not in [empty.__code__.co_code,
387: (29)                                        empty_with_docstring.__code__.co_code]))):
388: (8)                    warn_deprecated(**{"name": name, "obj_type": "method", **kwargs})
389: (8)                    return bound_child
390: (4)                return None
391: (0)            @contextlib.contextmanager
392: (0)            def suppress_matplotlib_deprecation_warning():
393: (4)                with warnings.catch_warnings():
394: (8)                    warnings.simplefilter("ignore", MatplotlibDeprecationWarning)
395: (8)                    yield
```

File 74 - __init__.py:

```
1: (0)              from . import _base
2: (0)              from ._axes import Axes  # noqa: F401
3: (0)              Subplot = Axes
4: (0)              class _SubplotBaseMeta(type):
5: (4)                  def __instancecheck__(self, obj):
6: (8)                      return (isinstance(obj, _base._AxesBase)
7: (16)                             and obj.get_subplotspec() is not None)
8: (0)              class SubplotBase(metaclass=_SubplotBaseMeta):
9: (4)                  pass
10: (0)             def subplot_class_factory(cls): return cls
```

File 75 - _axes.py:

```
1: (0)              import functools
2: (0)              import itertools
3: (0)              import logging
4: (0)              import math
5: (0)              from numbers import Integral, Number, Real
6: (0)              import re
7: (0)              import numpy as np
8: (0)              from numpy import ma
9: (0)              import matplotlib as mpl
10: (0)             import matplotlib.category  # Register category unit converter as side effect.
11: (0)             import matplotlib.cbook as cbook
12: (0)             import matplotlib.collections as mcoll
13: (0)             import matplotlib.colors as mcolors
14: (0)             import matplotlib.contour as mcontour
15: (0)             import matplotlib.dates  # noqa # Register date unit converter as side effect.
16: (0)             import matplotlib.image as mimage
17: (0)             import matplotlib.legend as mlegend
18: (0)             import matplotlib.lines as mlines
19: (0)             import matplotlib.markers as mmarkers
20: (0)             import matplotlib.mlab as mlab
21: (0)             import matplotlib.patches as mpatches
22: (0)             import matplotlib.path as mpath
```

```
23: (0)             import matplotlib.quiver as mquiver
24: (0)             import matplotlib.stackplot as mstack
25: (0)             import matplotlib.streamplot as mstream
26: (0)             import matplotlib.table as mtable
27: (0)             import matplotlib.text as mtext
28: (0)             import matplotlib.ticker as mticker
29: (0)             import matplotlib.transforms as mtransforms
30: (0)             import matplotlib.tri as mtri
31: (0)             import matplotlib.units as munits
32: (0)             from matplotlib import _api, _docstring, _preprocess_data
33: (0)             from matplotlib.axes._base import (
34: (4)                 _AxesBase, _TransformedBoundsLocator, _process_plot_format)
35: (0)             from matplotlib.axes._secondary_axes import SecondaryAxis
36: (0)             from matplotlib.container import BarContainer, ErrorbarContainer,
StemContainer
37: (0)             _log = logging.getLogger(__name__)
38: (0)             def _make_axes_method(func):
39: (4)                 """
40: (4)                 Patch the qualname for functions that are directly added to Axes.
41: (4)                 Some Axes functionality is defined in functions in other submodules.
42: (4)                 These are simply added as attributes to Axes. As a result, their
43: (4)                 ``__qualname__`` is e.g. only "table" and not "Axes.table". This
44: (4)                 function fixes that.
45: (4)                 Note that the function itself is patched, so that
46: (4)                 ``matplotlib.table.table.__qualname__` will also show "Axes.table".
47: (4)                 However, since these functions are not intended to be standalone,
48: (4)                 this is bearable.
49: (4)                 """
50: (4)                 func.__qualname__ = f"Axes.{func.__name__}"
51: (4)                 return func
52: (0)             @_docstring.interpd
53: (0)             class Axes(_AxesBase):
54: (4)                 """
55: (4)                 An Axes object encapsulates all the elements of an individual (sub-)plot
in
56: (4)                 a figure.
57: (4)                 It contains most of the (sub-)plot elements: `~.axis.Axis`,
58: (4)                 `~.axis.Tick`, `~.lines.Line2D`, `~.text.Text`, `~.patches.Polygon`, etc.,
59: (4)                 and sets the coordinate system.
60: (4)                 Like all visible elements in a figure, Axes is an `.Artist` subclass.
61: (4)                 The `Axes` instance supports callbacks through a callbacks attribute which
62: (4)                 is a `~.cbook.CallbackRegistry` instance.  The events you can connect to
63: (4)                 are 'xlim_changed' and 'ylim_changed' and the callback will be called with
64: (4)                 func(*ax*) where *ax* is the `Axes` instance.
65: (4)                 .. note::
66: (8)                     As a user, you do not instantiate Axes directly, but use Axes creation
67: (8)                     methods instead; e.g. from `.pyplot` or `.Figure`:
68: (8)                     `~.pyplot.subplots`, `~.pyplot.subplot_mosaic` or `.Figure.add_axes`.
69: (4)                 Attributes
70: (4)                 ----------
71: (4)                 dataLim : `.Bbox`
72: (8)                     The bounding box enclosing all data displayed in the Axes.
73: (4)                 viewLim : `.Bbox`
74: (8)                     The view limits in data coordinates.
75: (4)                 """
76: (4)                 def get_title(self, loc="center"):
77: (8)                     """
78: (8)                     Get an Axes title.
79: (8)                     Get one of the three available Axes titles. The available titles
80: (8)                     are positioned above the Axes in the center, flush with the left
81: (8)                     edge, and flush with the right edge.
82: (8)                     Parameters
83: (8)                     ----------
84: (8)                     loc : {'center', 'left', 'right'}, str, default: 'center'
85: (12)                        Which title to return.
86: (8)                     Returns
87: (8)                     -------
88: (8)                     str
89: (12)                        The title text string.
```

```
 90: (8)                        """
 91: (8)                        titles = {'left': self._left_title,
 92: (18)                                  'center': self.title,
 93: (18)                                  'right': self._right_title}
 94: (8)                        title = _api.check_getitem(titles, loc=loc.lower())
 95: (8)                        return title.get_text()
 96: (4)                    def set_title(self, label, fontdict=None, loc=None, pad=None, *, y=None,
 97: (18)                                  **kwargs):
 98: (8)                        """
 99: (8)                        Set a title for the Axes.
 100: (8)                       Set one of the three available Axes titles. The available titles
 101: (8)                       are positioned above the Axes in the center, flush with the left
 102: (8)                       edge, and flush with the right edge.
 103: (8)                       Parameters
 104: (8)                       ----------
 105: (8)                       label : str
 106: (12)                          Text to use for the title
 107: (8)                       fontdict : dict
 108: (12)                          .. admonition:: Discouraged
 109: (15)                             The use of *fontdict* is discouraged. Parameters should be
 passed as
 110: (15)                             individual keyword arguments or using dictionary-unpacking
 111: (15)                             ``set_title(..., **fontdict)``.
 112: (12)                          A dictionary controlling the appearance of the title text,
 113: (12)                          the default *fontdict* is::
 114: (15)                             {'fontsize': rcParams['axes.titlesize'],
 115: (16)                              'fontweight': rcParams['axes.titleweight'],
 116: (16)                              'color': rcParams['axes.titlecolor'],
 117: (16)                              'verticalalignment': 'baseline',
 118: (16)                              'horizontalalignment': loc}
 119: (8)                       loc : {'center', 'left', 'right'}, default: :rc:`axes.titlelocation`
 120: (12)                          Which title to set.
 121: (8)                       y : float, default: :rc:`axes.titley`
 122: (12)                          Vertical Axes location for the title (1.0 is the top).  If
 123: (12)                          None (the default) and :rc:`axes.titley` is also None, y is
 124: (12)                          determined automatically to avoid decorators on the Axes.
 125: (8)                       pad : float, default: :rc:`axes.titlepad`
 126: (12)                          The offset of the title from the top of the Axes, in points.
 127: (8)                       Returns
 128: (8)                       -------
 129: (8)                       `.Text`
 130: (12)                          The matplotlib text instance representing the title
 131: (8)                       Other Parameters
 132: (8)                       ----------------
 133: (8)                       **kwargs : `~matplotlib.text.Text` properties
 134: (12)                          Other keyword arguments are text properties, see `.Text` for a
 list
 135: (12)                          of valid text properties.
 136: (8)                       """
 137: (8)                       if loc is None:
 138: (12)                          loc = mpl.rcParams['axes.titlelocation']
 139: (8)                       if y is None:
 140: (12)                          y = mpl.rcParams['axes.titley']
 141: (8)                       if y is None:
 142: (12)                          y = 1.0
 143: (8)                       else:
 144: (12)                          self._autotitlepos = False
 145: (8)                       kwargs['y'] = y
 146: (8)                       titles = {'left': self._left_title,
 147: (18)                                 'center': self.title,
 148: (18)                                 'right': self._right_title}
 149: (8)                       title = _api.check_getitem(titles, loc=loc.lower())
 150: (8)                       default = {
 151: (12)                          'fontsize': mpl.rcParams['axes.titlesize'],
 152: (12)                          'fontweight': mpl.rcParams['axes.titleweight'],
 153: (12)                          'verticalalignment': 'baseline',
 154: (12)                          'horizontalalignment': loc.lower()}
 155: (8)                       titlecolor = mpl.rcParams['axes.titlecolor']
 156: (8)                       if not cbook._str_lower_equal(titlecolor, 'auto'):
```

```
157: (12)                        default["color"] = titlecolor
158: (8)                    if pad is None:
159: (12)                        pad = mpl.rcParams['axes.titlepad']
160: (8)                    self._set_title_offset_trans(float(pad))
161: (8)                    title.set_text(label)
162: (8)                    title.update(default)
163: (8)                    if fontdict is not None:
164: (12)                        title.update(fontdict)
165: (8)                    title._internal_update(kwargs)
166: (8)                    return title
167: (4)                def get_legend_handles_labels(self, legend_handler_map=None):
168: (8)                    """
169: (8)                    Return handles and labels for legend
170: (8)                    ``ax.legend()`` is equivalent to ::
171: (10)                     h, l = ax.get_legend_handles_labels()
172: (10)                     ax.legend(h, l)
173: (8)                    """
174: (8)                    handles, labels = mlegend._get_legend_handles_labels(
175: (12)                        [self], legend_handler_map)
176: (8)                    return handles, labels
177: (4)                @_docstring.dedent_interpd
178: (4)                def legend(self, *args, **kwargs):
179: (8)                    """
180: (8)                    Place a legend on the Axes.
181: (8)                    Call signatures::
182: (12)                        legend()
183: (12)                        legend(handles, labels)
184: (12)                        legend(handles=handles)
185: (12)                        legend(labels)
186: (8)                    The call signatures correspond to the following different ways to use
187: (8)                    this method:
188: (8)                    **1. Automatic detection of elements to be shown in the legend**
189: (8)                    The elements to be added to the legend are automatically determined,
190: (8)                    when you do not pass in any extra arguments.
191: (8)                    In this case, the labels are taken from the artist. You can specify
192: (8)                    them either at artist creation or by calling the
193: (8)                    :meth:`~.Artist.set_label` method on the artist::
194: (12)                        ax.plot([1, 2, 3], label='Inline label')
195: (12)                        ax.legend()
196: (8)                    or::
197: (12)                        line, = ax.plot([1, 2, 3])
198: (12)                        line.set_label('Label via method')
199: (12)                        ax.legend()
200: (8)                    .. note::
201: (12)                        Specific artists can be excluded from the automatic legend element
202: (12)                        selection by using a label starting with an underscore, "_".
203: (12)                        A string starting with an underscore is the default label for all
204: (12)                        artists, so calling `.Axes.legend` without any arguments and
205: (12)                        without setting the labels manually will result in a
``UserWarning``
206: (12)                        and an empty legend being drawn.
207: (8)                    **2. Explicitly listing the artists and labels in the legend**
208: (8)                    For full control of which artists have a legend entry, it is possible
209: (8)                    to pass an iterable of legend artists followed by an iterable of
210: (8)                    legend labels respectively::
211: (12)                        ax.legend([line1, line2, line3], ['label1', 'label2', 'label3'])
212: (8)                    **3. Explicitly listing the artists in the legend**
213: (8)                    This is similar to 2, but the labels are taken from the artists'
214: (8)                    label properties. Example::
215: (12)                        line1, = ax.plot([1, 2, 3], label='label1')
216: (12)                        line2, = ax.plot([1, 2, 3], label='label2')
217: (12)                        ax.legend(handles=[line1, line2])
218: (8)                    **4. Labeling existing plot elements**
219: (8)                    .. admonition:: Discouraged
220: (12)                        This call signature is discouraged, because the relation between
221: (12)                        plot elements and labels is only implicit by their order and can
222: (12)                        easily be mixed up.
223: (8)                    To make a legend for all artists on an Axes, call this function with
224: (8)                    an iterable of strings, one for each legend item. For example::
```

```
(12)                        ax.plot([1, 2, 3])
(12)                        ax.plot([5, 6, 7])
(12)                        ax.legend(['First line', 'Second line'])
(8)                     Parameters
(8)                     ----------
(8)                     handles : list of (`.Artist` or tuple of `.Artist`), optional
(12)                        A list of Artists (lines, patches) to be added to the legend.
(12)                        Use this together with *labels*, if you need full control on what
(12)                        is shown in the legend and the automatic mechanism described above
(12)                        is not sufficient.
(12)                        The length of handles and labels should be the same in this
(12)                        case. If they are not, they are truncated to the smaller length.
(12)                        If an entry contains a tuple, then the legend handler for all
Artists in the
(12)                        tuple will be placed alongside a single label.
(8)                     labels : list of str, optional
(12)                        A list of labels to show next to the artists.
(12)                        Use this together with *handles*, if you need full control on what
(12)                        is shown in the legend and the automatic mechanism described above
(12)                        is not sufficient.
(8)                     Returns
(8)                     -------
(8)                     `~matplotlib.legend.Legend`
(8)                     Other Parameters
(8)                     ----------------
(8)                     %(_legend_kw_axes)s
(8)                     See Also
(8)                     --------
(8)                     .Figure.legend
(8)                     Notes
(8)                     -----
(8)                     Some artists are not supported by this function.  See
(8)                     :ref:`legend_guide` for details.
(8)                     Examples
(8)                     --------
(8)                     .. plot:: gallery/text_labels_and_annotations/legend.py
(8)                     """
(8)                     handles, labels, kwargs = mlegend._parse_legend_args([self], *args,
**kwargs)
(8)                     self.legend_ = mlegend.Legend(self, handles, labels, **kwargs)
(8)                     self.legend_._remove_method = self._remove_legend
(8)                     return self.legend_
(4)                 def _remove_legend(self, legend):
(8)                     self.legend_ = None
(4)                 def inset_axes(self, bounds, *, transform=None, zorder=5, **kwargs):
(8)                     """
(8)                     Add a child inset Axes to this existing Axes.
(8)                     Parameters
(8)                     ----------
(8)                     bounds : [x0, y0, width, height]
(12)                        Lower-left corner of inset Axes, and its width and height.
(8)                     transform : `.Transform`
(12)                        Defaults to `ax.transAxes`, i.e. the units of *rect* are in
(12)                        Axes-relative coordinates.
(8)                     projection : {None, 'aitoff', 'hammer', 'lambert', 'mollweide', \
(0)             'polar', 'rectilinear', str}, optional
(12)                        The projection type of the inset `~.axes.Axes`. *str* is the name
(12)                        of a custom projection, see `~matplotlib.projections`. The default
(12)                        None results in a 'rectilinear' projection.
(8)                     polar : bool, default: False
(12)                        If True, equivalent to projection='polar'.
(8)                     axes_class : subclass type of `~.axes.Axes`, optional
(12)                        The `.axes.Axes` subclass that is instantiated.  This parameter
(12)                        is incompatible with *projection* and *polar*.  See
(12)                        :ref:`axisartist_users-guide-index` for examples.
(8)                     zorder : number
(12)                        Defaults to 5 (same as `.Axes.legend`).  Adjust higher or lower
(12)                        to change whether it is above or below data plotted on the
(12)                        parent Axes.
```

```
292: (8)                    **kwargs
293: (12)                       Other keyword arguments are passed on to the inset Axes class.
294: (8)                    Returns
295: (8)                    -------
296: (8)                    ax
297: (12)                       The created `~.axes.Axes` instance.
298: (8)                    Examples
299: (8)                    --------
300: (8)                    This example makes two inset Axes, the first is in Axes-relative
301: (8)                    coordinates, and the second in data-coordinates::
302: (12)                       fig, ax = plt.subplots()
303: (12)                       ax.plot(range(10))
304: (12)                       axin1 = ax.inset_axes([0.8, 0.1, 0.15, 0.15])
305: (12)                       axin2 = ax.inset_axes(
306: (20)                               [5, 7, 2.3, 2.3], transform=ax.transData)
307: (8)                    """
308: (8)                    if transform is None:
309: (12)                       transform = self.transAxes
310: (8)                    kwargs.setdefault('label', 'inset_axes')
311: (8)                    inset_locator = _TransformedBoundsLocator(bounds, transform)
312: (8)                    bounds = inset_locator(self, None).bounds
313: (8)                    projection_class, pkw =
self.figure._process_projection_requirements(**kwargs)
314: (8)                    inset_ax = projection_class(self.figure, bounds, zorder=zorder, **pkw)
315: (8)                    inset_ax.set_axes_locator(inset_locator)
316: (8)                    self.add_child_axes(inset_ax)
317: (8)                    return inset_ax
318: (4)                @_docstring.dedent_interpd
319: (4)                def indicate_inset(self, bounds, inset_ax=None, *, transform=None,
320: (23)                                  facecolor='none', edgecolor='0.5', alpha=0.5,
321: (23)                                  zorder=4.99, **kwargs):
322: (8)                    """
323: (8)                    Add an inset indicator to the Axes.  This is a rectangle on the plot
324: (8)                    at the position indicated by *bounds* that optionally has lines that
325: (8)                    connect the rectangle to an inset Axes (`.Axes.inset_axes`).
326: (8)                    Warnings
327: (8)                    --------
328: (8)                    This method is experimental as of 3.0, and the API may change.
329: (8)                    Parameters
330: (8)                    ----------
331: (8)                    bounds : [x0, y0, width, height]
332: (12)                       Lower-left corner of rectangle to be marked, and its width
333: (12)                       and height.
334: (8)                    inset_ax : `.Axes`
335: (12)                       An optional inset Axes to draw connecting lines to.  Two lines are
336: (12)                       drawn connecting the indicator box to the inset Axes on corners
337: (12)                       chosen so as to not overlap with the indicator box.
338: (8)                    transform : `.Transform`
339: (12)                       Transform for the rectangle coordinates. Defaults to
340: (12)                       `ax.transAxes`, i.e. the units of *rect* are in Axes-relative
341: (12)                       coordinates.
342: (8)                    facecolor : :mpltype:`color`, default: 'none'
343: (12)                       Facecolor of the rectangle.
344: (8)                    edgecolor : :mpltype:`color`, default: '0.5'
345: (12)                       Color of the rectangle and color of the connecting lines.
346: (8)                    alpha : float, default: 0.5
347: (12)                       Transparency of the rectangle and connector lines.
348: (8)                    zorder : float, default: 4.99
349: (12)                       Drawing order of the rectangle and connector lines.  The default,
350: (12)                       4.99, is just below the default level of inset Axes.
351: (8)                    **kwargs
352: (12)                       Other keyword arguments are passed on to the `.Rectangle` patch:
353: (12)                       %(Rectangle:kwdoc)s
354: (8)                    Returns
355: (8)                    -------
356: (8)                    rectangle_patch : `.patches.Rectangle`
357: (13)                        The indicator frame.
358: (8)                    connector_lines : 4-tuple of `.patches.ConnectionPatch`
359: (12)                       The four connector lines connecting to (lower_left, upper_left,
```

```
360: (12)                        lower_right upper_right) corners of *inset_ax*. Two lines are
361: (12)                        set with visibility to *False*,  but the user can set the
362: (12)                        visibility to True if the automatic choice is not deemed correct.
363: (8)                    """
364: (8)                    self.apply_aspect()
365: (8)                    if transform is None:
366: (12)                        transform = self.transData
367: (8)                    kwargs.setdefault('label', '_indicate_inset')
368: (8)                    x, y, width, height = bounds
369: (8)                    rectangle_patch = mpatches.Rectangle(
370: (12)                        (x, y), width, height,
371: (12)                        facecolor=facecolor, edgecolor=edgecolor, alpha=alpha,
372: (12)                        zorder=zorder, transform=transform, **kwargs)
373: (8)                    self.add_patch(rectangle_patch)
374: (8)                    connects = []
375: (8)                    if inset_ax is not None:
376: (12)                        for xy_inset_ax in [(0, 0), (0, 1), (1, 0), (1, 1)]:
377: (16)                            ex, ey = xy_inset_ax
378: (16)                            if self.xaxis.get_inverted():
379: (20)                                ex = 1 - ex
380: (16)                            if self.yaxis.get_inverted():
381: (20)                                ey = 1 - ey
382: (16)                            xy_data = x + ex * width, y + ey * height
383: (16)                            p = mpatches.ConnectionPatch(
384: (20)                                xyA=xy_inset_ax, coordsA=inset_ax.transAxes,
385: (20)                                xyB=xy_data, coordsB=self.transData,
386: (20)                                arrowstyle="-", zorder=zorder,
387: (20)                                edgecolor=edgecolor, alpha=alpha)
388: (16)                            connects.append(p)
389: (16)                            self.add_patch(p)
390: (12)                        pos = inset_ax.get_position()
391: (12)                        bboxins = pos.transformed(self.figure.transSubfigure)
392: (12)                        rectbbox = mtransforms.Bbox.from_bounds(
393: (16)                            *bounds
394: (12)                        ).transformed(transform)
395: (12)                        x0 = rectbbox.x0 < bboxins.x0
396: (12)                        x1 = rectbbox.x1 < bboxins.x1
397: (12)                        y0 = rectbbox.y0 < bboxins.y0
398: (12)                        y1 = rectbbox.y1 < bboxins.y1
399: (12)                        connects[0].set_visible(x0 ^ y0)
400: (12)                        connects[1].set_visible(x0 == y1)
401: (12)                        connects[2].set_visible(x1 == y0)
402: (12)                        connects[3].set_visible(x1 ^ y1)
403: (8)                    return rectangle_patch, tuple(connects) if connects else None
404: (4)                def indicate_inset_zoom(self, inset_ax, **kwargs):
405: (8)                    """
406: (8)                    Add an inset indicator rectangle to the Axes based on the axis
407: (8)                    limits for an *inset_ax* and draw connectors between *inset_ax*
408: (8)                    and the rectangle.
409: (8)                    Warnings
410: (8)                    --------
411: (8)                    This method is experimental as of 3.0, and the API may change.
412: (8)                    Parameters
413: (8)                    ----------
414: (8)                    inset_ax : `.Axes`
415: (12)                        Inset Axes to draw connecting lines to.  Two lines are
416: (12)                        drawn connecting the indicator box to the inset Axes on corners
417: (12)                        chosen so as to not overlap with the indicator box.
418: (8)                    **kwargs
419: (12)                        Other keyword arguments are passed on to `.Axes.indicate_inset`
420: (8)                    Returns
421: (8)                    -------
422: (8)                    rectangle_patch : `.patches.Rectangle`
423: (13)                         Rectangle artist.
424: (8)                    connector_lines : 4-tuple of `.patches.ConnectionPatch`
425: (12)                        Each of four connector lines coming from the rectangle drawn on
426: (12)                        this axis, in the order lower left, upper left, lower right,
427: (12)                        upper right.
428: (12)                        Two are set with visibility to *False*,  but the user can
```

```
429: (12)                        set the visibility to *True* if the automatic choice is not deemed
430: (12)                        correct.
431: (8)                    """
432: (8)                    xlim = inset_ax.get_xlim()
433: (8)                    ylim = inset_ax.get_ylim()
434: (8)                    rect = (xlim[0], ylim[0], xlim[1] - xlim[0], ylim[1] - ylim[0])
435: (8)                    return self.indicate_inset(rect, inset_ax, **kwargs)
436: (4)                @_docstring.dedent_interpd
437: (4)                def secondary_xaxis(self, location, *, functions=None, transform=None,
**kwargs):
438: (8)                    """
439: (8)                    Add a second x-axis to this `~.axes.Axes`.
440: (8)                    For example if we want to have a second scale for the data plotted on
441: (8)                    the xaxis.
442: (8)                    %(_secax_docstring)s
443: (8)                    Examples
444: (8)                    --------
445: (8)                    The main axis shows frequency, and the secondary axis shows period.
446: (8)                    .. plot::
447: (12)                       fig, ax = plt.subplots()
448: (12)                       ax.loglog(range(1, 360, 5), range(1, 360, 5))
449: (12)                       ax.set_xlabel('frequency [Hz]')
450: (12)                       def invert(x):
451: (16)                           x = np.array(x).astype(float)
452: (16)                           near_zero = np.isclose(x, 0)
453: (16)                           x[near_zero] = np.inf
454: (16)                           x[~near_zero] = 1 / x[~near_zero]
455: (16)                           return x
456: (12)                       secax = ax.secondary_xaxis('top', functions=(invert, invert))
457: (12)                       secax.set_xlabel('Period [s]')
458: (12)                       plt.show()
459: (8)                    To add a secondary axis relative to your data, you can pass a
transform
460: (8)                    to the new axis.
461: (8)                    .. plot::
462: (12)                       fig, ax = plt.subplots()
463: (12)                       ax.plot(range(0, 5), range(-1, 4))
464: (12)                       secax = ax.secondary_xaxis(0, transform=ax.transData)
465: (8)                    """
466: (8)                    if not (location in ['top', 'bottom'] or isinstance(location, Real)):
467: (12)                       raise ValueError('secondary_xaxis location must be either '
468: (29)                                        'a float or "top"/"bottom"')
469: (8)                    secondary_ax = SecondaryAxis(self, 'x', location, functions,
470: (37)                                                transform, **kwargs)
471: (8)                    self.add_child_axes(secondary_ax)
472: (8)                    return secondary_ax
473: (4)                @_docstring.dedent_interpd
474: (4)                def secondary_yaxis(self, location, *, functions=None, transform=None,
**kwargs):
475: (8)                    """
476: (8)                    Add a second y-axis to this `~.axes.Axes`.
477: (8)                    For example if we want to have a second scale for the data plotted on
478: (8)                    the yaxis.
479: (8)                    %(_secax_docstring)s
480: (8)                    Examples
481: (8)                    --------
482: (8)                    Add a secondary Axes that converts from radians to degrees
483: (8)                    .. plot::
484: (12)                       fig, ax = plt.subplots()
485: (12)                       ax.plot(range(1, 360, 5), range(1, 360, 5))
486: (12)                       ax.set_ylabel('degrees')
487: (12)                       secax = ax.secondary_yaxis('right', functions=(np.deg2rad,
488: (59)                                                                      np.rad2deg))
489: (12)                       secax.set_ylabel('radians')
490: (8)                    To add a secondary axis relative to your data, you can pass a
transform
491: (8)                    to the new axis.
492: (8)                    .. plot::
493: (12)                       fig, ax = plt.subplots()
```

```
494: (12)                        ax.plot(range(0, 5), range(-1, 4))
495: (12)                        secax = ax.secondary_yaxis(3, transform=ax.transData)
496: (8)                    """
497: (8)                    if not (location in ['left', 'right'] or isinstance(location, Real)):
498: (12)                        raise ValueError('secondary_yaxis location must be either '
499: (29)                                         'a float or "left"/"right"')
500: (8)                    secondary_ax = SecondaryAxis(self, 'y', location, functions,
501: (37)                                                 transform, **kwargs)
502: (8)                    self.add_child_axes(secondary_ax)
503: (8)                    return secondary_ax
504: (4)                @_docstring.dedent_interpd
505: (4)                def text(self, x, y, s, fontdict=None, **kwargs):
506: (8)                    """
507: (8)                    Add text to the Axes.
508: (8)                    Add the text *s* to the Axes at location *x*, *y* in data coordinates,
509: (8)                    with a default ``horizontalalignment`` on the ``left`` and
510: (8)                    ``verticalalignment`` at the ``baseline``. See
511: (8)                    :doc:`/gallery/text_labels_and_annotations/text_alignment`.
512: (8)                    Parameters
513: (8)                    ----------
514: (8)                    x, y : float
515: (12)                       The position to place the text. By default, this is in data
516: (12)                       coordinates. The coordinate system can be changed using the
517: (12)                       *transform* parameter.
518: (8)                    s : str
519: (12)                       The text.
520: (8)                    fontdict : dict, default: None
521: (12)                       .. admonition:: Discouraged
522: (15)                          The use of *fontdict* is discouraged. Parameters should be
passed as
523: (15)                          individual keyword arguments or using dictionary-unpacking
524: (15)                          ``text(..., **fontdict)``.
525: (12)                       A dictionary to override the default text properties. If fontdict
526: (12)                       is None, the defaults are determined by `.rcParams`.
527: (8)                    Returns
528: (8)                    -------
529: (8)                    `.Text`
530: (12)                       The created `.Text` instance.
531: (8)                    Other Parameters
532: (8)                    ----------------
533: (8)                    **kwargs : `~matplotlib.text.Text` properties.
534: (12)                       Other miscellaneous text parameters.
535: (12)                       %(Text:kwdoc)s
536: (8)                    Examples
537: (8)                    --------
538: (8)                    Individual keyword arguments can be used to override any given
539: (8)                    parameter::
540: (12)                       >>> text(x, y, s, fontsize=12)
541: (8)                    The default transform specifies that text is in data coords,
542: (8)                    alternatively, you can specify text in axis coords ((0, 0) is
543: (8)                    lower-left and (1, 1) is upper-right).  The example below places
544: (8)                    text in the center of the Axes::
545: (12)                       >>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
546: (12)                       ...      verticalalignment='center', transform=ax.transAxes)
547: (8)                    You can put a rectangular box around the text instance (e.g., to
548: (8)                    set a background color) by using the keyword *bbox*.  *bbox* is
549: (8)                    a dictionary of `~matplotlib.patches.Rectangle`
550: (8)                    properties.  For example::
551: (12)                       >>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
552: (8)                    """
553: (8)                    effective_kwargs = {
554: (12)                       'verticalalignment': 'baseline',
555: (12)                       'horizontalalignment': 'left',
556: (12)                       'transform': self.transData,
557: (12)                       'clip_on': False,
558: (12)                       **(fontdict if fontdict is not None else {}),
559: (12)                       **kwargs,
560: (8)                    }
561: (8)                    t = mtext.Text(x, y, text=s, **effective_kwargs)
```

```
(8)                     if t.get_clip_path() is None:
(12)                        t.set_clip_path(self.patch)
(8)                     self._add_text(t)
(8)                     return t
(4)                 @_docstring.dedent_interpd
(4)                 def annotate(self, text, xy, xytext=None, xycoords='data',
textcoords=None,
(17)                         arrowprops=None, annotation_clip=None, **kwargs):
(8)                     a = mtext.Annotation(text, xy, xytext=xytext, xycoords=xycoords,
(29)                                         textcoords=textcoords, arrowprops=arrowprops,
(29)                                         annotation_clip=annotation_clip, **kwargs)
(8)                     a.set_transform(mtransforms.IdentityTransform())
(8)                     if kwargs.get('clip_on', False) and a.get_clip_path() is None:
(12)                        a.set_clip_path(self.patch)
(8)                     self._add_text(a)
(8)                     return a
(4)                 annotate.__doc__ = mtext.Annotation.__init__.__doc__
(4)                 @_docstring.dedent_interpd
(4)                 def axhline(self, y=0, xmin=0, xmax=1, **kwargs):
(8)                     """
(8)                     Add a horizontal line across the Axes.
(8)                     Parameters
(8)                     ----------
(8)                     y : float, default: 0
(12)                        y position in data coordinates of the horizontal line.
(8)                     xmin : float, default: 0
(12)                        Should be between 0 and 1, 0 being the far left of the plot, 1 the
(12)                        far right of the plot.
(8)                     xmax : float, default: 1
(12)                        Should be between 0 and 1, 0 being the far left of the plot, 1 the
(12)                        far right of the plot.
(8)                     Returns
(8)                     -------
(8)                     `~matplotlib.lines.Line2D`
(8)                     Other Parameters
(8)                     ----------------
(8)                     **kwargs
(12)                        Valid keyword arguments are `.Line2D` properties, except for
(12)                        'transform':
(12)                        %(Line2D:kwdoc)s
(8)                     See Also
(8)                     --------
(8)                     hlines : Add horizontal lines in data coordinates.
(8)                     axhspan : Add a horizontal span (rectangle) across the axis.
(8)                     axline : Add a line with an arbitrary slope.
(8)                     Examples
(8)                     --------
(8)                     * draw a thick red hline at 'y' = 0 that spans the xrange::
(12)                        >>> axhline(linewidth=4, color='r')
(8)                     * draw a default hline at 'y' = 1 that spans the xrange::
(12)                        >>> axhline(y=1)
(8)                     * draw a default hline at 'y' = .5 that spans the middle half of
(10)                      the xrange::
(12)                        >>> axhline(y=.5, xmin=0.25, xmax=0.75)
(8)                     """
(8)                     self._check_no_units([xmin, xmax], ['xmin', 'xmax'])
(8)                     if "transform" in kwargs:
(12)                        raise ValueError("'transform' is not allowed as a keyword "
(29)                                         "argument; axhline generates its own transform.")
(8)                     ymin, ymax = self.get_ybound()
(8)                     yy, = self._process_unit_info([("y", y)], kwargs)
(8)                     scaley = (yy < ymin) or (yy > ymax)
(8)                     trans = self.get_yaxis_transform(which='grid')
(8)                     l = mlines.Line2D([xmin, xmax], [y, y], transform=trans, **kwargs)
(8)                     self.add_line(l)
(8)                     l.get_path()._interpolation_steps =
mpl.axis.GRIDLINE_INTERPOLATION_STEPS
(8)                     if scaley:
(12)                        self._request_autoscale_view("y")
```

```
(8)                     return l
(4)                 @_docstring.dedent_interpd
(4)                 def axvline(self, x=0, ymin=0, ymax=1, **kwargs):
(8)                     """
(8)                     Add a vertical line across the Axes.
(8)                     Parameters
(8)                     ----------
(8)                     x : float, default: 0
(12)                        x position in data coordinates of the vertical line.
(8)                     ymin : float, default: 0
(12)                        Should be between 0 and 1, 0 being the bottom of the plot, 1 the
(12)                        top of the plot.
(8)                     ymax : float, default: 1
(12)                        Should be between 0 and 1, 0 being the bottom of the plot, 1 the
(12)                        top of the plot.
(8)                     Returns
(8)                     -------
(8)                     `~matplotlib.lines.Line2D`
(8)                     Other Parameters
(8)                     ----------------
(8)                     **kwargs
(12)                        Valid keyword arguments are `.Line2D` properties, except for
(12)                        'transform':
(12)                        %(Line2D:kwdoc)s
(8)                     See Also
(8)                     --------
(8)                     vlines : Add vertical lines in data coordinates.
(8)                     axvspan : Add a vertical span (rectangle) across the axis.
(8)                     axline : Add a line with an arbitrary slope.
(8)                     Examples
(8)                     --------
(8)                     * draw a thick red vline at *x* = 0 that spans the yrange::
(12)                        >>> axvline(linewidth=4, color='r')
(8)                     * draw a default vline at *x* = 1 that spans the yrange::
(12)                        >>> axvline(x=1)
(8)                     * draw a default vline at *x* = .5 that spans the middle half of
(10)                      the yrange::
(12)                        >>> axvline(x=.5, ymin=0.25, ymax=0.75)
(8)                     """
(8)                     self._check_no_units([ymin, ymax], ['ymin', 'ymax'])
(8)                     if "transform" in kwargs:
(12)                        raise ValueError("'transform' is not allowed as a keyword "
(29)                                         "argument; axvline generates its own transform.")
(8)                     xmin, xmax = self.get_xbound()
(8)                     xx, = self._process_unit_info([("x", x)], kwargs)
(8)                     scalex = (xx < xmin) or (xx > xmax)
(8)                     trans = self.get_xaxis_transform(which='grid')
(8)                     l = mlines.Line2D([x, x], [ymin, ymax], transform=trans, **kwargs)
(8)                     self.add_line(l)
(8)                     l.get_path()._interpolation_steps =
mpl.axis.GRIDLINE_INTERPOLATION_STEPS
(8)                     if scalex:
(12)                        self._request_autoscale_view("x")
(8)                     return l
(4)                 @staticmethod
(4)                 def _check_no_units(vals, names):
(8)                     for val, name in zip(vals, names):
(12)                        if not munits._is_natively_supported(val):
(16)                            raise ValueError(f"{name} must be a single scalar value, "
(33)                                             f"but got {val}")
(4)                 @_docstring.dedent_interpd
(4)                 def axline(self, xy1, xy2=None, *, slope=None, **kwargs):
(8)                     """
(8)                     Add an infinitely long straight line.
(8)                     The line can be defined either by two points *xy1* and *xy2*, or
(8)                     by one point *xy1* and a *slope*.
(8)                     This draws a straight line "on the screen", regardless of the x and y
(8)                     scales, and is thus also suitable for drawing exponential decays in
(8)                     semilog plots, power laws in loglog plots, etc. However, *slope*
```

```
(8)                     should only be used with linear scales; It has no clear meaning for
(8)                     all other scales, and thus the behavior is undefined. Please specify
(8)                     the line using the points *xy1*, *xy2* for non-linear scales.
(8)                     The *transform* keyword argument only applies to the points *xy1*,
(8)                     *xy2*. The *slope* (if given) is always in data coordinates. This can
(8)                     be used e.g. with ``ax.transAxes`` for drawing grid lines with a fixed
(8)                     slope.
(8)                     Parameters
(8)                     ----------
(8)                     xy1, xy2 : (float, float)
(12)                        Points for the line to pass through.
(12)                        Either *xy2* or *slope* has to be given.
(8)                     slope : float, optional
(12)                        The slope of the line. Either *xy2* or *slope* has to be given.
(8)                     Returns
(8)                     -------
(8)                     `.AxLine`
(8)                     Other Parameters
(8)                     ----------------
(8)                     **kwargs
(12)                        Valid kwargs are `.Line2D` properties
(12)                        %(Line2D:kwdoc)s
(8)                     See Also
(8)                     --------
(8)                     axhline : for horizontal lines
(8)                     axvline : for vertical lines
(8)                     Examples
(8)                     --------
(8)                     Draw a thick red line passing through (0, 0) and (1, 1)::
(12)                        >>> axline((0, 0), (1, 1), linewidth=4, color='r')
(8)                     """
(8)                     if slope is not None and (self.get_xscale() != 'linear' or
(34)                                              self.get_yscale() != 'linear'):
(12)                        raise TypeError("'slope' cannot be used with non-linear scales")
(8)                     datalim = [xy1] if xy2 is None else [xy1, xy2]
(8)                     if "transform" in kwargs:
(12)                        datalim = []
(8)                     line = mlines.AxLine(xy1, xy2, slope, **kwargs)
(8)                     self._set_artist_props(line)
(8)                     if line.get_clip_path() is None:
(12)                        line.set_clip_path(self.patch)
(8)                     if not line.get_label():
(12)                        line.set_label(f"_child{len(self._children)}")
(8)                     self._children.append(line)
(8)                     line._remove_method = self._children.remove
(8)                     self.update_datalim(datalim)
(8)                     self._request_autoscale_view()
(8)                     return line
(4)                 @_docstring.dedent_interpd
(4)                 def axhspan(self, ymin, ymax, xmin=0, xmax=1, **kwargs):
(8)                     """
(8)                     Add a horizontal span (rectangle) across the Axes.
(8)                     The rectangle spans from *ymin* to *ymax* vertically, and, by default,
(8)                     the whole x-axis horizontally.  The x-span can be set using *xmin*
(8)                     (default: 0) and *xmax* (default: 1) which are in axis units; e.g.
(8)                     ``xmin = 0.5`` always refers to the middle of the x-axis regardless of
(8)                     the limits set by `~.Axes.set_xlim`.
(8)                     Parameters
(8)                     ----------
(8)                     ymin : float
(12)                        Lower y-coordinate of the span, in data units.
(8)                     ymax : float
(12)                        Upper y-coordinate of the span, in data units.
(8)                     xmin : float, default: 0
(12)                        Lower x-coordinate of the span, in x-axis (0-1) units.
(8)                     xmax : float, default: 1
(12)                        Upper x-coordinate of the span, in x-axis (0-1) units.
(8)                     Returns
(8)                     -------
```

```
766: (8)                    `~matplotlib.patches.Rectangle`
767: (12)                       Horizontal span (rectangle) from (xmin, ymin) to (xmax, ymax).
768: (8)                    Other Parameters
769: (8)                    ----------------
770: (8)                    **kwargs : `~matplotlib.patches.Rectangle` properties
771: (8)                    %(Rectangle:kwdoc)s
772: (8)                    See Also
773: (8)                    --------
774: (8)                    axvspan : Add a vertical span across the Axes.
775: (8)                    """
776: (8)                    self._check_no_units([xmin, xmax], ['xmin', 'xmax'])
777: (8)                    (ymin, ymax), = self._process_unit_info([("y", [ymin, ymax])], kwargs)
778: (8)                    p = mpatches.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin,
**kwargs)
779: (8)                    p.set_transform(self.get_yaxis_transform(which="grid"))
780: (8)                    ix = self.dataLim.intervalx.copy()
781: (8)                    mx = self.dataLim.minposx
782: (8)                    self.add_patch(p)
783: (8)                    self.dataLim.intervalx = ix
784: (8)                    self.dataLim.minposx = mx
785: (8)                    p.get_path()._interpolation_steps =
mpl.axis.GRIDLINE_INTERPOLATION_STEPS
786: (8)                    self._request_autoscale_view("y")
787: (8)                    return p
788: (4)                @_docstring.dedent_interpd
789: (4)                def axvspan(self, xmin, xmax, ymin=0, ymax=1, **kwargs):
790: (8)                    """
791: (8)                    Add a vertical span (rectangle) across the Axes.
792: (8)                    The rectangle spans from *xmin* to *xmax* horizontally, and, by
793: (8)                    default, the whole y-axis vertically.  The y-span can be set using
794: (8)                    *ymin* (default: 0) and *ymax* (default: 1) which are in axis units;
795: (8)                    e.g. ``ymin = 0.5`` always refers to the middle of the y-axis
796: (8)                    regardless of the limits set by `~.Axes.set_ylim`.
797: (8)                    Parameters
798: (8)                    ----------
799: (8)                    xmin : float
800: (12)                       Lower x-coordinate of the span, in data units.
801: (8)                    xmax : float
802: (12)                       Upper x-coordinate of the span, in data units.
803: (8)                    ymin : float, default: 0
804: (12)                       Lower y-coordinate of the span, in y-axis units (0-1).
805: (8)                    ymax : float, default: 1
806: (12)                       Upper y-coordinate of the span, in y-axis units (0-1).
807: (8)                    Returns
808: (8)                    -------
809: (8)                    `~matplotlib.patches.Rectangle`
810: (12)                       Vertical span (rectangle) from (xmin, ymin) to (xmax, ymax).
811: (8)                    Other Parameters
812: (8)                    ----------------
813: (8)                    **kwargs : `~matplotlib.patches.Rectangle` properties
814: (8)                    %(Rectangle:kwdoc)s
815: (8)                    See Also
816: (8)                    --------
817: (8)                    axhspan : Add a horizontal span across the Axes.
818: (8)                    Examples
819: (8)                    --------
820: (8)                    Draw a vertical, green, translucent rectangle from x = 1.25 to
821: (8)                    x = 1.55 that spans the yrange of the Axes.
822: (8)                    >>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
823: (8)                    """
824: (8)                    self._check_no_units([ymin, ymax], ['ymin', 'ymax'])
825: (8)                    (xmin, xmax), = self._process_unit_info([("x", [xmin, xmax])], kwargs)
826: (8)                    p = mpatches.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin,
**kwargs)
827: (8)                    p.set_transform(self.get_xaxis_transform(which="grid"))
828: (8)                    iy = self.dataLim.intervaly.copy()
829: (8)                    my = self.dataLim.minposy
830: (8)                    self.add_patch(p)
831: (8)                    self.dataLim.intervaly = iy
```

```
832: (8)                    self.dataLim.minposy = my
833: (8)                    p.get_path()._interpolation_steps =
mpl.axis.GRIDLINE_INTERPOLATION_STEPS
834: (8)                    self._request_autoscale_view("x")
835: (8)                    return p
836: (4)                @_preprocess_data(replace_names=["y", "xmin", "xmax", "colors"],
837: (22)                                  label_namer="y")
838: (4)                def hlines(self, y, xmin, xmax, colors=None, linestyles='solid',
839: (15)                           label='', **kwargs):
840: (8)                    """
841: (8)                    Plot horizontal lines at each *y* from *xmin* to *xmax*.
842: (8)                    Parameters
843: (8)                    ----------
844: (8)                    y : float or array-like
845: (12)                       y-indexes where to plot the lines.
846: (8)                    xmin, xmax : float or array-like
847: (12)                       Respective beginning and end of each line. If scalars are
848: (12)                       provided, all lines will have the same length.
849: (8)                    colors : :mpltype:`color` or list of color , default:
:rc:`lines.color`
850: (8)                    linestyles : {'solid', 'dashed', 'dashdot', 'dotted'}, default:
'solid'
851: (8)                    label : str, default: ''
852: (8)                    Returns
853: (8)                    -------
854: (8)                    `~matplotlib.collections.LineCollection`
855: (8)                    Other Parameters
856: (8)                    ----------------
857: (8)                    data : indexable object, optional
858: (12)                       DATA_PARAMETER_PLACEHOLDER
859: (8)                    **kwargs :  `~matplotlib.collections.LineCollection` properties.
860: (8)                    See Also
861: (8)                    --------
862: (8)                    vlines : vertical lines
863: (8)                    axhline : horizontal line across the Axes
864: (8)                    """
865: (8)                    xmin, xmax, y = self._process_unit_info(
866: (12)                       [("x", xmin), ("x", xmax), ("y", y)], kwargs)
867: (8)                    if not np.iterable(y):
868: (12)                       y = [y]
869: (8)                    if not np.iterable(xmin):
870: (12)                       xmin = [xmin]
871: (8)                    if not np.iterable(xmax):
872: (12)                       xmax = [xmax]
873: (8)                    y, xmin, xmax = cbook._combine_masks(y, xmin, xmax)
874: (8)                    y = np.ravel(y)
875: (8)                    xmin = np.ravel(xmin)
876: (8)                    xmax = np.ravel(xmax)
877: (8)                    masked_verts = np.ma.empty((len(y), 2, 2))
878: (8)                    masked_verts[:, 0, 0] = xmin
879: (8)                    masked_verts[:, 0, 1] = y
880: (8)                    masked_verts[:, 1, 0] = xmax
881: (8)                    masked_verts[:, 1, 1] = y
882: (8)                    lines = mcoll.LineCollection(masked_verts, colors=colors,
883: (37)                                                linestyles=linestyles, label=label)
884: (8)                    self.add_collection(lines, autolim=False)
885: (8)                    lines._internal_update(kwargs)
886: (8)                    if len(y) > 0:
887: (12)                       updatex = True
888: (12)                       updatey = True
889: (12)                       if self.name == "rectilinear":
890: (16)                           datalim = lines.get_datalim(self.transData)
891: (16)                           t = lines.get_transform()
892: (16)                           updatex, updatey =
t.contains_branch_seperately(self.transData)
893: (16)                           minx = np.nanmin(datalim.xmin)
894: (16)                           maxx = np.nanmax(datalim.xmax)
895: (16)                           miny = np.nanmin(datalim.ymin)
896: (16)                           maxy = np.nanmax(datalim.ymax)
```

```
897: (12)                        else:
898: (16)                            minx = np.nanmin(masked_verts[..., 0])
899: (16)                            maxx = np.nanmax(masked_verts[..., 0])
900: (16)                            miny = np.nanmin(masked_verts[..., 1])
901: (16)                            maxy = np.nanmax(masked_verts[..., 1])
902: (12)                        corners = (minx, miny), (maxx, maxy)
903: (12)                        self.update_datalim(corners, updatex, updatey)
904: (12)                        self._request_autoscale_view()
905: (8)                    return lines
906: (4)                @_preprocess_data(replace_names=["x", "ymin", "ymax", "colors"],
907: (22)                                  label_namer="x")
908: (4)                def vlines(self, x, ymin, ymax, colors=None, linestyles='solid',
909: (15)                           label='', **kwargs):
910: (8)                    """
911: (8)                    Plot vertical lines at each *x* from *ymin* to *ymax*.
912: (8)                    Parameters
913: (8)                    ----------
914: (8)                    x : float or array-like
915: (12)                       x-indexes where to plot the lines.
916: (8)                    ymin, ymax : float or array-like
917: (12)                       Respective beginning and end of each line. If scalars are
918: (12)                       provided, all lines will have the same length.
919: (8)                    colors : :mpltype:`color` or list of color, default: :rc:`lines.color`
920: (8)                    linestyles : {'solid', 'dashed', 'dashdot', 'dotted'}, default:
'solid'
921: (8)                    label : str, default: ''
922: (8)                    Returns
923: (8)                    -------
924: (8)                    `~matplotlib.collections.LineCollection`
925: (8)                    Other Parameters
926: (8)                    ----------------
927: (8)                    data : indexable object, optional
928: (12)                       DATA_PARAMETER_PLACEHOLDER
929: (8)                    **kwargs : `~matplotlib.collections.LineCollection` properties.
930: (8)                    See Also
931: (8)                    --------
932: (8)                    hlines : horizontal lines
933: (8)                    axvline : vertical line across the Axes
934: (8)                    """
935: (8)                    x, ymin, ymax = self._process_unit_info(
936: (12)                       [("x", x), ("y", ymin), ("y", ymax)], kwargs)
937: (8)                    if not np.iterable(x):
938: (12)                       x = [x]
939: (8)                    if not np.iterable(ymin):
940: (12)                       ymin = [ymin]
941: (8)                    if not np.iterable(ymax):
942: (12)                       ymax = [ymax]
943: (8)                    x, ymin, ymax = cbook._combine_masks(x, ymin, ymax)
944: (8)                    x = np.ravel(x)
945: (8)                    ymin = np.ravel(ymin)
946: (8)                    ymax = np.ravel(ymax)
947: (8)                    masked_verts = np.ma.empty((len(x), 2, 2))
948: (8)                    masked_verts[:, 0, 0] = x
949: (8)                    masked_verts[:, 0, 1] = ymin
950: (8)                    masked_verts[:, 1, 0] = x
951: (8)                    masked_verts[:, 1, 1] = ymax
952: (8)                    lines = mcoll.LineCollection(masked_verts, colors=colors,
953: (37)                                                linestyles=linestyles, label=label)
954: (8)                    self.add_collection(lines, autolim=False)
955: (8)                    lines._internal_update(kwargs)
956: (8)                    if len(x) > 0:
957: (12)                       updatex = True
958: (12)                       updatey = True
959: (12)                       if self.name == "rectilinear":
960: (16)                           datalim = lines.get_datalim(self.transData)
961: (16)                           t = lines.get_transform()
962: (16)                           updatex, updatey =
t.contains_branch_seperately(self.transData)
963: (16)                           minx = np.nanmin(datalim.xmin)
```

```
964: (16)                            maxx = np.nanmax(datalim.xmax)
965: (16)                            miny = np.nanmin(datalim.ymin)
966: (16)                            maxy = np.nanmax(datalim.ymax)
967: (12)                        else:
968: (16)                            minx = np.nanmin(masked_verts[..., 0])
969: (16)                            maxx = np.nanmax(masked_verts[..., 0])
970: (16)                            miny = np.nanmin(masked_verts[..., 1])
971: (16)                            maxy = np.nanmax(masked_verts[..., 1])
972: (12)                        corners = (minx, miny), (maxx, maxy)
973: (12)                        self.update_datalim(corners, updatex, updatey)
974: (12)                        self._request_autoscale_view()
975: (8)                    return lines
976: (4)                @_preprocess_data(replace_names=["positions", "lineoffsets",
977: (37)                                                 "linelengths", "linewidths",
978: (37)                                                 "colors", "linestyles"])
979: (4)                @_docstring.dedent_interpd
980: (4)                def eventplot(self, positions, orientation='horizontal', lineoffsets=1,
981: (18)                              linelengths=1, linewidths=None, colors=None, alpha=None,
982: (18)                              linestyles='solid', **kwargs):
983: (8)                    """
984: (8)                    Plot identical parallel lines at the given positions.
985: (8)                    This type of plot is commonly used in neuroscience for representing
986: (8)                    neural events, where it is usually called a spike raster, dot raster,
987: (8)                    or raster plot.
988: (8)                    However, it is useful in any situation where you wish to show the
989: (8)                    timing or position of multiple sets of discrete events, such as the
990: (8)                    arrival times of people to a business on each day of the month or the
991: (8)                    date of hurricanes each year of the last century.
992: (8)                    Parameters
993: (8)                    ----------
994: (8)                    positions : array-like or list of array-like
995: (12)                       A 1D array-like defines the positions of one sequence of events.
996: (12)                       Multiple groups of events may be passed as a list of array-likes.
997: (12)                       Each group can be styled independently by passing lists of values
998: (12)                       to *lineoffsets*, *linelengths*, *linewidths*, *colors* and
999: (12)                       *linestyles*.
1000: (12)                      Note that *positions* can be a 2D array, but in practice different
1001: (12)                      event groups usually have different counts so that one will use a
1002: (12)                      list of different-length arrays rather than a 2D array.
1003: (8)                   orientation : {'horizontal', 'vertical'}, default: 'horizontal'
1004: (12)                      The direction of the event sequence:
1005: (12)                      - 'horizontal': the events are arranged horizontally.
1006: (14)                        The indicator lines are vertical.
1007: (12)                      - 'vertical': the events are arranged vertically.
1008: (14)                        The indicator lines are horizontal.
1009: (8)                   lineoffsets : float or array-like, default: 1
1010: (12)                      The offset of the center of the lines from the origin, in the
1011: (12)                      direction orthogonal to *orientation*.
1012: (12)                      If *positions* is 2D, this can be a sequence with length matching
1013: (12)                      the length of *positions*.
1014: (8)                   linelengths : float or array-like, default: 1
1015: (12)                      The total height of the lines (i.e. the lines stretches from
1016: (12)                      ``lineoffset - linelength/2`` to ``lineoffset + linelength/2``).
1017: (12)                      If *positions* is 2D, this can be a sequence with length matching
1018: (12)                      the length of *positions*.
1019: (8)                   linewidths : float or array-like, default: :rc:`lines.linewidth`
1020: (12)                      The line width(s) of the event lines, in points.
1021: (12)                      If *positions* is 2D, this can be a sequence with length matching
1022: (12)                      the length of *positions*.
1023: (8)                   colors : :mpltype:`color` or list of color, default: :rc:`lines.color`
1024: (12)                      The color(s) of the event lines.
1025: (12)                      If *positions* is 2D, this can be a sequence with length matching
1026: (12)                      the length of *positions*.
1027: (8)                   alpha : float or array-like, default: 1
1028: (12)                      The alpha blending value(s), between 0 (transparent) and 1
1029: (12)                      (opaque).
1030: (12)                      If *positions* is 2D, this can be a sequence with length matching
1031: (12)                      the length of *positions*.
1032: (8)                   linestyles : str or tuple or list of such values, default: 'solid'
```

```
(12)                        Default is 'solid'. Valid strings are ['solid', 'dashed',
(12)                        'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples
(12)                        should be of the form::
(16)                            (offset, onoffseq),
(12)                        where *onoffseq* is an even length tuple of on and off ink
(12)                        in points.
(12)                        If *positions* is 2D, this can be a sequence with length matching
(12)                        the length of *positions*.
(8)                     data : indexable object, optional
(12)                        DATA_PARAMETER_PLACEHOLDER
(8)                     **kwargs
(12)                        Other keyword arguments are line collection properties.  See
(12)                        `.LineCollection` for a list of the valid properties.
(8)                     Returns
(8)                     -------
(8)                     list of `.EventCollection`
(12)                        The `.EventCollection` that were added.
(8)                     Notes
(8)                     -----
(8)                     For *linelengths*, *linewidths*, *colors*, *alpha* and *linestyles*,
if
(8)                     only a single value is given, that value is applied to all lines. If
an
(8)                     array-like is given, it must have the same length as *positions*, and
(8)                     each value will be applied to the corresponding row of the array.
(8)                     Examples
(8)                     --------
(8)                     .. plot:: gallery/lines_bars_and_markers/eventplot_demo.py
(8)                     """
(8)                     lineoffsets, linelengths = self._process_unit_info(
(16)                            [("y", lineoffsets), ("y", linelengths)], kwargs)
(8)                     if not np.iterable(positions):
(12)                        positions = [positions]
(8)                     elif any(np.iterable(position) for position in positions):
(12)                        positions = [np.asanyarray(position) for position in positions]
(8)                     else:
(12)                        positions = [np.asanyarray(positions)]
(8)                     poss = []
(8)                     for position in positions:
(12)                        poss += self._process_unit_info([("x", position)], kwargs)
(8)                     positions = poss
(8)                     colors = cbook._local_over_kwdict(colors, kwargs, 'color')
(8)                     linewidths = cbook._local_over_kwdict(linewidths, kwargs, 'linewidth')
(8)                     linestyles = cbook._local_over_kwdict(linestyles, kwargs, 'linestyle')
(8)                     if not np.iterable(lineoffsets):
(12)                        lineoffsets = [lineoffsets]
(8)                     if not np.iterable(linelengths):
(12)                        linelengths = [linelengths]
(8)                     if not np.iterable(linewidths):
(12)                        linewidths = [linewidths]
(8)                     if not np.iterable(colors):
(12)                        colors = [colors]
(8)                     if not np.iterable(alpha):
(12)                        alpha = [alpha]
(8)                     if hasattr(linestyles, 'lower') or not np.iterable(linestyles):
(12)                        linestyles = [linestyles]
(8)                     lineoffsets = np.asarray(lineoffsets)
(8)                     linelengths = np.asarray(linelengths)
(8)                     linewidths = np.asarray(linewidths)
(8)                     if len(lineoffsets) == 0:
(12)                        raise ValueError('lineoffsets cannot be empty')
(8)                     if len(linelengths) == 0:
(12)                        raise ValueError('linelengths cannot be empty')
(8)                     if len(linestyles) == 0:
(12)                        raise ValueError('linestyles cannot be empty')
(8)                     if len(linewidths) == 0:
(12)                        raise ValueError('linewidths cannot be empty')
(8)                     if len(alpha) == 0:
(12)                        raise ValueError('alpha cannot be empty')
```

```
(8)                 if len(colors) == 0:
(12)                    colors = [None]
(8)                 try:
(12)                    colors = mcolors.to_rgba_array(colors)
(8)                 except ValueError:
(12)                    pass
(8)                 if len(lineoffsets) == 1 and len(positions) != 1:
(12)                    lineoffsets = np.tile(lineoffsets, len(positions))
(12)                    lineoffsets[0] = 0
(12)                    lineoffsets = np.cumsum(lineoffsets)
(8)                 if len(linelengths) == 1:
(12)                    linelengths = np.tile(linelengths, len(positions))
(8)                 if len(linewidths) == 1:
(12)                    linewidths = np.tile(linewidths, len(positions))
(8)                 if len(colors) == 1:
(12)                    colors = list(colors) * len(positions)
(8)                 if len(alpha) == 1:
(12)                    alpha = list(alpha) * len(positions)
(8)                 if len(linestyles) == 1:
(12)                    linestyles = [linestyles] * len(positions)
(8)                 if len(lineoffsets) != len(positions):
(12)                    raise ValueError('lineoffsets and positions are unequal sized '
(29)                                     'sequences')
(8)                 if len(linelengths) != len(positions):
(12)                    raise ValueError('linelengths and positions are unequal sized '
(29)                                     'sequences')
(8)                 if len(linewidths) != len(positions):
(12)                    raise ValueError('linewidths and positions are unequal sized '
(29)                                     'sequences')
(8)                 if len(colors) != len(positions):
(12)                    raise ValueError('colors and positions are unequal sized '
(29)                                     'sequences')
(8)                 if len(alpha) != len(positions):
(12)                    raise ValueError('alpha and positions are unequal sized '
(29)                                     'sequences')
(8)                 if len(linestyles) != len(positions):
(12)                    raise ValueError('linestyles and positions are unequal sized '
(29)                                     'sequences')
(8)                 colls = []
(8)                 for position, lineoffset, linelength, linewidth, color, alpha_, \
(12)                    linestyle in \
(16)                        zip(positions, lineoffsets, linelengths, linewidths,
(20)                            colors, alpha, linestyles):
(12)                    coll = mcoll.EventCollection(position,
(41)                                                 orientation=orientation,
(41)                                                 lineoffset=lineoffset,
(41)                                                 linelength=linelength,
(41)                                                 linewidth=linewidth,
(41)                                                 color=color,
(41)                                                 alpha=alpha_,
(41)                                                 linestyle=linestyle)
(12)                    self.add_collection(coll, autolim=False)
(12)                    coll._internal_update(kwargs)
(12)                    colls.append(coll)
(8)                 if len(positions) > 0:
(12)                    min_max = [(np.min(_p), np.max(_p)) for _p in positions
(23)                               if len(_p) > 0]
(12)                    if len(min_max) > 0:
(16)                        mins, maxes = zip(*min_max)
(16)                        minpos = np.min(mins)
(16)                        maxpos = np.max(maxes)
(16)                        minline = (lineoffsets - linelengths).min()
(16)                        maxline = (lineoffsets + linelengths).max()
(16)                        if orientation == "vertical":
(20)                            corners = (minline, minpos), (maxline, maxpos)
(16)                        else:  # "horizontal"
(20)                            corners = (minpos, minline), (maxpos, maxline)
(16)                        self.update_datalim(corners)
(16)                        self._request_autoscale_view()
```

```
        return colls
    @_docstring.dedent_interpd
    def plot(self, *args, scalex=True, scaley=True, data=None, **kwargs):
        """
        Plot y versus x as lines and/or markers.
        Call signatures::
            plot([x], y, [fmt], *, data=None, **kwargs)
            plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
        The coordinates of the points or line nodes are given by *x*, *y*.
        The optional parameter *fmt* is a convenient way for defining basic
        formatting like color, marker and linestyle. It's a shortcut string
        notation described in the *Notes* section below.
        >>> plot(x, y)        # plot x and y using default line style and
color
        >>> plot(x, y, 'bo')  # plot x and y using blue circle markers
        >>> plot(y)           # plot y using x as index array 0..N-1
        >>> plot(y, 'r+')     # ditto, but with red plusses
        You can use `.Line2D` properties as keyword arguments for more
        control on the appearance. Line properties and *fmt* can be mixed.
        The following two calls yield identical results:
        >>> plot(x, y, 'go--', linewidth=2, markersize=12)
        >>> plot(x, y, color='green', marker='o', linestyle='dashed',
        ...      linewidth=2, markersize=12)
        When conflicting with *fmt*, keyword arguments take precedence.
        **Plotting labelled data**
        There's a convenient way for plotting objects with labelled data (i.e.
        data that can be accessed by index ``obj['y']``). Instead of giving
        the data in *x* and *y*, you can provide the object in the *data*
        parameter and just give the labels for *x* and *y*::
        >>> plot('xlabel', 'ylabel', data=obj)
        All indexable objects are supported. This could e.g. be a `dict`, a
        `pandas.DataFrame` or a structured numpy array.
        **Plotting multiple sets of data**
        There are various ways to plot multiple sets of data.
        - The most straight forward way is just to call `plot` multiple times.
          Example:
          >>> plot(x1, y1, 'bo')
          >>> plot(x2, y2, 'go')
        - If *x* and/or *y* are 2D arrays, a separate data set will be drawn
          for every column. If both *x* and *y* are 2D, they must have the
          same shape. If only one of them is 2D with shape (N, m) the other
          must have length N and will be used for every data set m.
          Example:
          >>> x = [1, 2, 3]
          >>> y = np.array([[1, 2], [3, 4], [5, 6]])
          >>> plot(x, y)
          is equivalent to:
          >>> for col in range(y.shape[1]):
          ...     plot(x, y[:, col])
        - The third way is to specify multiple sets of *[x]*, *y*, *[fmt]*
          groups::
          >>> plot(x1, y1, 'g^', x2, y2, 'g-')
          In this case, any additional keyword argument applies to all
          datasets. Also, this syntax cannot be combined with the *data*
          parameter.
        By default, each line is assigned a different style specified by a
        'style cycle'. The *fmt* and line property parameters are only
        necessary if you want explicit deviations from these defaults.
        Alternatively, you can also change the style cycle using
        :rc:`axes.prop_cycle`.
        Parameters
        ----------
        x, y : array-like or scalar
            The horizontal / vertical coordinates of the data points.
            *x* values are optional and default to ``range(len(y))``.
            Commonly, these parameters are 1D arrays.
            They can also be scalars, or two-dimensional (in that case, the
            columns represent separate data sets).
            These arguments cannot be passed as keywords.
```

```
(8)                 fmt : str, optional
(12)                    A format string, e.g. 'ro' for red circles. See the *Notes*
(12)                    section for a full description of the format strings.
(12)                    Format strings are just an abbreviation for quickly setting
(12)                    basic line properties. All of these and more can also be
(12)                    controlled by keyword arguments.
(12)                    This argument cannot be passed as keyword.
(8)                 data : indexable object, optional
(12)                    An object with labelled data. If given, provide the label names to
(12)                    plot in *x* and *y*.
(12)                    .. note::
(16)                        Technically there's a slight ambiguity in calls where the
(16)                        second label is a valid *fmt*. ``plot('n', 'o', data=obj)``
(16)                        could be ``plt(x, y)`` or ``plt(y, fmt)``. In such cases,
(16)                        the former interpretation is chosen, but a warning is issued.
(16)                        You may suppress the warning by adding an empty format string
(16)                        ``plot('n', 'o', '', data=obj)``.
(8)                 Returns
(8)                 -------
(8)                 list of `.Line2D`
(12)                    A list of lines representing the plotted data.
(8)                 Other Parameters
(8)                 ----------------
(8)                 scalex, scaley : bool, default: True
(12)                    These parameters determine if the view limits are adapted to the
(12)                    data limits. The values are passed on to
(12)                    `~.axes.Axes.autoscale_view`.
(8)                 **kwargs : `~matplotlib.lines.Line2D` properties, optional
(12)                    *kwargs* are used to specify properties like a line label (for
(12)                    auto legends), linewidth, antialiasing, marker face color.
(12)                    Example::
(12)                    >>> plot([1, 2, 3], [1, 2, 3], 'go-', label='line 1', linewidth=2)
(12)                    >>> plot([1, 2, 3], [1, 4, 9], 'rs', label='line 2')
(12)                    If you specify multiple lines with one plot call, the kwargs apply
(12)                    to all those lines. In case the label object is iterable, each
(12)                    element is used as labels for each set of data.
(12)                    Here is a list of available `.Line2D` properties:
(12)                    %(Line2D:kwdoc)s
(8)                 See Also
(8)                 --------
(8)                 scatter : XY scatter plot with markers of varying size and/or color (
(12)                    sometimes also called bubble chart).
(8)                 Notes
(8)                 -----
(8)                 **Format Strings**
(8)                 A format string consists of a part for color, marker and line::
(12)                    fmt = '[marker][line][color]'
(8)                 Each of them is optional. If not provided, the value from the style
(8)                 cycle is used. Exception: If ``line`` is given, but no ``marker``,
(8)                 the data will be a line without markers.
(8)                 Other combinations such as ``[color][marker][line]`` are also
(8)                 supported, but note that their parsing may be ambiguous.
(8)                 **Markers**
(8)                 =============   ==============================
(8)                 character       description
(8)                 =============   ==============================
(8)                 ``'.'``         point marker
(8)                 ``','``         pixel marker
(8)                 ``'o'``         circle marker
(8)                 ``'v'``         triangle_down marker
(8)                 ``'^'``         triangle_up marker
(8)                 ``'<'``         triangle_left marker
(8)                 ``'>'``         triangle_right marker
(8)                 ``'1'``         tri_down marker
(8)                 ``'2'``         tri_up marker
(8)                 ``'3'``         tri_left marker
(8)                 ``'4'``         tri_right marker
(8)                 ``'8'``         octagon marker
(8)                 ``'s'``         square marker
```

```
1306: (8)                    ``'p'``         pentagon marker
1307: (8)                    ``'P'``         plus (filled) marker
1308: (8)                    ``'*'``         star marker
1309: (8)                    ``'h'``         hexagon1 marker
1310: (8)                    ``'H'``         hexagon2 marker
1311: (8)                    ``'+'``         plus marker
1312: (8)                    ``'x'``         x marker
1313: (8)                    ``'X'``         x (filled) marker
1314: (8)                    ``'D'``         diamond marker
1315: (8)                    ``'d'``         thin_diamond marker
1316: (8)                    ``'|'``         vline marker
1317: (8)                    ``'_'``         hline marker
1318: (8)                    =============   ==============================
1319: (8)                    **Line Styles**
1320: (8)                    =============    ==============================
1321: (8)                    character        description
1322: (8)                    =============    ==============================
1323: (8)                    ``'-'``          solid line style
1324: (8)                    ``'--'``         dashed line style
1325: (8)                    ``'-.'``         dash-dot line style
1326: (8)                    ``':'``          dotted line style
1327: (8)                    =============    ==============================
1328: (8)                    Example format strings::
1329: (12)                       'b'    # blue markers with default shape
1330: (12)                       'or'   # red circles
1331: (12)                       '-g'   # green solid line
1332: (12)                       '--'   # dashed line with default color
1333: (12)                       '^k:'  # black triangle_up markers connected by a dotted line
1334: (8)                    **Colors**
1335: (8)                    The supported color abbreviations are the single letter codes
1336: (8)                    =============    ==============================
1337: (8)                    character        color
1338: (8)                    =============    ==============================
1339: (8)                    ``'b'``          blue
1340: (8)                    ``'g'``          green
1341: (8)                    ``'r'``          red
1342: (8)                    ``'c'``          cyan
1343: (8)                    ``'m'``          magenta
1344: (8)                    ``'y'``          yellow
1345: (8)                    ``'k'``          black
1346: (8)                    ``'w'``          white
1347: (8)                    =============    ==============================
1348: (8)                    and the ``'CN'`` colors that index into the default property cycle.
1349: (8)                    If the color is the only part of the format string, you can
1350: (8)                    additionally use any  `matplotlib.colors` spec, e.g. full names
1351: (8)                    (``'green'``) or hex strings (``'#008000'``).
1352: (8)                    """
1353: (8)                    kwargs = cbook.normalize_kwargs(kwargs, mlines.Line2D)
1354: (8)                    lines = [*self._get_lines(self, *args, data=data, **kwargs)]
1355: (8)                    for line in lines:
1356: (12)                       self.add_line(line)
1357: (8)                    if scalex:
1358: (12)                       self._request_autoscale_view("x")
1359: (8)                    if scaley:
1360: (12)                       self._request_autoscale_view("y")
1361: (8)                    return lines
1362: (4)                @_api.deprecated("3.9", alternative="plot")
1363: (4)                @_preprocess_data(replace_names=["x", "y"], label_namer="y")
1364: (4)                @_docstring.dedent_interpd
1365: (4)                def plot_date(self, x, y, fmt='o', tz=None, xdate=True, ydate=False,
1366: (18)                              **kwargs):
1367: (8)                    """
1368: (8)                    Plot coercing the axis to treat floats as dates.
1369: (8)                    .. deprecated:: 3.9
1370: (12)                       This method exists for historic reasons and will be removed in
version 3.11.
1371: (12)                       - ``datetime``-like data should directly be plotted using
1372: (14)                         `~.Axes.plot`.
1373: (12)                       -  If you need to plot plain numeric data as :ref:`date-format` or
```

```
(15)                        need to set a timezone, call ``ax.xaxis.axis_date`` /
(15)                        ``ax.yaxis.axis_date`` before `~.Axes.plot`. See
(15)                        `.Axis.axis_date`.
(8)                     Similar to `.plot`, this plots *y* vs. *x* as lines or markers.
(8)                     However, the axis labels are formatted as dates depending on *xdate*
(8)                     and *ydate*.  Note that `.plot` will work with `datetime` and
(8)                     `numpy.datetime64` objects without resorting to this method.
(8)                     Parameters
(8)                     ----------
(8)                     x, y : array-like
(12)                        The coordinates of the data points. If *xdate* or *ydate* is
(12)                        *True*, the respective values *x* or *y* are interpreted as
(12)                        :ref:`Matplotlib dates <date-format>`.
(8)                     fmt : str, optional
(12)                        The plot format string. For details, see the corresponding
(12)                        parameter in `.plot`.
(8)                     tz : timezone string or `datetime.tzinfo`, default: :rc:`timezone`
(12)                        The time zone to use in labeling dates.
(8)                     xdate : bool, default: True
(12)                        If *True*, the *x*-axis will be interpreted as Matplotlib dates.
(8)                     ydate : bool, default: False
(12)                        If *True*, the *y*-axis will be interpreted as Matplotlib dates.
(8)                     Returns
(8)                     -------
(8)                     list of `.Line2D`
(12)                        Objects representing the plotted data.
(8)                     Other Parameters
(8)                     ----------------
(8)                     data : indexable object, optional
(12)                        DATA_PARAMETER_PLACEHOLDER
(8)                     **kwargs
(12)                        Keyword arguments control the `.Line2D` properties:
(12)                        %(Line2D:kwdoc)s
(8)                     See Also
(8)                     --------
(8)                     matplotlib.dates : Helper functions on dates.
(8)                     matplotlib.dates.date2num : Convert dates to num.
(8)                     matplotlib.dates.num2date : Convert num to dates.
(8)                     matplotlib.dates.drange : Create an equally spaced sequence of dates.
(8)                     Notes
(8)                     -----
(8)                     If you are using custom date tickers and formatters, it may be
(8)                     necessary to set the formatters/locators after the call to
(8)                     `.plot_date`. `.plot_date` will set the default tick locator to
(8)                     `.AutoDateLocator` (if the tick locator is not already set to a
(8)                     `.DateLocator` instance) and the default tick formatter to
(8)                     `.AutoDateFormatter` (if the tick formatter is not already set to a
(8)                     `.DateFormatter` instance).
(8)                     """
(8)                     if xdate:
(12)                        self.xaxis_date(tz)
(8)                     if ydate:
(12)                        self.yaxis_date(tz)
(8)                     return self.plot(x, y, fmt, **kwargs)
(4)                 @_docstring.dedent_interpd
(4)                 def loglog(self, *args, **kwargs):
(8)                     """
(8)                     Make a plot with log scaling on both the x- and y-axis.
(8)                     Call signatures::
(12)                        loglog([x], y, [fmt], data=None, **kwargs)
(12)                        loglog([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
(8)                     This is just a thin wrapper around `.plot` which additionally changes
(8)                     both the x-axis and the y-axis to log scaling. All the concepts and
(8)                     parameters of plot can be used here as well.
(8)                     The additional parameters *base*, *subs* and *nonpositive* control the
(8)                     x/y-axis properties. They are just forwarded to `.Axes.set_xscale` and
(8)                     `.Axes.set_yscale`. To use different properties on the x-axis and the
(8)                     y-axis, use e.g.
(8)                     ``ax.set_xscale("log", base=10); ax.set_yscale("log", base=2)``.
```

```
(8)                     Parameters
(8)                     ----------
(8)                     base : float, default: 10
(12)                        Base of the logarithm.
(8)                     subs : sequence, optional
(12)                        The location of the minor ticks. If *None*, reasonable locations
(12)                        are automatically chosen depending on the number of decades in the
(12)                        plot. See `.Axes.set_xscale`/`.Axes.set_yscale` for details.
(8)                     nonpositive : {'mask', 'clip'}, default: 'clip'
(12)                        Non-positive values can be masked as invalid, or clipped to a very
(12)                        small positive number.
(8)                     **kwargs
(12)                        All parameters supported by `.plot`.
(8)                     Returns
(8)                     -------
(8)                     list of `.Line2D`
(12)                        Objects representing the plotted data.
(8)                     """
(8)                     dx = {k: v for k, v in kwargs.items()
(14)                          if k in ['base', 'subs', 'nonpositive',
(23)                                   'basex', 'subsx', 'nonposx']}
(8)                     self.set_xscale('log', **dx)
(8)                     dy = {k: v for k, v in kwargs.items()
(14)                          if k in ['base', 'subs', 'nonpositive',
(23)                                   'basey', 'subsy', 'nonposy']}
(8)                     self.set_yscale('log', **dy)
(8)                     return self.plot(
(12)                        *args, **{k: v for k, v in kwargs.items() if k not in {*dx, *dy}})
(4)                 @_docstring.dedent_interpd
(4)                 def semilogx(self, *args, **kwargs):
(8)                     """
(8)                     Make a plot with log scaling on the x-axis.
(8)                     Call signatures::
(12)                        semilogx([x], y, [fmt], data=None, **kwargs)
(12)                        semilogx([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
(8)                     This is just a thin wrapper around `.plot` which additionally changes
(8)                     the x-axis to log scaling. All the concepts and parameters of plot can
(8)                     be used here as well.
(8)                     The additional parameters *base*, *subs*, and *nonpositive* control
the
(8)                     x-axis properties. They are just forwarded to `.Axes.set_xscale`.
(8)                     Parameters
(8)                     ----------
(8)                     base : float, default: 10
(12)                        Base of the x logarithm.
(8)                     subs : array-like, optional
(12)                        The location of the minor xticks. If *None*, reasonable locations
(12)                        are automatically chosen depending on the number of decades in the
(12)                        plot. See `.Axes.set_xscale` for details.
(8)                     nonpositive : {'mask', 'clip'}, default: 'clip'
(12)                        Non-positive values in x can be masked as invalid, or clipped to a
(12)                        very small positive number.
(8)                     **kwargs
(12)                        All parameters supported by `.plot`.
(8)                     Returns
(8)                     -------
(8)                     list of `.Line2D`
(12)                        Objects representing the plotted data.
(8)                     """
(8)                     d = {k: v for k, v in kwargs.items()
(13)                         if k in ['base', 'subs', 'nonpositive',
(22)                                  'basex', 'subsx', 'nonposx']}
(8)                     self.set_xscale('log', **d)
(8)                     return self.plot(
(12)                        *args, **{k: v for k, v in kwargs.items() if k not in d})
(4)                 @_docstring.dedent_interpd
(4)                 def semilogy(self, *args, **kwargs):
(8)                     """
(8)                     Make a plot with log scaling on the y-axis.
```

```
(8)                     Call signatures::
(12)                        semilogy([x], y, [fmt], data=None, **kwargs)
(12)                        semilogy([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
(8)                     This is just a thin wrapper around `.plot` which additionally changes
(8)                     the y-axis to log scaling. All the concepts and parameters of plot can
(8)                     be used here as well.
(8)                     The additional parameters *base*, *subs*, and *nonpositive* control
the
(8)                     y-axis properties. They are just forwarded to `.Axes.set_yscale`.
(8)                     Parameters
(8)                     ----------
(8)                     base : float, default: 10
(12)                        Base of the y logarithm.
(8)                     subs : array-like, optional
(12)                        The location of the minor yticks. If *None*, reasonable locations
(12)                        are automatically chosen depending on the number of decades in the
(12)                        plot. See `.Axes.set_yscale` for details.
(8)                     nonpositive : {'mask', 'clip'}, default: 'clip'
(12)                        Non-positive values in y can be masked as invalid, or clipped to a
(12)                        very small positive number.
(8)                     **kwargs
(12)                        All parameters supported by `.plot`.
(8)                     Returns
(8)                     -------
(8)                     list of `.Line2D`
(12)                        Objects representing the plotted data.
(8)                     """
(8)                     d = {k: v for k, v in kwargs.items()
(13)                         if k in ['base', 'subs', 'nonpositive',
(22)                                  'basey', 'subsy', 'nonposy']}
(8)                     self.set_yscale('log', **d)
(8)                     return self.plot(
(12)                        *args, **{k: v for k, v in kwargs.items() if k not in d})
(4)                 @_preprocess_data(replace_names=["x"], label_namer="x")
(4)                 def acorr(self, x, **kwargs):
(8)                     """
(8)                     Plot the autocorrelation of *x*.
(8)                     Parameters
(8)                     ----------
(8)                     x : array-like
(12)                        Not run through Matplotlib's unit conversion, so this should
(12)                        be a unit-less array.
(8)                     detrend : callable, default: `.mlab.detrend_none` (no detrending)
(12)                        A detrending function applied to *x*.  It must have the
(12)                        signature ::
(16)                            detrend(x: np.ndarray) -> np.ndarray
(8)                     normed : bool, default: True
(12)                        If ``True``, input vectors are normalised to unit length.
(8)                     usevlines : bool, default: True
(12)                        Determines the plot style.
(12)                        If ``True``, vertical lines are plotted from 0 to the acorr value
(12)                        using `.Axes.vlines`. Additionally, a horizontal line is plotted
(12)                        at y=0 using `.Axes.axhline`.
(12)                        If ``False``, markers are plotted at the acorr values using
(12)                        `.Axes.plot`.
(8)                     maxlags : int, default: 10
(12)                        Number of lags to show. If ``None``, will return all
(12)                        ``2 * len(x) - 1`` lags.
(8)                     Returns
(8)                     -------
(8)                     lags : array (length ``2*maxlags+1``)
(12)                        The lag vector.
(8)                     c : array  (length ``2*maxlags+1``)
(12)                        The auto correlation vector.
(8)                     line : `.LineCollection` or `.Line2D`
(12)                        `.Artist` added to the Axes of the correlation:
(12)                        - `.LineCollection` if *usevlines* is True.
(12)                        - `.Line2D` if *usevlines* is False.
(8)                     b : `~matplotlib.lines.Line2D` or None
```

```
(12)                    Horizontal line at 0 if *usevlines* is True
(12)                    None *usevlines* is False.
(8)                 Other Parameters
(8)                 ----------------
(8)                 linestyle : `~matplotlib.lines.Line2D` property, optional
(12)                    The linestyle for plotting the data points.
(12)                    Only used if *usevlines* is ``False``.
(8)                 marker : str, default: 'o'
(12)                    The marker for plotting the data points.
(12)                    Only used if *usevlines* is ``False``.
(8)                 data : indexable object, optional
(12)                    DATA_PARAMETER_PLACEHOLDER
(8)                 **kwargs
(12)                    Additional parameters are passed to `.Axes.vlines` and
(12)                    `.Axes.axhline` if *usevlines* is ``True``; otherwise they are
(12)                    passed to `.Axes.plot`.
(8)                 Notes
(8)                 -----
(8)                 The cross correlation is performed with `numpy.correlate` with
(8)                 ``mode = "full"``.
(8)                 """
(8)                 return self.xcorr(x, x, **kwargs)
(4)             @_preprocess_data(replace_names=["x", "y"], label_namer="y")
(4)             def xcorr(self, x, y, normed=True, detrend=mlab.detrend_none,
(14)                      usevlines=True, maxlags=10, **kwargs):
(8)                 r"""
(8)                 Plot the cross correlation between *x* and *y*.
(8)                 The correlation with lag k is defined as
(8)                 :math:`\sum_n x[n+k] \cdot y^*[n]`, where :math:`y^*` is the complex
(8)                 conjugate of :math:`y`.
(8)                 Parameters
(8)                 ----------
(8)                 x, y : array-like of length n
(12)                    Neither *x* nor *y* are run through Matplotlib's unit conversion,
so
(12)                    these should be unit-less arrays.
(8)                 detrend : callable, default: `.mlab.detrend_none` (no detrending)
(12)                    A detrending function applied to *x* and *y*.  It must have the
(12)                    signature ::
(16)                        detrend(x: np.ndarray) -> np.ndarray
(8)                 normed : bool, default: True
(12)                    If ``True``, input vectors are normalised to unit length.
(8)                 usevlines : bool, default: True
(12)                    Determines the plot style.
(12)                    If ``True``, vertical lines are plotted from 0 to the xcorr value
(12)                    using `.Axes.vlines`. Additionally, a horizontal line is plotted
(12)                    at y=0 using `.Axes.axhline`.
(12)                    If ``False``, markers are plotted at the xcorr values using
(12)                    `.Axes.plot`.
(8)                 maxlags : int, default: 10
(12)                    Number of lags to show. If None, will return all ``2 * len(x) -
1``
(12)                    lags.
(8)                 Returns
(8)                 -------
(8)                 lags : array (length ``2*maxlags+1``)
(12)                    The lag vector.
(8)                 c : array  (length ``2*maxlags+1``)
(12)                    The auto correlation vector.
(8)                 line : `.LineCollection` or `.Line2D`
(12)                    `.Artist` added to the Axes of the correlation:
(12)                    - `.LineCollection` if *usevlines* is True.
(12)                    - `.Line2D` if *usevlines* is False.
(8)                 b : `~matplotlib.lines.Line2D` or None
(12)                    Horizontal line at 0 if *usevlines* is True
(12)                    None *usevlines* is False.
(8)                 Other Parameters
(8)                 ----------------
(8)                 linestyle : `~matplotlib.lines.Line2D` property, optional
```

```
1646: (12)                      The linestyle for plotting the data points.
1647: (12)                      Only used if *usevlines* is ``False``.
1648: (8)                   marker : str, default: 'o'
1649: (12)                      The marker for plotting the data points.
1650: (12)                      Only used if *usevlines* is ``False``.
1651: (8)                   data : indexable object, optional
1652: (12)                      DATA_PARAMETER_PLACEHOLDER
1653: (8)                   **kwargs
1654: (12)                      Additional parameters are passed to `.Axes.vlines` and
1655: (12)                      `.Axes.axhline` if *usevlines* is ``True``; otherwise they are
1656: (12)                      passed to `.Axes.plot`.
1657: (8)                   Notes
1658: (8)                   -----
1659: (8)                   The cross correlation is performed with `numpy.correlate` with
1660: (8)                   ``mode = "full"``.
1661: (8)                   """
1662: (8)                   Nx = len(x)
1663: (8)                   if Nx != len(y):
1664: (12)                      raise ValueError('x and y must be equal length')
1665: (8)                   x = detrend(np.asarray(x))
1666: (8)                   y = detrend(np.asarray(y))
1667: (8)                   correls = np.correlate(x, y, mode="full")
1668: (8)                   if normed:
1669: (12)                      correls = correls / np.sqrt(np.dot(x, x) * np.dot(y, y))
1670: (8)                   if maxlags is None:
1671: (12)                      maxlags = Nx - 1
1672: (8)                   if maxlags >= Nx or maxlags < 1:
1673: (12)                      raise ValueError('maxlags must be None or strictly '
1674: (29)                                       'positive < %d' % Nx)
1675: (8)                   lags = np.arange(-maxlags, maxlags + 1)
1676: (8)                   correls = correls[Nx - 1 - maxlags:Nx + maxlags]
1677: (8)                   if usevlines:
1678: (12)                      a = self.vlines(lags, [0], correls, **kwargs)
1679: (12)                      kwargs.pop('label', '')
1680: (12)                      b = self.axhline(**kwargs)
1681: (8)                   else:
1682: (12)                      kwargs.setdefault('marker', 'o')
1683: (12)                      kwargs.setdefault('linestyle', 'None')
1684: (12)                      a, = self.plot(lags, correls, **kwargs)
1685: (12)                      b = None
1686: (8)                   return lags, correls, a, b
1687: (4)               def step(self, x, y, *args, where='pre', data=None, **kwargs):
1688: (8)                   """
1689: (8)                   Make a step plot.
1690: (8)                   Call signatures::
1691: (12)                      step(x, y, [fmt], *, data=None, where='pre', **kwargs)
1692: (12)                      step(x, y, [fmt], x2, y2, [fmt2], ..., *, where='pre', **kwargs)
1693: (8)                   This is just a thin wrapper around `.plot` which changes some
1694: (8)                   formatting options. Most of the concepts and parameters of plot can be
1695: (8)                   used here as well.
1696: (8)                   .. note::
1697: (12)                      This method uses a standard plot with a step drawstyle: The *x*
1698: (12)                      values are the reference positions and steps extend
left/right/both
1699: (12)                      directions depending on *where*.
1700: (12)                      For the common case where you know the values and edges of the
1701: (12)                      steps, use `~.Axes.stairs` instead.
1702: (8)                   Parameters
1703: (8)                   ----------
1704: (8)                   x : array-like
1705: (12)                      1D sequence of x positions. It is assumed, but not checked, that
1706: (12)                      it is uniformly increasing.
1707: (8)                   y : array-like
1708: (12)                      1D sequence of y levels.
1709: (8)                   fmt : str, optional
1710: (12)                      A format string, e.g. 'g' for a green line. See `.plot` for a more
1711: (12)                      detailed description.
1712: (12)                      Note: While full format strings are accepted, it is recommended to
1713: (12)                      only specify the color. Line styles are currently ignored (use
```

```
(12)                        the keyword argument *linestyle* instead). Markers are accepted
(12)                        and plotted on the given positions, however, this is a rarely
(12)                        needed feature for step plots.
(8)                     where : {'pre', 'post', 'mid'}, default: 'pre'
(12)                        Define where the steps should be placed:
(12)                        - 'pre': The y value is continued constantly to the left from
(14)                          every *x* position, i.e. the interval ``(x[i-1], x[i]]`` has the
(14)                          value ``y[i]``.
(12)                        - 'post': The y value is continued constantly to the right from
(14)                          every *x* position, i.e. the interval ``[x[i], x[i+1])`` has the
(14)                          value ``y[i]``.
(12)                        - 'mid': Steps occur half-way between the *x* positions.
(8)                     data : indexable object, optional
(12)                        An object with labelled data. If given, provide the label names to
(12)                        plot in *x* and *y*.
(8)                     **kwargs
(12)                        Additional parameters are the same as those for `.plot`.
(8)                     Returns
(8)                     -------
(8)                     list of `.Line2D`
(12)                        Objects representing the plotted data.
(8)                     """
(8)                     _api.check_in_list(('pre', 'post', 'mid'), where=where)
(8)                     kwargs['drawstyle'] = 'steps-' + where
(8)                     return self.plot(x, y, *args, data=data, **kwargs)
(4)                 @staticmethod
(4)                 def _convert_dx(dx, x0, xconv, convert):
(8)                     """
(8)                     Small helper to do logic of width conversion flexibly.
(8)                     *dx* and *x0* have units, but *xconv* has already been converted
(8)                     to unitless (and is an ndarray).  This allows the *dx* to have units
(8)                     that are different from *x0*, but are still accepted by the
(8)                     ``__add__`` operator of *x0*.
(8)                     """
(8)                     assert type(xconv) is np.ndarray
(8)                     if xconv.size == 0:
(12)                        return convert(dx)
(8)                     try:
(12)                        try:
(16)                            x0 = cbook._safe_first_finite(x0)
(12)                        except (TypeError, IndexError, KeyError):
(16)                            pass
(12)                        try:
(16)                            x = cbook._safe_first_finite(xconv)
(12)                        except (TypeError, IndexError, KeyError):
(16)                            x = xconv
(12)                        delist = False
(12)                        if not np.iterable(dx):
(16)                            dx = [dx]
(16)                            delist = True
(12)                        dx = [convert(x0 + ddx) - x for ddx in dx]
(12)                        if delist:
(16)                            dx = dx[0]
(8)                     except (ValueError, TypeError, AttributeError):
(12)                        dx = convert(dx)
(8)                     return dx
(4)                 @_preprocess_data()
(4)                 @_docstring.dedent_interpd
(4)                 def bar(self, x, height, width=0.8, bottom=None, *, align="center",
(12)                        **kwargs):
(8)                     r"""
(8)                     Make a bar plot.
(8)                     The bars are positioned at *x* with the given *align*\ment. Their
(8)                     dimensions are given by *height* and *width*. The vertical baseline
(8)                     is *bottom* (default 0).
(8)                     Many parameters can take either a single value applying to all bars
(8)                     or a sequence of values, one for each bar.
(8)                     Parameters
(8)                     ----------
```

```
                    x : float or array-like
                        The x coordinates of the bars. See also *align* for the
                        alignment of the bars to the coordinates.
                    height : float or array-like
                        The height(s) of the bars.
                        Note that if *bottom* has units (e.g. datetime), *height* should be in
                        units that are a difference from the value of *bottom* (e.g. timedelta).
                    width : float or array-like, default: 0.8
                        The width(s) of the bars.
                        Note that if *x* has units (e.g. datetime), then *width* should be in
                        units that are a difference (e.g. timedelta) around the *x* values.
                    bottom : float or array-like, default: 0
                        The y coordinate(s) of the bottom side(s) of the bars.
                        Note that if *bottom* has units, then the y-axis will get a Locator and
                        Formatter appropriate for the units (e.g. dates, or categorical).
                    align : {'center', 'edge'}, default: 'center'
                        Alignment of the bars to the *x* coordinates:
                        - 'center': Center the base on the *x* positions.
                        - 'edge': Align the left edges of the bars with the *x* positions.
                        To align the bars on the right edge pass a negative *width* and
                        ``align='edge'``.
                    Returns
                    -------
                    `.BarContainer`
                        Container with all the bars and optionally errorbars.
                    Other Parameters
                    ----------------
                    color : :mpltype:`color` or list of :mpltype:`color`, optional
                        The colors of the bar faces.
                    edgecolor : :mpltype:`color` or list of :mpltype:`color`, optional
                        The colors of the bar edges.
                    linewidth : float or array-like, optional
                        Width of the bar edge(s). If 0, don't draw edges.
                    tick_label : str or list of str, optional
                        The tick labels of the bars.
                        Default: None (Use default numeric labels.)
                    label : str or list of str, optional
                        A single label is attached to the resulting `.BarContainer` as a
                        label for the whole dataset.
                        If a list is provided, it must be the same length as *x* and
                        labels the individual bars. Repeated labels are not de-duplicated
                        and will cause repeated label entries, so this is best used when
                        bars also differ in style (e.g., by passing a list to *color*.)
                    xerr, yerr : float or array-like of shape(N,) or shape(2, N), optional
                        If not *None*, add horizontal / vertical errorbars to the bar tips.
                        The values are +/- sizes relative to the data:
                        - scalar: symmetric +/- values for all bars
                        - shape(N,): symmetric +/- values for each bar
                        - shape(2, N): Separate - and + values for each bar. First row
                          contains the lower errors, the second row contains the upper
                          errors.
                        - *None*: No errorbar. (Default)
                        See :doc:`/gallery/statistics/errorbar_features` for an example on
                        the usage of *xerr* and *yerr*.
                    ecolor : :mpltype:`color` or list of :mpltype:`color`, default: 'black'
                        The line color of the errorbars.
                    capsize : float, default: :rc:`errorbar.capsize`
                       The length of the error bar caps in points.
                    error_kw : dict, optional
                        Dictionary of keyword arguments to be passed to the
                        `~.Axes.errorbar` method. Values of *ecolor* or *capsize* defined
                        here take precedence over the independent keyword arguments.
```

```
1845: (8)                   log : bool, default: False
1846: (12)                      If *True*, set the y-axis to be log scale.
1847: (8)                   data : indexable object, optional
1848: (12)                      DATA_PARAMETER_PLACEHOLDER
1849: (8)                   **kwargs : `.Rectangle` properties
1850: (8)                   %(Rectangle:kwdoc)s
1851: (8)                   See Also
1852: (8)                   --------
1853: (8)                   barh : Plot a horizontal bar plot.
1854: (8)                   Notes
1855: (8)                   -----
1856: (8)                   Stacked bars can be achieved by passing individual *bottom* values per
1857: (8)                   bar. See :doc:`/gallery/lines_bars_and_markers/bar_stacked`.
1858: (8)                   """
1859: (8)                   kwargs = cbook.normalize_kwargs(kwargs, mpatches.Patch)
1860: (8)                   color = kwargs.pop('color', None)
1861: (8)                   if color is None:
1862: (12)                      color = self._get_patches_for_fill.get_next_color()
1863: (8)                   edgecolor = kwargs.pop('edgecolor', None)
1864: (8)                   linewidth = kwargs.pop('linewidth', None)
1865: (8)                   hatch = kwargs.pop('hatch', None)
1866: (8)                   xerr = kwargs.pop('xerr', None)
1867: (8)                   yerr = kwargs.pop('yerr', None)
1868: (8)                   error_kw = kwargs.pop('error_kw', None)
1869: (8)                   error_kw = {} if error_kw is None else error_kw.copy()
1870: (8)                   ezorder = error_kw.pop('zorder', None)
1871: (8)                   if ezorder is None:
1872: (12)                      ezorder = kwargs.get('zorder', None)
1873: (12)                      if ezorder is not None:
1874: (16)                          ezorder += 0.01
1875: (8)                   error_kw.setdefault('zorder', ezorder)
1876: (8)                   ecolor = kwargs.pop('ecolor', 'k')
1877: (8)                   capsize = kwargs.pop('capsize', mpl.rcParams["errorbar.capsize"])
1878: (8)                   error_kw.setdefault('ecolor', ecolor)
1879: (8)                   error_kw.setdefault('capsize', capsize)
1880: (8)                   orientation = kwargs.pop('orientation', 'vertical')
1881: (8)                   _api.check_in_list(['vertical', 'horizontal'],
orientation=orientation)
1882: (8)                   log = kwargs.pop('log', False)
1883: (8)                   label = kwargs.pop('label', '')
1884: (8)                   tick_labels = kwargs.pop('tick_label', None)
1885: (8)                   y = bottom  # Matches barh call signature.
1886: (8)                   if orientation == 'vertical':
1887: (12)                      if y is None:
1888: (16)                          y = 0
1889: (8)                   else:  # horizontal
1890: (12)                      if x is None:
1891: (16)                          x = 0
1892: (8)                   if orientation == 'vertical':
1893: (12)                      self._process_unit_info(
1894: (16)                          [("x", x), ("y", y), ("y", height)], kwargs, convert=False)
1895: (12)                      if log:
1896: (16)                          self.set_yscale('log', nonpositive='clip')
1897: (8)                   else:  # horizontal
1898: (12)                      self._process_unit_info(
1899: (16)                          [("x", x), ("x", width), ("y", y)], kwargs, convert=False)
1900: (12)                      if log:
1901: (16)                          self.set_xscale('log', nonpositive='clip')
1902: (8)                   if self.xaxis is not None:
1903: (12)                      x0 = x
1904: (12)                      x = np.asarray(self.convert_xunits(x))
1905: (12)                      width = self._convert_dx(width, x0, x, self.convert_xunits)
1906: (12)                      if xerr is not None:
1907: (16)                          xerr = self._convert_dx(xerr, x0, x, self.convert_xunits)
1908: (8)                   if self.yaxis is not None:
1909: (12)                      y0 = y
1910: (12)                      y = np.asarray(self.convert_yunits(y))
1911: (12)                      height = self._convert_dx(height, y0, y, self.convert_yunits)
1912: (12)                      if yerr is not None:
```

```
(16)                            yerr = self._convert_dx(yerr, y0, y, self.convert_yunits)
(8)                     x, height, width, y, linewidth, hatch = np.broadcast_arrays(
(12)                        np.atleast_1d(x), height, width, y, linewidth, hatch)
(8)                     if orientation == 'vertical':
(12)                        tick_label_axis = self.xaxis
(12)                        tick_label_position = x
(8)                     else:  # horizontal
(12)                        tick_label_axis = self.yaxis
(12)                        tick_label_position = y
(8)                     if not isinstance(label, str) and np.iterable(label):
(12)                        bar_container_label = '_nolegend_'
(12)                        patch_labels = label
(8)                     else:
(12)                        bar_container_label = label
(12)                        patch_labels = ['_nolegend_'] * len(x)
(8)                     if len(patch_labels) != len(x):
(12)                        raise ValueError(f'number of labels ({len(patch_labels)}) '
(29)                                         f'does not match number of bars ({len(x)}).')
(8)                     linewidth = itertools.cycle(np.atleast_1d(linewidth))
(8)                     hatch = itertools.cycle(np.atleast_1d(hatch))
(8)                     color = itertools.chain(itertools.cycle(mcolors.to_rgba_array(color)),
(32)                                            itertools.repeat('none'))
(8)                     if edgecolor is None:
(12)                        edgecolor = itertools.repeat(None)
(8)                     else:
(12)                        edgecolor = itertools.chain(
(16)                            itertools.cycle(mcolors.to_rgba_array(edgecolor)),
(16)                            itertools.repeat('none'))
(8)                     _api.check_in_list(['center', 'edge'], align=align)
(8)                     if align == 'center':
(12)                        if orientation == 'vertical':
(16)                            try:
(20)                                left = x - width / 2
(16)                            except TypeError as e:
(20)                                raise TypeError(f'the dtypes of parameters x ({x.dtype}) '
(36)                                                f'and width ({width.dtype}) '
(36)                                                f'are incompatible') from e
(16)                            bottom = y
(12)                        else:  # horizontal
(16)                            try:
(20)                                bottom = y - height / 2
(16)                            except TypeError as e:
(20)                                raise TypeError(f'the dtypes of parameters y ({y.dtype}) '
(36)                                                f'and height ({height.dtype}) '
(36)                                                f'are incompatible') from e
(16)                            left = x
(8)                     else:  # edge
(12)                        left = x
(12)                        bottom = y
(8)                     patches = []
(8)                     args = zip(left, bottom, width, height, color, edgecolor, linewidth,
(19)                               hatch, patch_labels)
(8)                     for l, b, w, h, c, e, lw, htch, lbl in args:
(12)                        r = mpatches.Rectangle(
(16)                            xy=(l, b), width=w, height=h,
(16)                            facecolor=c,
(16)                            edgecolor=e,
(16)                            linewidth=lw,
(16)                            label=lbl,
(16)                            hatch=htch,
(16)                            )
(12)                        r._internal_update(kwargs)
(12)                        r.get_path()._interpolation_steps = 100
(12)                        if orientation == 'vertical':
(16)                            r.sticky_edges.y.append(b)
(12)                        else:  # horizontal
(16)                            r.sticky_edges.x.append(l)
(12)                        self.add_patch(r)
(12)                        patches.append(r)
```

```
1982: (8)                    if xerr is not None or yerr is not None:
1983: (12)                       if orientation == 'vertical':
1984: (16)                           ex = [l + 0.5 * w for l, w in zip(left, width)]
1985: (16)                           ey = [b + h for b, h in zip(bottom, height)]
1986: (12)                       else:  # horizontal
1987: (16)                           ex = [l + w for l, w in zip(left, width)]
1988: (16)                           ey = [b + 0.5 * h for b, h in zip(bottom, height)]
1989: (12)                       error_kw.setdefault("label", '_nolegend_')
1990: (12)                       errorbar = self.errorbar(ex, ey, yerr=yerr, xerr=xerr, fmt='none',
1991: (37)                                                **error_kw)
1992: (8)                    else:
1993: (12)                       errorbar = None
1994: (8)                    self._request_autoscale_view()
1995: (8)                    if orientation == 'vertical':
1996: (12)                       datavalues = height
1997: (8)                    else:  # horizontal
1998: (12)                       datavalues = width
1999: (8)                    bar_container = BarContainer(patches, errorbar, datavalues=datavalues,
2000: (37)                                                orientation=orientation,
2001: (37)                                                label=bar_container_label)
2002: (8)                    self.add_container(bar_container)
2003: (8)                    if tick_labels is not None:
2004: (12)                       tick_labels = np.broadcast_to(tick_labels, len(patches))
2005: (12)                       tick_label_axis.set_ticks(tick_label_position)
2006: (12)                       tick_label_axis.set_ticklabels(tick_labels)
2007: (8)                    return bar_container
2008: (4)                @_docstring.dedent_interpd
2009: (4)                def barh(self, y, width, height=0.8, left=None, *, align="center",
2010: (13)                         data=None, **kwargs):
2011: (8)                    r"""
2012: (8)                    Make a horizontal bar plot.
2013: (8)                    The bars are positioned at *y* with the given *align*\ment. Their
2014: (8)                    dimensions are given by *width* and *height*. The horizontal baseline
2015: (8)                    is *left* (default 0).
2016: (8)                    Many parameters can take either a single value applying to all bars
2017: (8)                    or a sequence of values, one for each bar.
2018: (8)                    Parameters
2019: (8)                    ----------
2020: (8)                    y : float or array-like
2021: (12)                       The y coordinates of the bars. See also *align* for the
2022: (12)                       alignment of the bars to the coordinates.
2023: (8)                    width : float or array-like
2024: (12)                       The width(s) of the bars.
2025: (12)                       Note that if *left* has units (e.g. datetime), *width* should be
in
2026: (12)                       units that are a difference from the value of *left* (e.g.
timedelta).
2027: (8)                    height : float or array-like, default: 0.8
2028: (12)                       The heights of the bars.
2029: (12)                       Note that if *y* has units (e.g. datetime), then *height* should
be in
2030: (12)                       units that are a difference (e.g. timedelta) around the *y*
values.
2031: (8)                    left : float or array-like, default: 0
2032: (12)                       The x coordinates of the left side(s) of the bars.
2033: (12)                       Note that if *left* has units, then the x-axis will get a Locator
and
2034: (12)                       Formatter appropriate for the units (e.g. dates, or categorical).
2035: (8)                    align : {'center', 'edge'}, default: 'center'
2036: (12)                       Alignment of the base to the *y* coordinates*:
2037: (12)                       - 'center': Center the bars on the *y* positions.
2038: (12)                       - 'edge': Align the bottom edges of the bars with the *y*
2039: (14)                         positions.
2040: (12)                       To align the bars on the top edge pass a negative *height* and
2041: (12)                       ``align='edge'``.
2042: (8)                    Returns
2043: (8)                    -------
2044: (8)                    `.BarContainer`
2045: (12)                       Container with all the bars and optionally errorbars.
```

```
2046: (8)                    Other Parameters
2047: (8)                    ----------------
2048: (8)                    color : :mpltype:`color` or list of :mpltype:`color`, optional
2049: (12)                       The colors of the bar faces.
2050: (8)                    edgecolor : :mpltype:`color` or list of :mpltype:`color`, optional
2051: (12)                       The colors of the bar edges.
2052: (8)                    linewidth : float or array-like, optional
2053: (12)                       Width of the bar edge(s). If 0, don't draw edges.
2054: (8)                    tick_label : str or list of str, optional
2055: (12)                       The tick labels of the bars.
2056: (12)                       Default: None (Use default numeric labels.)
2057: (8)                    label : str or list of str, optional
2058: (12)                       A single label is attached to the resulting `.BarContainer` as a
2059: (12)                       label for the whole dataset.
2060: (12)                       If a list is provided, it must be the same length as *y* and
2061: (12)                       labels the individual bars. Repeated labels are not de-duplicated
2062: (12)                       and will cause repeated label entries, so this is best used when
2063: (12)                       bars also differ in style (e.g., by passing a list to *color*.)
2064: (8)                    xerr, yerr : float or array-like of shape(N,) or shape(2, N), optional
2065: (12)                       If not *None*, add horizontal / vertical errorbars to the bar
tips.
2066: (12)                       The values are +/- sizes relative to the data:
2067: (12)                       - scalar: symmetric +/- values for all bars
2068: (12)                       - shape(N,): symmetric +/- values for each bar
2069: (12)                       - shape(2, N): Separate - and + values for each bar. First row
2070: (14)                         contains the lower errors, the second row contains the upper
2071: (14)                         errors.
2072: (12)                       - *None*: No errorbar. (default)
2073: (12)                       See :doc:`/gallery/statistics/errorbar_features` for an example on
2074: (12)                       the usage of *xerr* and *yerr*.
2075: (8)                    ecolor : :mpltype:`color` or list of :mpltype:`color`, default:
'black'
2076: (12)                       The line color of the errorbars.
2077: (8)                    capsize : float, default: :rc:`errorbar.capsize`
2078: (11)                      The length of the error bar caps in points.
2079: (8)                    error_kw : dict, optional
2080: (12)                       Dictionary of keyword arguments to be passed to the
2081: (12)                       `~.Axes.errorbar` method. Values of *ecolor* or *capsize* defined
2082: (12)                       here take precedence over the independent keyword arguments.
2083: (8)                    log : bool, default: False
2084: (12)                       If ``True``, set the x-axis to be log scale.
2085: (8)                    data : indexable object, optional
2086: (12)                       If given, all parameters also accept a string ``s``, which is
2087: (12)                       interpreted as ``data[s]`` (unless this raises an exception).
2088: (8)                    **kwargs : `.Rectangle` properties
2089: (8)                    %(Rectangle:kwdoc)s
2090: (8)                    See Also
2091: (8)                    --------
2092: (8)                    bar : Plot a vertical bar plot.
2093: (8)                    Notes
2094: (8)                    -----
2095: (8)                    Stacked bars can be achieved by passing individual *left* values per
2096: (8)                    bar. See
2097: (8)
:doc:`/gallery/lines_bars_and_markers/horizontal_barchart_distribution`.
2098: (8)                    """
2099: (8)                    kwargs.setdefault('orientation', 'horizontal')
2100: (8)                    patches = self.bar(x=left, height=height, width=width, bottom=y,
2101: (27)                                       align=align, data=data, **kwargs)
2102: (8)                    return patches
2103: (4)                def bar_label(self, container, labels=None, *, fmt="%g",
label_type="edge",
2104: (18)                               padding=0, **kwargs):
2105: (8)                    """
2106: (8)                    Label a bar plot.
2107: (8)                    Adds labels to bars in the given `.BarContainer`.
2108: (8)                    You may need to adjust the axis limits to fit the labels.
2109: (8)                    Parameters
2110: (8)                    ----------
```

```
(8)                     container : `.BarContainer`
(12)                        Container with all the bars and optionally errorbars, likely
(12)                        returned from `.bar` or `.barh`.
(8)                     labels : array-like, optional
(12)                        A list of label texts, that should be displayed. If not given, the
(12)                        label texts will be the data values formatted with *fmt*.
(8)                     fmt : str or callable, default: '%g'
(12)                        An unnamed %-style or {}-style format string for the label or a
(12)                        function to call with the value as the first argument.
(12)                        When *fmt* is a string and can be interpreted in both formats,
(12)                        %-style takes precedence over {}-style.
(12)                        .. versionadded:: 3.7
(15)                           Support for {}-style format string and callables.
(8)                     label_type : {'edge', 'center'}, default: 'edge'
(12)                        The label type. Possible values:
(12)                        - 'edge': label placed at the end-point of the bar segment, and
the
(14)                          value displayed will be the position of that end-point.
(12)                        - 'center': label placed in the center of the bar segment, and the
(14)                          value displayed will be the length of that segment.
(14)                          (useful for stacked bars, i.e.,
(14)                          :doc:`/gallery/lines_bars_and_markers/bar_label_demo`)
(8)                     padding : float, default: 0
(12)                        Distance of label from the end of the bar, in points.
(8)                     **kwargs
(12)                        Any remaining keyword arguments are passed through to
(12)                        `.Axes.annotate`. The alignment parameters (
(12)                        *horizontalalignment* / *ha*, *verticalalignment* / *va*) are
(12)                        not supported because the labels are automatically aligned to
(12)                        the bars.
(8)                     Returns
(8)                     -------
(8)                     list of `.Annotation`
(12)                        A list of `.Annotation` instances for the labels.
(8)                     """
(8)                     for key in ['horizontalalignment', 'ha', 'verticalalignment', 'va']:
(12)                        if key in kwargs:
(16)                            raise ValueError(
(20)                                f"Passing {key!r} to bar_label() is not supported.")
(8)                     a, b = self.yaxis.get_view_interval()
(8)                     y_inverted = a > b
(8)                     c, d = self.xaxis.get_view_interval()
(8)                     x_inverted = c > d
(8)                     def sign(x):
(12)                        return 1 if x >= 0 else -1
(8)                     _api.check_in_list(['edge', 'center'], label_type=label_type)
(8)                     bars = container.patches
(8)                     errorbar = container.errorbar
(8)                     datavalues = container.datavalues
(8)                     orientation = container.orientation
(8)                     if errorbar:
(12)                        lines = errorbar.lines  # attribute of "ErrorbarContainer" (tuple)
(12)                        barlinecols = lines[2]  # 0: data_line, 1: caplines, 2:
barlinecols
(12)                        barlinecol = barlinecols[0]  # the "LineCollection" of error bars
(12)                        errs = barlinecol.get_segments()
(8)                     else:
(12)                        errs = []
(8)                     if labels is None:
(12)                        labels = []
(8)                     annotations = []
(8)                     for bar, err, dat, lbl in itertools.zip_longest(
(16)                            bars, errs, datavalues, labels
(8)                     ):
(12)                        (x0, y0), (x1, y1) = bar.get_bbox().get_points()
(12)                        xc, yc = (x0 + x1) / 2, (y0 + y1) / 2
(12)                        if orientation == "vertical":
(16)                            extrema = max(y0, y1) if dat >= 0 else min(y0, y1)
(16)                            length = abs(y0 - y1)
```

```
(12)                        else:  # horizontal
(16)                            extrema = max(x0, x1) if dat >= 0 else min(x0, x1)
(16)                            length = abs(x0 - x1)
(12)                        if err is None or np.size(err) == 0:
(16)                            endpt = extrema
(12)                        elif orientation == "vertical":
(16)                            endpt = err[:, 1].max() if dat >= 0 else err[:, 1].min()
(12)                        else:  # horizontal
(16)                            endpt = err[:, 0].max() if dat >= 0 else err[:, 0].min()
(12)                        if label_type == "center":
(16)                            value = sign(dat) * length
(12)                        else:  # edge
(16)                            value = extrema
(12)                        if label_type == "center":
(16)                            xy = (0.5, 0.5)
(16)                            kwargs["xycoords"] = (
(20)                                lambda r, b=bar:
(24)                                    mtransforms.Bbox.intersection(
(28)                                        b.get_window_extent(r), b.get_clip_box()
(24)                                    ) or mtransforms.Bbox.null()
(16)                            )
(12)                        else:  # edge
(16)                            if orientation == "vertical":
(20)                                xy = xc, endpt
(16)                            else:  # horizontal
(20)                                xy = endpt, yc
(12)                        if orientation == "vertical":
(16)                            y_direction = -1 if y_inverted else 1
(16)                            xytext = 0, y_direction * sign(dat) * padding
(12)                        else:  # horizontal
(16)                            x_direction = -1 if x_inverted else 1
(16)                            xytext = x_direction * sign(dat) * padding, 0
(12)                        if label_type == "center":
(16)                            ha, va = "center", "center"
(12)                        else:  # edge
(16)                            if orientation == "vertical":
(20)                                ha = 'center'
(20)                                if y_inverted:
(24)                                    va = 'top' if dat > 0 else 'bottom'  # also handles
NaN
(20)                                else:
(24)                                    va = 'top' if dat < 0 else 'bottom'  # also handles
NaN
(16)                            else:  # horizontal
(20)                                if x_inverted:
(24)                                    ha = 'right' if dat > 0 else 'left'  # also handles
NaN
(20)                                else:
(24)                                    ha = 'right' if dat < 0 else 'left'  # also handles
NaN
(20)                                va = 'center'
(12)                        if np.isnan(dat):
(16)                            lbl = ''
(12)                        if lbl is None:
(16)                            if isinstance(fmt, str):
(20)                                lbl = cbook._auto_format_str(fmt, value)
(16)                            elif callable(fmt):
(20)                                lbl = fmt(value)
(16)                            else:
(20)                                raise TypeError("fmt must be a str or callable")
(12)                        annotation = self.annotate(lbl,
(39)                                                   xy, xytext, textcoords="offset points",
(39)                                                   ha=ha, va=va, **kwargs)
(12)                        annotations.append(annotation)
(8)                     return annotations
(4)                 @_preprocess_data()
(4)                 @_docstring.dedent_interpd
(4)                 def broken_barh(self, xranges, yrange, **kwargs):
(8)                     """
```

```
2243: (8)                    Plot a horizontal sequence of rectangles.
2244: (8)                    A rectangle is drawn for each element of *xranges*. All rectangles
2245: (8)                    have the same vertical position and size defined by *yrange*.
2246: (8)                    Parameters
2247: (8)                    ----------
2248: (8)                    xranges : sequence of tuples (*xmin*, *xwidth*)
2249: (12)                       The x-positions and extents of the rectangles. For each tuple
2250: (12)                       (*xmin*, *xwidth*) a rectangle is drawn from *xmin* to *xmin* +
2251: (12)                       *xwidth*.
2252: (8)                    yrange : (*ymin*, *yheight*)
2253: (12)                       The y-position and extent for all the rectangles.
2254: (8)                    Returns
2255: (8)                    -------
2256: (8)                    `~.collections.PolyCollection`
2257: (8)                    Other Parameters
2258: (8)                    ----------------
2259: (8)                    data : indexable object, optional
2260: (12)                       DATA_PARAMETER_PLACEHOLDER
2261: (8)                    **kwargs : `.PolyCollection` properties
2262: (12)                       Each *kwarg* can be either a single argument applying to all
2263: (12)                       rectangles, e.g.::
2264: (16)                           facecolors='black'
2265: (12)                       or a sequence of arguments over which is cycled, e.g.::
2266: (16)                           facecolors=('black', 'blue')
2267: (12)                       would create interleaving black and blue rectangles.
2268: (12)                       Supported keywords:
2269: (12)                       %(PolyCollection:kwdoc)s
2270: (8)                    """
2271: (8)                    xdata = cbook._safe_first_finite(xranges) if len(xranges) else None
2272: (8)                    ydata = cbook._safe_first_finite(yrange) if len(yrange) else None
2273: (8)                    self._process_unit_info(
2274: (12)                       [("x", xdata), ("y", ydata)], kwargs, convert=False)
2275: (8)                    vertices = []
2276: (8)                    y0, dy = yrange
2277: (8)                    y0, y1 = self.convert_yunits((y0, y0 + dy))
2278: (8)                    for xr in xranges:  # convert the absolute values, not the x and dx
2279: (12)                       try:
2280: (16)                           x0, dx = xr
2281: (12)                       except Exception:
2282: (16)                           raise ValueError(
2283: (20)                               "each range in xrange must be a sequence with two "
2284: (20)                               "elements (i.e. xrange must be an (N, 2) array)") from
None
2285: (12)                       x0, x1 = self.convert_xunits((x0, x0 + dx))
2286: (12)                       vertices.append([(x0, y0), (x0, y1), (x1, y1), (x1, y0)])
2287: (8)                    col = mcoll.PolyCollection(np.array(vertices), **kwargs)
2288: (8)                    self.add_collection(col, autolim=True)
2289: (8)                    self._request_autoscale_view()
2290: (8)                    return col
2291: (4)                @_preprocess_data()
2292: (4)                def stem(self, *args, linefmt=None, markerfmt=None, basefmt=None,
bottom=0,
2293: (13)                        label=None, orientation='vertical'):
2294: (8)                    """
2295: (8)                    Create a stem plot.
2296: (8)                    A stem plot draws lines perpendicular to a baseline at each location
2297: (8)                    *locs* from the baseline to *heads*, and places a marker there. For
2298: (8)                    vertical stem plots (the default), the *locs* are *x* positions, and
2299: (8)                    the *heads* are *y* values. For horizontal stem plots, the *locs* are
2300: (8)                    *y* positions, and the *heads* are *x* values.
2301: (8)                    Call signature::
2302: (10)                     stem([locs,] heads, linefmt=None, markerfmt=None, basefmt=None)
2303: (8)                    The *locs*-positions are optional. *linefmt* may be provided as
2304: (8)                    positional, but all other formats must be provided as keyword
2305: (8)                    arguments.
2306: (8)                    Parameters
2307: (8)                    ----------
2308: (8)                    locs : array-like, default: (0, 1, ..., len(heads) - 1)
2309: (12)                       For vertical stem plots, the x-positions of the stems.
```

```
            For horizontal stem plots, the y-positions of the stems.
        heads : array-like
            For vertical stem plots, the y-values of the stem heads.
            For horizontal stem plots, the x-values of the stem heads.
        linefmt : str, optional
            A string defining the color and/or linestyle of the vertical lines:

            =========  =============
            Character  Line Style
            =========  =============
            ``'-'``    solid line
            ``'--'``   dashed line
            ``'-.'``   dash-dot line
            ``':'``    dotted line
            =========  =============
            Default: 'C0-', i.e. solid line with the first color of the color
            cycle.
            Note: Markers specified through this parameter (e.g. 'x') will be
            silently ignored. Instead, markers should be specified using
            *markerfmt*.
        markerfmt : str, optional
            A string defining the color and/or shape of the markers at the stem

            heads. If the marker is not given, use the marker 'o', i.e. filled
            circles. If the color is not given, use the color from *linefmt*.
        basefmt : str, default: 'C3-' ('C2-' in classic mode)
            A format string defining the properties of the baseline.
        orientation : {'vertical', 'horizontal'}, default: 'vertical'
            The orientation of the stems.
        bottom : float, default: 0
            The y/x-position of the baseline (depending on *orientation*).
        label : str, optional
            The label to use for the stems in legends.
        data : indexable object, optional
            DATA_PARAMETER_PLACEHOLDER
        Returns
        -------
        `.StemContainer`
            The container may be treated like a tuple
            (*markerline*, *stemlines*, *baseline*)
        Notes
        -----
        .. seealso::
            The MATLAB function
            `stem <https://www.mathworks.com/help/matlab/ref/stem.html>`_
            which inspired this method.
        """
        if not 1 <= len(args) <= 3:
            raise _api.nargs_error('stem', '1-3', len(args))
        _api.check_in_list(['horizontal', 'vertical'], orientation=orientation)
        if len(args) == 1:
            heads, = args
            locs = np.arange(len(heads))
            args = ()
        elif isinstance(args[1], str):
            heads, *args = args
            locs = np.arange(len(heads))
        else:
            locs, heads, *args = args
        if orientation == 'vertical':
            locs, heads = self._process_unit_info([("x", locs), ("y", heads)])
        else:  # horizontal
            heads, locs = self._process_unit_info([("x", heads), ("y", locs)])
        if linefmt is None:
            linefmt = args[0] if len(args) > 0 else "C0-"
        linestyle, linemarker, linecolor = _process_plot_format(linefmt)
        if markerfmt is None:
            markerfmt = "o"
```

```
(8)                 if markerfmt == '':
(12)                    markerfmt = ' '  # = empty line style; '' would resolve rcParams
(8)                 markerstyle, markermarker, markercolor = \
(12)                    _process_plot_format(markerfmt)
(8)                 if markermarker is None:
(12)                    markermarker = 'o'
(8)                 if markerstyle is None:
(12)                    markerstyle = 'None'
(8)                 if markercolor is None:
(12)                    markercolor = linecolor
(8)                 if basefmt is None:
(12)                    basefmt = ("C2-" if mpl.rcParams["_internal.classic_mode"] else
(23)                               "C3-")
(8)                 basestyle, basemarker, basecolor = _process_plot_format(basefmt)
(8)                 if linestyle is None:
(12)                    linestyle = mpl.rcParams['lines.linestyle']
(8)                 xlines = self.vlines if orientation == "vertical" else self.hlines
(8)                 stemlines = xlines(
(12)                    locs, bottom, heads,
(12)                    colors=linecolor, linestyles=linestyle, label="_nolegend_")
(8)                 if orientation == 'horizontal':
(12)                    marker_x = heads
(12)                    marker_y = locs
(12)                    baseline_x = [bottom, bottom]
(12)                    baseline_y = [np.min(locs), np.max(locs)]
(8)                 else:
(12)                    marker_x = locs
(12)                    marker_y = heads
(12)                    baseline_x = [np.min(locs), np.max(locs)]
(12)                    baseline_y = [bottom, bottom]
(8)                 markerline, = self.plot(marker_x, marker_y,
(32)                                        color=markercolor, linestyle=markerstyle,
(32)                                        marker=markermarker, label="_nolegend_")
(8)                 baseline, = self.plot(baseline_x, baseline_y,
(30)                                      color=basecolor, linestyle=basestyle,
(30)                                      marker=basemarker, label="_nolegend_")
(8)                 stem_container = StemContainer((markerline, stemlines, baseline),
(39)                                               label=label)
(8)                 self.add_container(stem_container)
(8)                 return stem_container
(4)             @_preprocess_data(replace_names=["x", "explode", "labels", "colors"])
(4)             def pie(self, x, explode=None, labels=None, colors=None,
(12)                    autopct=None, pctdistance=0.6, shadow=False, labeldistance=1.1,
(12)                    startangle=0, radius=1, counterclock=True,
(12)                    wedgeprops=None, textprops=None, center=(0, 0),
(12)                    frame=False, rotatelabels=False, *, normalize=True, hatch=None):
(8)                 """
(8)                 Plot a pie chart.
(8)                 Make a pie chart of array *x*.  The fractional area of each wedge is
(8)                 given by ``x/sum(x)``.
(8)                 The wedges are plotted counterclockwise, by default starting from the
(8)                 x-axis.
(8)                 Parameters
(8)                 ----------
(8)                 x : 1D array-like
(12)                    The wedge sizes.
(8)                 explode : array-like, default: None
(12)                    If not *None*, is a ``len(x)`` array which specifies the fraction
(12)                    of the radius with which to offset each wedge.
(8)                 labels : list, default: None
(12)                    A sequence of strings providing the labels for each wedge
(8)                 colors : :mpltype:`color` or list of :mpltype:`color`, default: None
(12)                    A sequence of colors through which the pie chart will cycle.  If
(12)                    *None*, will use the colors in the currently active cycle.
(8)                 hatch : str or list, default: None
(12)                    Hatching pattern applied to all pie wedges or sequence of patterns
(12)                    through which the chart will cycle. For a list of valid patterns,
(12)                    see :doc:`/gallery/shapes_and_collections/hatch_style_reference`.
(12)                    .. versionadded:: 3.7
```

```
autopct : None or str or callable, default: None
    If not *None*, *autopct* is a string or function used to label the
    wedges with their numeric value. The label will be placed inside
    the wedge. If *autopct* is a format string, the label will be
    ``fmt % pct``. If *autopct* is a function, then it will be called.
pctdistance : float, default: 0.6
    The relative distance along the radius at which the text
    generated by *autopct* is drawn. To draw the text outside the pie,
    set *pctdistance* > 1. This parameter is ignored if *autopct* is
    ``None``.
labeldistance : float or None, default: 1.1
    The relative distance along the radius at which the labels are
    drawn. To draw the labels inside the pie, set  *labeldistance* < 1.

    If set to ``None``, labels are not drawn but are still stored for
    use in `.legend`.
shadow : bool or dict, default: False
    If bool, whether to draw a shadow beneath the pie. If dict, draw a shadow

    passing the properties in the dict to `.Shadow`.
    .. versionadded:: 3.8
        *shadow* can be a dict.
startangle : float, default: 0 degrees
    The angle by which the start of the pie is rotated,
    counterclockwise from the x-axis.
radius : float, default: 1
    The radius of the pie.
counterclock : bool, default: True
    Specify fractions direction, clockwise or counterclockwise.
wedgeprops : dict, default: None
    Dict of arguments passed to each `.patches.Wedge` of the pie.
    For example, ``wedgeprops = {'linewidth': 3}`` sets the width of
    the wedge border lines equal to 3. By default, ``clip_on=False``.
    When there is a conflict between these properties and other
    keywords, properties passed to *wedgeprops* take precedence.
textprops : dict, default: None
    Dict of arguments to pass to the text objects.
center : (float, float), default: (0, 0)
    The coordinates of the center of the chart.
frame : bool, default: False
    Plot Axes frame with the chart if true.
rotatelabels : bool, default: False
    Rotate each label to the angle of the corresponding slice if true.
normalize : bool, default: True
    When *True*, always make a full pie by normalizing x so that
    ``sum(x) == 1``. *False* makes a partial pie if ``sum(x) <= 1``
    and raises a `ValueError` for ``sum(x) > 1``.
data : indexable object, optional
    DATA_PARAMETER_PLACEHOLDER
Returns
-------
patches : list
    A sequence of `matplotlib.patches.Wedge` instances
texts : list
    A list of the label `.Text` instances.
autotexts : list
    A list of `.Text` instances for the numeric labels. This will only
    be returned if the parameter *autopct* is not *None*.
Notes
-----
The pie chart will probably look best if the figure and Axes are
square, or the Axes aspect is equal.
This method sets the aspect ratio of the axis to "equal".
The Axes aspect ratio can be controlled with `.Axes.set_aspect`.
"""
self.set_aspect('equal')
x = np.asarray(x, np.float32)
if x.ndim > 1:
    raise ValueError("x must be 1D")
```

```
(8)                     if np.any(x < 0):
(12)                        raise ValueError("Wedge sizes 'x' must be non negative values")
(8)                     sx = x.sum()
(8)                     if normalize:
(12)                        x = x / sx
(8)                     elif sx > 1:
(12)                        raise ValueError('Cannot plot an unnormalized pie with sum(x) > 1')
(8)                     if labels is None:
(12)                        labels = [''] * len(x)
(8)                     if explode is None:
(12)                        explode = [0] * len(x)
(8)                     if len(x) != len(labels):
(12)                        raise ValueError("'label' must be of length 'x'")
(8)                     if len(x) != len(explode):
(12)                        raise ValueError("'explode' must be of length 'x'")
(8)                     if colors is None:
(12)                        get_next_color = self._get_patches_for_fill.get_next_color
(8)                     else:
(12)                        color_cycle = itertools.cycle(colors)
(12)                        def get_next_color():
(16)                            return next(color_cycle)
(8)                     hatch_cycle = itertools.cycle(np.atleast_1d(hatch))
(8)                     _api.check_isinstance(Real, radius=radius, startangle=startangle)
(8)                     if radius <= 0:
(12)                        raise ValueError(f'radius must be a positive number, not {radius}')
(8)                     theta1 = startangle / 360
(8)                     if wedgeprops is None:
(12)                        wedgeprops = {}
(8)                     if textprops is None:
(12)                        textprops = {}
(8)                     texts = []
(8)                     slices = []
(8)                     autotexts = []
(8)                     for frac, label, expl in zip(x, labels, explode):
(12)                        x, y = center
(12)                        theta2 = (theta1 + frac) if counterclock else (theta1 - frac)
(12)                        thetam = 2 * np.pi * 0.5 * (theta1 + theta2)
(12)                        x += expl * math.cos(thetam)
(12)                        y += expl * math.sin(thetam)
(12)                        w = mpatches.Wedge((x, y), radius, 360. * min(theta1, theta2),
(31)                                           360. * max(theta1, theta2),
(31)                                           facecolor=get_next_color(),
(31)                                           hatch=next(hatch_cycle),
(31)                                           clip_on=False,
(31)                                           label=label)
(12)                        w.set(**wedgeprops)
(12)                        slices.append(w)
(12)                        self.add_patch(w)
(12)                        if shadow:
(16)                            shadow_dict = {'ox': -0.02, 'oy': -0.02, 'label': '_nolegend_'}
(16)                            if isinstance(shadow, dict):
(20)                                shadow_dict.update(shadow)
(16)                            self.add_patch(mpatches.Shadow(w, **shadow_dict))
(12)                        if labeldistance is not None:
(16)                            xt = x + labeldistance * radius * math.cos(thetam)
(16)                            yt = y + labeldistance * radius * math.sin(thetam)
(16)                            label_alignment_h = 'left' if xt > 0 else 'right'
(16)                            label_alignment_v = 'center'
(16)                            label_rotation = 'horizontal'
(16)                            if rotatelabels:
(20)                                label_alignment_v = 'bottom' if yt > 0 else 'top'
(20)                                label_rotation = (np.rad2deg(thetam)
(38)                                                  + (0 if xt > 0 else 180))
(16)                            t = self.text(xt, yt, label,
(30)                                          clip_on=False,
(30)                                          horizontalalignment=label_alignment_h,
```

```
2578: (30)                                      verticalalignment=label_alignment_v,
2579: (30)                                      rotation=label_rotation,
2580: (30)                                      size=mpl.rcParams['xtick.labelsize'])
2581: (16)                          t.set(**textprops)
2582: (16)                          texts.append(t)
2583: (12)                      if autopct is not None:
2584: (16)                          xt = x + pctdistance * radius * math.cos(thetam)
2585: (16)                          yt = y + pctdistance * radius * math.sin(thetam)
2586: (16)                          if isinstance(autopct, str):
2587: (20)                              s = autopct % (100. * frac)
2588: (16)                          elif callable(autopct):
2589: (20)                              s = autopct(100. * frac)
2590: (16)                          else:
2591: (20)                              raise TypeError(
2592: (24)                                  'autopct must be callable or a format string')
2593: (16)                          if mpl._val_or_rc(textprops.get("usetex"), "text.usetex"):
2594: (20)                              s = re.sub(r"([^\\])%", r"\1\\%", s)
2595: (16)                          t = self.text(xt, yt, s,
2596: (30)                                      clip_on=False,
2597: (30)                                      horizontalalignment='center',
2598: (30)                                      verticalalignment='center')
2599: (16)                          t.set(**textprops)
2600: (16)                          autotexts.append(t)
2601: (12)                      theta1 = theta2
2602: (8)                   if frame:
2603: (12)                      self._request_autoscale_view()
2604: (8)                   else:
2605: (12)                      self.set(frame_on=False, xticks=[], yticks=[],
2606: (21)                               xlim=(-1.25 + center[0], 1.25 + center[0]),
2607: (21)                               ylim=(-1.25 + center[1], 1.25 + center[1]))
2608: (8)                   if autopct is None:
2609: (12)                      return slices, texts
2610: (8)                   else:
2611: (12)                      return slices, texts, autotexts
2612: (4)               @staticmethod
2613: (4)               def _errorevery_to_mask(x, errorevery):
2614: (8)                   """
2615: (8)                   Normalize `errorbar`'s *errorevery* to be a boolean mask for data *x*.
2616: (8)                   This function is split out to be usable both by 2D and 3D errorbars.
2617: (8)                   """
2618: (8)                   if isinstance(errorevery, Integral):
2619: (12)                      errorevery = (0, errorevery)
2620: (8)                   if isinstance(errorevery, tuple):
2621: (12)                      if (len(errorevery) == 2 and
2622: (20)                              isinstance(errorevery[0], Integral) and
2623: (20)                              isinstance(errorevery[1], Integral)):
2624: (16)                          errorevery = slice(errorevery[0], None, errorevery[1])
2625: (12)                      else:
2626: (16)                          raise ValueError(
2627: (20)                              f'{errorevery=!r} is a not a tuple of two integers')
2628: (8)                   elif isinstance(errorevery, slice):
2629: (12)                      pass
2630: (8)                   elif not isinstance(errorevery, str) and np.iterable(errorevery):
2631: (12)                      try:
2632: (16)                          x[errorevery]  # fancy indexing
2633: (12)                      except (ValueError, IndexError) as err:
2634: (16)                          raise ValueError(
2635: (20)                              f"{errorevery=!r} is iterable but not a valid NumPy fancy
"
2636: (20)                              "index to match 'xerr'/'yerr'") from err
2637: (8)                   else:
2638: (12)                      raise ValueError(f"{errorevery=!r} is not a recognized value")
2639: (8)                   everymask = np.zeros(len(x), bool)
2640: (8)                   everymask[errorevery] = True
2641: (8)                   return everymask
2642: (4)               @_preprocess_data(replace_names=["x", "y", "xerr", "yerr"],
2643: (22)                                label_namer="y")
2644: (4)               @_docstring.dedent_interpd
2645: (4)               def errorbar(self, x, y, yerr=None, xerr=None,
```

```
 2646: (17)                         fmt='', ecolor=None, elinewidth=None, capsize=None,
 2647: (17)                         barsabove=False, lolims=False, uplims=False,
 2648: (17)                         xlolims=False, xuplims=False, errorevery=1, capthick=None,
 2649: (17)                         **kwargs):
 2650: (8)                  """
 2651: (8)                  Plot y versus x as lines and/or markers with attached errorbars.
 2652: (8)                  *x*, *y* define the data locations, *xerr*, *yerr* define the errorbar
 2653: (8)                  sizes. By default, this draws the data markers/lines as well as the
 2654: (8)                  errorbars. Use fmt='none' to draw errorbars without any data markers.
 2655: (8)                  .. versionadded:: 3.7
 2656: (11)                    Caps and error lines are drawn in polar coordinates on polar plots.
 2657: (8)                  Parameters
 2658: (8)                  ----------
 2659: (8)                  x, y : float or array-like
 2660: (12)                     The data positions.
 2661: (8)                  xerr, yerr : float or array-like, shape(N,) or shape(2, N), optional
 2662: (12)                     The errorbar sizes:
 2663: (12)                     - scalar: Symmetric +/- values for all data points.
 2664: (12)                     - shape(N,): Symmetric +/-values for each data point.
 2665: (12)                     - shape(2, N): Separate - and + values for each bar. First row
 2666: (14)                       contains the lower errors, the second row contains the upper
 2667: (14)                       errors.
 2668: (12)                     - *None*: No errorbar.
 2669: (12)                     All values must be >= 0.
 2670: (12)                     See :doc:`/gallery/statistics/errorbar_features`
 2671: (12)                     for an example on the usage of ``xerr`` and ``yerr``.
 2672: (8)                  fmt : str, default: ''
 2673: (12)                     The format for the data points / data lines. See `.plot` for
 2674: (12)                     details.
 2675: (12)                     Use 'none' (case-insensitive) to plot errorbars without any data
 2676: (12)                     markers.
 2677: (8)                  ecolor : :mpltype:`color`, default: None
 2678: (12)                     The color of the errorbar lines.  If None, use the color of the
 2679: (12)                     line connecting the markers.
 2680: (8)                  elinewidth : float, default: None
 2681: (12)                     The linewidth of the errorbar lines. If None, the linewidth of
 2682: (12)                     the current style is used.
 2683: (8)                  capsize : float, default: :rc:`errorbar.capsize`
 2684: (12)                     The length of the error bar caps in points.
 2685: (8)                  capthick : float, default: None
 2686: (12)                     An alias to the keyword argument *markeredgewidth* (a.k.a. *mew*).
 2687: (12)                     This setting is a more sensible name for the property that
 2688: (12)                     controls the thickness of the error bar cap in points. For
 2689: (12)                     backwards compatibility, if *mew* or *markeredgewidth* are given,
 2690: (12)                     then they will over-ride *capthick*. This may change in future
 2691: (12)                     releases.
 2692: (8)                  barsabove : bool, default: False
 2693: (12)                     If True, will plot the errorbars above the plot
 2694: (12)                     symbols. Default is below.
 2695: (8)                  lolims, uplims, xlolims, xuplims : bool or array-like, default: False
 2696: (12)                     These arguments can be used to indicate that a value gives only
 2697: (12)                     upper/lower limits.  In that case a caret symbol is used to
 2698: (12)                     indicate this. *lims*-arguments may be scalars, or array-likes of
 2699: (12)                     the same length as *xerr* and *yerr*.  To use limits with inverted
 2700: (12)                     axes, `~.Axes.set_xlim` or `~.Axes.set_ylim` must be called before
 2701: (12)                     :meth:`errorbar`.  Note the tricky parameter names: setting e.g.
 2702: (12)                     *lolims* to True means that the y-value is a *lower* limit of the
 2703: (12)                     True value, so, only an *upward*-pointing arrow will be drawn!
 2704: (8)                  errorevery : int or (int, int), default: 1
 2705: (12)                     draws error bars on a subset of the data. *errorevery* =N draws
 2706: (12)                     error bars on the points (x[::N], y[::N]).
 2707: (12)                     *errorevery* =(start, N) draws error bars on the points
 2708: (12)                     (x[start::N], y[start::N]). e.g. errorevery=(6, 3)
 2709: (12)                     adds error bars to the data at (x[6], x[9], x[12], x[15], ...).
 2710: (12)                     Used to avoid overlapping error bars when two series share x-axis
 2711: (12)                     values.
 2712: (8)                  Returns
 2713: (8)                  -------
 2714: (8)                  `.ErrorbarContainer`
```

```
(12)                        The container contains:
(12)                        - data_line : A `~matplotlib.lines.Line2D` instance of x, y plot
markers
(14)                          and/or line.
(12)                        - caplines : A tuple of `~matplotlib.lines.Line2D` instances of
the error
(14)                          bar caps.
(12)                        - barlinecols : A tuple of `.LineCollection` with the horizontal
and
(14)                          vertical error ranges.
(8)                     Other Parameters
(8)                     ----------------
(8)                     data : indexable object, optional
(12)                        DATA_PARAMETER_PLACEHOLDER
(8)                     **kwargs
(12)                        All other keyword arguments are passed on to the `~.Axes.plot`
call
(12)                        drawing the markers. For example, this code makes big red squares
(12)                        with thick green edges::
(16)                            x, y, yerr = rand(3, 10)
(16)                            errorbar(x, y, yerr, marker='s', mfc='red',
(25)                                     mec='green', ms=20, mew=4)
(12)                        where *mfc*, *mec*, *ms* and *mew* are aliases for the longer
(12)                        property names, *markerfacecolor*, *markeredgecolor*, *markersize*
(12)                        and *markeredgewidth*.
(12)                        Valid kwargs for the marker properties are:
(12)                        - *dashes*
(12)                        - *dash_capstyle*
(12)                        - *dash_joinstyle*
(12)                        - *drawstyle*
(12)                        - *fillstyle*
(12)                        - *linestyle*
(12)                        - *marker*
(12)                        - *markeredgecolor*
(12)                        - *markeredgewidth*
(12)                        - *markerfacecolor*
(12)                        - *markerfacecoloralt*
(12)                        - *markersize*
(12)                        - *markevery*
(12)                        - *solid_capstyle*
(12)                        - *solid_joinstyle*
(12)                        Refer to the corresponding `.Line2D` property for more details:
(12)                        %(Line2D:kwdoc)s
(8)                     """
(8)                     kwargs = cbook.normalize_kwargs(kwargs, mlines.Line2D)
(8)                     kwargs = {k: v for k, v in kwargs.items() if v is not None}
(8)                     kwargs.setdefault('zorder', 2)
(8)                     if not isinstance(x, np.ndarray):
(12)                        x = np.asarray(x, dtype=object)
(8)                     if not isinstance(y, np.ndarray):
(12)                        y = np.asarray(y, dtype=object)
(8)                     def _upcast_err(err):
(12)                        """
(12)                        Safely handle tuple of containers that carry units.
(12)                        This function covers the case where the input to the xerr/yerr is
a
(12)                        length 2 tuple of equal length ndarray-subclasses that carry the
(12)                        unit information in the container.
(12)                        If we have a tuple of nested numpy array (subclasses), we defer
(12)                        coercing the units to be consistent to the underlying unit
(12)                        library (and implicitly the broadcasting).
(12)                        Otherwise, fallback to casting to an object array.
(12)                        """
(12)                        if (
(20)                                np.iterable(err) and
(20)                                len(err) > 0 and
(20)                                isinstance(cbook._safe_first_finite(err), np.ndarray)
(12)                        ):
(16)                            atype = type(cbook._safe_first_finite(err))
```

```
2779: (16)                          if atype is np.ndarray:
2780: (20)                              return np.asarray(err, dtype=object)
2781: (16)                          return atype(err)
2782: (12)                      return np.asarray(err, dtype=object)
2783: (8)                   if xerr is not None and not isinstance(xerr, np.ndarray):
2784: (12)                      xerr = _upcast_err(xerr)
2785: (8)                   if yerr is not None and not isinstance(yerr, np.ndarray):
2786: (12)                      yerr = _upcast_err(yerr)
2787: (8)                   x, y = np.atleast_1d(x, y)  # Make sure all the args are iterable.
2788: (8)                   if len(x) != len(y):
2789: (12)                      raise ValueError("'x' and 'y' must have the same size")
2790: (8)                   everymask = self._errorevery_to_mask(x, errorevery)
2791: (8)                   label = kwargs.pop("label", None)
2792: (8)                   kwargs['label'] = '_nolegend_'
2793: (8)                   (data_line, base_style), = self._get_lines._plot_args(
2794: (12)                      self, (x, y) if fmt == '' else (x, y, fmt), kwargs,
return_kwargs=True)
2795: (8)                   if barsabove:
2796: (12)                      data_line.set_zorder(kwargs['zorder'] - .1)
2797: (8)                   else:
2798: (12)                      data_line.set_zorder(kwargs['zorder'] + .1)
2799: (8)                   if fmt.lower() != 'none':
2800: (12)                      self.add_line(data_line)
2801: (8)                   else:
2802: (12)                      data_line = None
2803: (12)                      base_style.pop('color')
2804: (12)                      if 'color' in kwargs:
2805: (16)                          base_style['color'] = kwargs.pop('color')
2806: (8)                   if 'color' not in base_style:
2807: (12)                      base_style['color'] = 'C0'
2808: (8)                   if ecolor is None:
2809: (12)                      ecolor = base_style['color']
2810: (8)                   for key in ['marker', 'markersize', 'markerfacecolor',
2811: (20)                              'markerfacecoloralt',
2812: (20)                              'markeredgewidth', 'markeredgecolor', 'markevery',
2813: (20)                              'linestyle', 'fillstyle', 'drawstyle', 'dash_capstyle',
2814: (20)                              'dash_joinstyle', 'solid_capstyle', 'solid_joinstyle',
2815: (20)                              'dashes']:
2816: (12)                      base_style.pop(key, None)
2817: (8)                   eb_lines_style = {**base_style, 'color': ecolor}
2818: (8)                   if elinewidth is not None:
2819: (12)                      eb_lines_style['linewidth'] = elinewidth
2820: (8)                   elif 'linewidth' in kwargs:
2821: (12)                      eb_lines_style['linewidth'] = kwargs['linewidth']
2822: (8)                   for key in ('transform', 'alpha', 'zorder', 'rasterized'):
2823: (12)                      if key in kwargs:
2824: (16)                          eb_lines_style[key] = kwargs[key]
2825: (8)                   eb_cap_style = {**base_style, 'linestyle': 'none'}
2826: (8)                   if capsize is None:
2827: (12)                      capsize = mpl.rcParams["errorbar.capsize"]
2828: (8)                   if capsize > 0:
2829: (12)                      eb_cap_style['markersize'] = 2. * capsize
2830: (8)                   if capthick is not None:
2831: (12)                      eb_cap_style['markeredgewidth'] = capthick
2832: (8)                   for key in ('markeredgewidth', 'transform', 'alpha',
2833: (20)                              'zorder', 'rasterized'):
2834: (12)                      if key in kwargs:
2835: (16)                          eb_cap_style[key] = kwargs[key]
2836: (8)                   eb_cap_style['color'] = ecolor
2837: (8)                   barcols = []
2838: (8)                   caplines = {'x': [], 'y': []}
2839: (8)                   def apply_mask(arrays, mask):
2840: (12)                      return [array[mask] for array in arrays]
2841: (8)                   for (dep_axis, dep, err, lolims, uplims, indep, lines_func,
2842: (13)                       marker, lomarker, himarker) in [
2843: (16)                          ("x", x, xerr, xlolims, xuplims, y, self.hlines,
2844: (17)                           "|", mlines.CARETRIGHTBASE, mlines.CARETLEFTBASE),
2845: (16)                          ("y", y, yerr, lolims, uplims, x, self.vlines,
2846: (17)                           "_", mlines.CARETUPBASE, mlines.CARETDOWNBASE),
```

```
(8)                     ]:
(12)                        if err is None:
(16)                            continue
(12)                        lolims = np.broadcast_to(lolims, len(dep)).astype(bool)
(12)                        uplims = np.broadcast_to(uplims, len(dep)).astype(bool)
(12)                        try:
(16)                            np.broadcast_to(err, (2, len(dep)))
(12)                        except ValueError:
(16)                            raise ValueError(
(20)                                f"'{dep_axis}err' (shape: {np.shape(err)}) must be a "
(20)                                f"scalar or a 1D or (2, n) array-like whose shape matches
"
(20)                                f"'{dep_axis}' (shape: {np.shape(dep)})") from None
(12)                        if err.dtype is np.dtype(object) and np.any(err == None):  # noqa:
E711
(16)                            raise ValueError(
(20)                                f"'{dep_axis}err' must not contain None. "
(20)                                "Use NaN if you want to skip a value.")
(12)                        res = np.zeros(err.shape, dtype=bool)  # Default in case of nan
(12)                        if np.any(np.less(err, -err, out=res, where=(err == err))):
(16)                            raise ValueError(
(20)                                f"'{dep_axis}err' must not contain negative values")
(12)                        low, high = dep + np.vstack([-(1 - lolims), 1 - uplims]) * err
(12)                        barcols.append(lines_func(
(16)                            *apply_mask([indep, low, high], everymask), **eb_lines_style))
(12)                        if self.name == "polar" and dep_axis == "x":
(16)                            for b in barcols:
(20)                                for p in b.get_paths():
(24)                                    p._interpolation_steps = 2
(12)                        nolims = ~(lolims | uplims)
(12)                        if nolims.any() and capsize > 0:
(16)                            indep_masked, lo_masked, hi_masked = apply_mask(
(20)                                [indep, low, high], nolims & everymask)
(16)                            for lh_masked in [lo_masked, hi_masked]:
(20)                                line = mlines.Line2D(indep_masked, indep_masked,
(41)                                                     marker=marker, **eb_cap_style)
(20)                                line.set(**{f"{dep_axis}data": lh_masked})
(20)                                caplines[dep_axis].append(line)
(12)                        for idx, (lims, hl) in enumerate([(lolims, high), (uplims, low)]):
(16)                            if not lims.any():
(20)                                continue
(16)                            hlmarker = (
(20)                                himarker
(20)                                if self._axis_map[dep_axis].get_inverted() ^ idx
(20)                                else lomarker)
(16)                            x_masked, y_masked, hl_masked = apply_mask(
(20)                                [x, y, hl], lims & everymask)
(16)                            line = mlines.Line2D(x_masked, y_masked,
(37)                                                 marker=hlmarker, **eb_cap_style)
(16)                            line.set(**{f"{dep_axis}data": hl_masked})
(16)                            caplines[dep_axis].append(line)
(16)                            if capsize > 0:
(20)                                caplines[dep_axis].append(mlines.Line2D(
(24)                                    x_masked, y_masked, marker=marker, **eb_cap_style))
(8)                     if self.name == 'polar':
(12)                        for axis in caplines:
(16)                            for l in caplines[axis]:
(20)                                for theta, r in zip(l.get_xdata(), l.get_ydata()):
(24)                                    rotation = mtransforms.Affine2D().rotate(theta)
(24)                                    if axis == 'y':
(28)                                        rotation.rotate(-np.pi / 2)
(24)                                    ms = mmarkers.MarkerStyle(marker=marker,
(50)                                                              transform=rotation)
(24)                                    self.add_line(mlines.Line2D([theta], [r], marker=ms,
(52)                                                                **eb_cap_style))
(8)                     else:
(12)                        for axis in caplines:
(16)                            for l in caplines[axis]:
(20)                                self.add_line(l)
```

```
(8)                 self._request_autoscale_view()
(8)                 caplines = caplines['x'] + caplines['y']
(8)                 errorbar_container = ErrorbarContainer(
(12)                    (data_line, tuple(caplines), tuple(barcols)),
(12)                    has_xerr=(xerr is not None), has_yerr=(yerr is not None),
(12)                    label=label)
(8)                 self.containers.append(errorbar_container)
(8)                 return errorbar_container  # (l0, caplines, barcols)
(4)             @_preprocess_data()
(4)             @_api.rename_parameter("3.9", "labels", "tick_labels")
(4)             def boxplot(self, x, notch=None, sym=None, vert=None, whis=None,
(16)                        positions=None, widths=None, patch_artist=None,
(16)                        bootstrap=None, usermedians=None, conf_intervals=None,
(16)                        meanline=None, showmeans=None, showcaps=None,
(16)                        showbox=None, showfliers=None, boxprops=None,
(16)                        tick_labels=None, flierprops=None, medianprops=None,
(16)                        meanprops=None, capprops=None, whiskerprops=None,
(16)                        manage_ticks=True, autorange=False, zorder=None,
(16)                        capwidths=None, label=None):
(8)                 """
(8)                 Draw a box and whisker plot.
(8)                 The box extends from the first quartile (Q1) to the third
(8)                 quartile (Q3) of the data, with a line at the median.
(8)                 The whiskers extend from the box to the farthest data point
(8)                 lying within 1.5x the inter-quartile range (IQR) from the box.
(8)                 Flier points are those past the end of the whiskers.
(8)                 See https://en.wikipedia.org/wiki/Box_plot for reference.
(8)                 .. code-block:: none
(18)                          Q1-1.5IQR   Q1   median  Q3   Q3+1.5IQR
(31)                                       |-----:-----|
(15)                       o      |--------|     :     |--------|    o  o
(31)                                       |-----:-----|
(13)                     flier             <----------->            fliers
(36)                                            IQR
(8)                 Parameters
(8)                 ----------
(8)                 x : Array or a sequence of vectors.
(12)                    The input data.  If a 2D array, a boxplot is drawn for each column
(12)                    in *x*.  If a sequence of 1D arrays, a boxplot is drawn for each
(12)                    array in *x*.
(8)                 notch : bool, default: :rc:`boxplot.notch`
(12)                    Whether to draw a notched boxplot (`True`), or a rectangular
(12)                    boxplot (`False`).  The notches represent the confidence interval
(12)                    (CI) around the median.  The documentation for *bootstrap*
(12)                    describes how the locations of the notches are computed by
(12)                    default, but their locations may also be overridden by setting the
(12)                    *conf_intervals* parameter.
(12)                    .. note::
(16)                        In cases where the values of the CI are less than the
(16)                        lower quartile or greater than the upper quartile, the
(16)                        notches will extend beyond the box, giving it a
(16)                        distinctive "flipped" appearance. This is expected
(16)                        behavior and consistent with other statistical
(16)                        visualization packages.
(8)                 sym : str, optional
(12)                    The default symbol for flier points.  An empty string ('') hides
(12)                    the fliers.  If `None`, then the fliers default to 'b+'.  More
(12)                    control is provided by the *flierprops* parameter.
(8)                 vert : bool, default: :rc:`boxplot.vertical`
(12)                    If `True`, draws vertical boxes.
(12)                    If `False`, draw horizontal boxes.
(8)                 whis : float or (float, float), default: 1.5
(12)                    The position of the whiskers.
(12)                    If a float, the lower whisker is at the lowest datum above
(12)                    ``Q1 - whis*(Q3-Q1)``, and the upper whisker at the highest datum
(12)                    below ``Q3 + whis*(Q3-Q1)``, where Q1 and Q3 are the first and
(12)                    third quartiles.  The default value of ``whis = 1.5`` corresponds
(12)                    to Tukey's original definition of boxplots.
(12)                    If a pair of floats, they indicate the percentiles at which to
```

```
(12)                        draw the whiskers (e.g., (5, 95)).  In particular, setting this to
(12)                        (0, 100) results in whiskers covering the whole range of the data.
(12)                        In the edge case where ``Q1 == Q3``, *whis* is automatically set
(12)                        to (0, 100) (cover the whole range of the data) if *autorange* is
(12)                        True.
(12)                        Beyond the whiskers, data are considered outliers and are plotted
(12)                        as individual points.
(8)                     bootstrap : int, optional
(12)                        Specifies whether to bootstrap the confidence intervals
(12)                        around the median for notched boxplots. If *bootstrap* is
(12)                        None, no bootstrapping is performed, and notches are
(12)                        calculated using a Gaussian-based asymptotic approximation
(12)                        (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and
(12)                        Kendall and Stuart, 1967). Otherwise, bootstrap specifies
(12)                        the number of times to bootstrap the median to determine its
(12)                        95% confidence intervals. Values between 1000 and 10000 are
(12)                        recommended.
(8)                     usermedians : 1D array-like, optional
(12)                        A 1D array-like of length ``len(x)``.  Each entry that is not
(12)                        `None` forces the value of the median for the corresponding
(12)                        dataset.  For entries that are `None`, the medians are computed
(12)                        by Matplotlib as normal.
(8)                     conf_intervals : array-like, optional
(12)                        A 2D array-like of shape ``(len(x), 2)``.  Each entry that is not
(12)                        None forces the location of the corresponding notch (which is
(12)                        only drawn if *notch* is `True`).  For entries that are `None`,
(12)                        the notches are computed by the method specified by the other
(12)                        parameters (e.g., *bootstrap*).
(8)                     positions : array-like, optional
(12)                        The positions of the boxes. The ticks and limits are
(12)                        automatically set to match the positions. Defaults to
(12)                        ``range(1, N+1)`` where N is the number of boxes to be drawn.
(8)                     widths : float or array-like
(12)                        The widths of the boxes.  The default is 0.5, or ``0.15*(distance
(12)                        between extreme positions)``, if that is smaller.
(8)                     patch_artist : bool, default: :rc:`boxplot.patchartist`
(12)                        If `False` produces boxes with the Line2D artist. Otherwise,
(12)                        boxes are drawn with Patch artists.
(8)                     tick_labels : list of str, optional
(12)                        The tick labels of each boxplot.
(12)                        Ticks are always placed at the box *positions*. If *tick_labels*
is given,
(12)                        the ticks are labelled accordingly. Otherwise, they keep their
numeric
(12)                        values.
(12)                        .. versionchanged:: 3.9
(16)                            Renamed from *labels*, which is deprecated since 3.9
(16)                            and will be removed in 3.11.
(8)                     manage_ticks : bool, default: True
(12)                        If True, the tick locations and labels will be adjusted to match
(12)                        the boxplot positions.
(8)                     autorange : bool, default: False
(12)                        When `True` and the data are distributed such that the 25th and
(12)                        75th percentiles are equal, *whis* is set to (0, 100) such
(12)                        that the whisker ends are at the minimum and maximum of the data.
(8)                     meanline : bool, default: :rc:`boxplot.meanline`
(12)                        If `True` (and *showmeans* is `True`), will try to render the
(12)                        mean as a line spanning the full width of the box according to
(12)                        *meanprops* (see below).  Not recommended if *shownotches* is also
(12)                        True.  Otherwise, means will be shown as points.
(8)                     zorder : float, default: ``Line2D.zorder = 2``
(12)                        The zorder of the boxplot.
(8)                     Returns
(8)                     -------
(8)                     dict
(10)                      A dictionary mapping each component of the boxplot to a list
(10)                      of the `.Line2D` instances created. That dictionary has the
(10)                      following keys (assuming vertical boxplots):
(10)                      - ``boxes``: the main body of the boxplot showing the
```

```
3050: (12)                      quartiles and the median's confidence intervals if
3051: (12)                      enabled.
3052: (10)                    - ``medians``: horizontal lines at the median of each box.
3053: (10)                    - ``whiskers``: the vertical lines extending to the most
3054: (12)                      extreme, non-outlier data points.
3055: (10)                    - ``caps``: the horizontal lines at the ends of the
3056: (12)                      whiskers.
3057: (10)                    - ``fliers``: points representing data that extend beyond
3058: (12)                      the whiskers (fliers).
3059: (10)                    - ``means``: points or lines representing the means.
3060: (8)                   Other Parameters
3061: (8)                   ----------------
3062: (8)                   showcaps : bool, default: :rc:`boxplot.showcaps`
3063: (12)                      Show the caps on the ends of whiskers.
3064: (8)                   showbox : bool, default: :rc:`boxplot.showbox`
3065: (12)                      Show the central box.
3066: (8)                   showfliers : bool, default: :rc:`boxplot.showfliers`
3067: (12)                      Show the outliers beyond the caps.
3068: (8)                   showmeans : bool, default: :rc:`boxplot.showmeans`
3069: (12)                      Show the arithmetic means.
3070: (8)                   capprops : dict, default: None
3071: (12)                      The style of the caps.
3072: (8)                   capwidths : float or array, default: None
3073: (12)                      The widths of the caps.
3074: (8)                   boxprops : dict, default: None
3075: (12)                      The style of the box.
3076: (8)                   whiskerprops : dict, default: None
3077: (12)                      The style of the whiskers.
3078: (8)                   flierprops : dict, default: None
3079: (12)                      The style of the fliers.
3080: (8)                   medianprops : dict, default: None
3081: (12)                      The style of the median.
3082: (8)                   meanprops : dict, default: None
3083: (12)                      The style of the mean.
3084: (8)                   label : str or list of str, optional
3085: (12)                      Legend labels. Use a single string when all boxes have the same
style and
3086: (12)                      you only want a single legend entry for them. Use a list of
strings to
3087: (12)                      label all boxes individually. To be distinguishable, the boxes
should be
3088: (12)                      styled individually, which is currently only possible by modifying
the
3089: (12)                      returned artists, see e.g.
:doc:`/gallery/statistics/boxplot_demo`.
3090: (12)                      In the case of a single string, the legend entry will technically
be
3091: (12)                      associated with the first box only. By default, the legend will
show the
3092: (12)                      median line (``result["medians"]``); if *patch_artist* is True,
the legend
3093: (12)                      will show the box `.Patch` artists (``result["boxes"]``) instead.
3094: (12)                      .. versionadded:: 3.9
3095: (8)                   data : indexable object, optional
3096: (12)                      DATA_PARAMETER_PLACEHOLDER
3097: (8)                   See Also
3098: (8)                   --------
3099: (8)                   .Axes.bxp : Draw a boxplot from pre-computed statistics.
3100: (8)                   violinplot : Draw an estimate of the probability density function.
3101: (8)                   """
3102: (8)                   if whis is None:
3103: (12)                      whis = mpl.rcParams['boxplot.whiskers']
3104: (8)                   if bootstrap is None:
3105: (12)                      bootstrap = mpl.rcParams['boxplot.bootstrap']
3106: (8)                   bxpstats = cbook.boxplot_stats(x, whis=whis, bootstrap=bootstrap,
3107: (39)                                                 labels=tick_labels,
autorange=autorange)
3108: (8)                   if notch is None:
3109: (12)                      notch = mpl.rcParams['boxplot.notch']
```

```
(8)                     if vert is None:
(12)                        vert = mpl.rcParams['boxplot.vertical']
(8)                     if patch_artist is None:
(12)                        patch_artist = mpl.rcParams['boxplot.patchartist']
(8)                     if meanline is None:
(12)                        meanline = mpl.rcParams['boxplot.meanline']
(8)                     if showmeans is None:
(12)                        showmeans = mpl.rcParams['boxplot.showmeans']
(8)                     if showcaps is None:
(12)                        showcaps = mpl.rcParams['boxplot.showcaps']
(8)                     if showbox is None:
(12)                        showbox = mpl.rcParams['boxplot.showbox']
(8)                     if showfliers is None:
(12)                        showfliers = mpl.rcParams['boxplot.showfliers']
(8)                     if boxprops is None:
(12)                        boxprops = {}
(8)                     if whiskerprops is None:
(12)                        whiskerprops = {}
(8)                     if capprops is None:
(12)                        capprops = {}
(8)                     if medianprops is None:
(12)                        medianprops = {}
(8)                     if meanprops is None:
(12)                        meanprops = {}
(8)                     if flierprops is None:
(12)                        flierprops = {}
(8)                     if patch_artist:
(12)                        boxprops['linestyle'] = 'solid'  # Not consistent with bxp.
(12)                        if 'color' in boxprops:
(16)                            boxprops['edgecolor'] = boxprops.pop('color')
(8)                     if sym is not None:
(12)                        if sym == '':
(16)                            flierprops = dict(linestyle='none', marker='', color='none')
(16)                            showfliers = False
(12)                        else:
(16)                            _, marker, color = _process_plot_format(sym)
(16)                            if marker is not None:
(20)                                flierprops['marker'] = marker
(16)                            if color is not None:
(20)                                flierprops['color'] = color
(20)                                flierprops['markerfacecolor'] = color
(20)                                flierprops['markeredgecolor'] = color
(8)                     if usermedians is not None:
(12)                        if (len(np.ravel(usermedians)) != len(bxpstats) or
(20)                                np.shape(usermedians)[0] != len(bxpstats)):
(16)                            raise ValueError(
(20)                                "'usermedians' and 'x' have different lengths")
(12)                        else:
(16)                            for stats, med in zip(bxpstats, usermedians):
(20)                                if med is not None:
(24)                                    stats['med'] = med
(8)                     if conf_intervals is not None:
(12)                        if len(conf_intervals) != len(bxpstats):
(16)                            raise ValueError(
(20)                                "'conf_intervals' and 'x' have different lengths")
(12)                        else:
(16)                            for stats, ci in zip(bxpstats, conf_intervals):
(20)                                if ci is not None:
(24)                                    if len(ci) != 2:
(28)                                        raise ValueError('each confidence interval must '
(45)                                                         'have two values')
(24)                                    else:
(28)                                        if ci[0] is not None:
(32)                                            stats['cilo'] = ci[0]
(28)                                        if ci[1] is not None:
(32)                                            stats['cihi'] = ci[1]
(8)                     artists = self.bxp(bxpstats, positions=positions, widths=widths,
(27)                                       vert=vert, patch_artist=patch_artist,
(27)                                       shownotches=notch, showmeans=showmeans,
```

```
                                        showcaps=showcaps, showbox=showbox,
                                        boxprops=boxprops, flierprops=flierprops,
                                        medianprops=medianprops, meanprops=meanprops,
                                        meanline=meanline, showfliers=showfliers,
                                        capprops=capprops, whiskerprops=whiskerprops,
                                        manage_ticks=manage_ticks, zorder=zorder,
                                        capwidths=capwidths, label=label)
                    return artists
                def bxp(self, bxpstats, positions=None, widths=None, vert=True,
                        patch_artist=False, shownotches=False, showmeans=False,
                        showcaps=True, showbox=True, showfliers=True,
                        boxprops=None, whiskerprops=None, flierprops=None,
                        medianprops=None, capprops=None, meanprops=None,
                        meanline=False, manage_ticks=True, zorder=None,
                        capwidths=None, label=None):
                    """
                    Draw a box and whisker plot from pre-computed statistics.
                    The box extends from the first quartile *q1* to the third
                    quartile *q3* of the data, with a line at the median (*med*).
                    The whiskers extend from *whislow* to *whishi*.
                    Flier points are markers past the end of the whiskers.
                    See https://en.wikipedia.org/wiki/Box_plot for reference.
                    .. code-block:: none
                               whislow    q1    med    q3    whishi
                                            |-----:-----|
                           o      |--------|     :     |--------|    o  o
                                            |-----:-----|
                         flier                                     fliers
                    .. note::
                        This is a low-level drawing function for when you already
                        have the statistical parameters. If you want a boxplot based
                        on a dataset, use `~.Axes.boxplot` instead.
                    Parameters
                    ----------
                    bxpstats : list of dicts
                      A list of dictionaries containing stats for each boxplot.
                      Required keys are:
                      - ``med``: Median (scalar).
                      - ``q1``, ``q3``: First & third quartiles (scalars).
                      - ``whislo``, ``whishi``: Lower & upper whisker positions (scalars).
                      Optional keys are:
                      - ``mean``: Mean (scalar).  Needed if ``showmeans=True``.
                      - ``fliers``: Data beyond the whiskers (array-like).
                        Needed if ``showfliers=True``.
                      - ``cilo``, ``cihi``: Lower & upper confidence intervals
                        about the median. Needed if ``shownotches=True``.
                      - ``label``: Name of the dataset (str).  If available,
                        this will be used a tick label for the boxplot
                    positions : array-like, default: [1, 2, ..., n]
                      The positions of the boxes. The ticks and limits
                      are automatically set to match the positions.
                    widths : float or array-like, default: None
                      The widths of the boxes.  The default is
                      ``clip(0.15*(distance between extreme positions), 0.15, 0.5)``.
                    capwidths : float or array-like, default: None
                      Either a scalar or a vector and sets the width of each cap.
                      The default is ``0.5*(width of the box)``, see *widths*.
                    vert : bool, default: True
                      If `True` (default), makes the boxes vertical.
                      If `False`, makes horizontal boxes.
                    patch_artist : bool, default: False
                      If `False` produces boxes with the `.Line2D` artist.
                      If `True` produces boxes with the `~matplotlib.patches.Patch`
artist.
                    shownotches, showmeans, showcaps, showbox, showfliers : bool
                      Whether to draw the CI notches, the mean value (both default to
                      False), the caps, the box, and the fliers (all three default to
                      True).
                    boxprops, whiskerprops, capprops, flierprops, medianprops, meanprops
```

```
:\
(1)            dict, optional
(10)                     Artist properties for the boxes, whiskers, caps, fliers, medians,
and
(10)                     means.
(8)                    meanline : bool, default: False
(10)                     If `True` (and *showmeans* is `True`), will try to render the mean
(10)                     as a line spanning the full width of the box according to
(10)                     *meanprops*. Not recommended if *shownotches* is also True.
(10)                     Otherwise, means will be shown as points.
(8)                    manage_ticks : bool, default: True
(10)                     If True, the tick locations and labels will be adjusted to match the
(10)                     boxplot positions.
(8)                    label : str or list of str, optional
(12)                         Legend labels. Use a single string when all boxes have the same
style and
(12)                         you only want a single legend entry for them. Use a list of
strings to
(12)                         label all boxes individually. To be distinguishable, the boxes
should be
(12)                         styled individually, which is currently only possible by modifying
the
(12)                         returned artists, see e.g.
:doc:`/gallery/statistics/boxplot_demo`.
(12)                         In the case of a single string, the legend entry will technically
be
(12)                         associated with the first box only. By default, the legend will
show the
(12)                         median line (``result["medians"]``); if *patch_artist* is True,
the legend
(12)                         will show the box `.Patch` artists (``result["boxes"]``) instead.
(12)                         .. versionadded:: 3.9
(8)                    zorder : float, default: ``Line2D.zorder = 2``
(10)                     The zorder of the resulting boxplot.
(8)                    Returns
(8)                    -------
(8)                    dict
(10)                     A dictionary mapping each component of the boxplot to a list
(10)                     of the `.Line2D` instances created. That dictionary has the
(10)                     following keys (assuming vertical boxplots):
(10)                     - ``boxes``: main bodies of the boxplot showing the quartiles, and
(12)                       the median's confidence intervals if enabled.
(10)                     - ``medians``: horizontal lines at the median of each box.
(10)                     - ``whiskers``: vertical lines up to the last non-outlier data.
(10)                     - ``caps``: horizontal lines at the ends of the whiskers.
(10)                     - ``fliers``: points representing data beyond the whiskers (fliers).
(10)                     - ``means``: points or lines representing the means.
(8)                    See Also
(8)                    --------
(8)                    boxplot : Draw a boxplot from data instead of pre-computed statistics.
(8)                    """
(8)                    medianprops = {
(12)                       "solid_capstyle": "butt",
(12)                       "dash_capstyle": "butt",
(12)                       **(medianprops or {}),
(8)                    }
(8)                    meanprops = {
(12)                       "solid_capstyle": "butt",
(12)                       "dash_capstyle": "butt",
(12)                       **(meanprops or {}),
(8)                    }
(8)                    whiskers = []
(8)                    caps = []
(8)                    boxes = []
(8)                    medians = []
(8)                    means = []
(8)                    fliers = []
(8)                    datalabels = []
(8)                    if zorder is None:
```

```
3306: (12)                      zorder = mlines.Line2D.zorder
3307: (8)                   zdelta = 0.1
3308: (8)                   def merge_kw_rc(subkey, explicit, zdelta=0, usemarker=True):
3309: (12)                      d = {k.split('.')[-1]: v for k, v in mpl.rcParams.items()
3310: (17)                           if k.startswith(f'boxplot.{subkey}props')}
3311: (12)                      d['zorder'] = zorder + zdelta
3312: (12)                      if not usemarker:
3313: (16)                          d['marker'] = ''
3314: (12)                      d.update(cbook.normalize_kwargs(explicit, mlines.Line2D))
3315: (12)                      return d
3316: (8)                   box_kw = {
3317: (12)                      'linestyle': mpl.rcParams['boxplot.boxprops.linestyle'],
3318: (12)                      'linewidth': mpl.rcParams['boxplot.boxprops.linewidth'],
3319: (12)                      'edgecolor': mpl.rcParams['boxplot.boxprops.color'],
3320: (12)                      'facecolor': ('white' if mpl.rcParams['_internal.classic_mode']
3321: (26)                                    else mpl.rcParams['patch.facecolor']),
3322: (12)                      'zorder': zorder,
3323: (12)                      **cbook.normalize_kwargs(boxprops, mpatches.PathPatch)
3324: (8)                   } if patch_artist else merge_kw_rc('box', boxprops, usemarker=False)
3325: (8)                   whisker_kw = merge_kw_rc('whisker', whiskerprops, usemarker=False)
3326: (8)                   cap_kw = merge_kw_rc('cap', capprops, usemarker=False)
3327: (8)                   flier_kw = merge_kw_rc('flier', flierprops)
3328: (8)                   median_kw = merge_kw_rc('median', medianprops, zdelta,
usemarker=False)
3329: (8)                   mean_kw = merge_kw_rc('mean', meanprops, zdelta)
3330: (8)                   removed_prop = 'marker' if meanline else 'linestyle'
3331: (8)                   if meanprops is None or removed_prop not in meanprops:
3332: (12)                      mean_kw[removed_prop] = ''
3333: (8)                   maybe_swap = slice(None) if vert else slice(None, None, -1)
3334: (8)                   def do_plot(xs, ys, **kwargs):
3335: (12)                      return self.plot(*[xs, ys][maybe_swap], **kwargs)[0]
3336: (8)                   def do_patch(xs, ys, **kwargs):
3337: (12)                      path = mpath.Path._create_closed(
3338: (16)                          np.column_stack([xs, ys][maybe_swap]))
3339: (12)                      patch = mpatches.PathPatch(path, **kwargs)
3340: (12)                      self.add_artist(patch)
3341: (12)                      return patch
3342: (8)                   N = len(bxpstats)
3343: (8)                   datashape_message = ("List of boxplot statistics and `{0}` "
3344: (29)                                       "values must have same the length")
3345: (8)                   if positions is None:
3346: (12)                      positions = list(range(1, N + 1))
3347: (8)                   elif len(positions) != N:
3348: (12)                      raise ValueError(datashape_message.format("positions"))
3349: (8)                   positions = np.array(positions)
3350: (8)                   if len(positions) > 0 and not all(isinstance(p, Real) for p in
positions):
3351: (12)                      raise TypeError("positions should be an iterable of numbers")
3352: (8)                   if widths is None:
3353: (12)                      widths = [np.clip(0.15 * np.ptp(positions), 0.15, 0.5)] * N
3354: (8)                   elif np.isscalar(widths):
3355: (12)                      widths = [widths] * N
3356: (8)                   elif len(widths) != N:
3357: (12)                      raise ValueError(datashape_message.format("widths"))
3358: (8)                   if capwidths is None:
3359: (12)                      capwidths = 0.5 * np.array(widths)
3360: (8)                   elif np.isscalar(capwidths):
3361: (12)                      capwidths = [capwidths] * N
3362: (8)                   elif len(capwidths) != N:
3363: (12)                      raise ValueError(datashape_message.format("capwidths"))
3364: (8)                   for pos, width, stats, capwidth in zip(positions, widths, bxpstats,
3365: (47)                                                         capwidths):
3366: (12)                      datalabels.append(stats.get('label', pos))
3367: (12)                      whis_x = [pos, pos]
3368: (12)                      whislo_y = [stats['q1'], stats['whislo']]
3369: (12)                      whishi_y = [stats['q3'], stats['whishi']]
3370: (12)                      cap_left = pos - capwidth * 0.5
3371: (12)                      cap_right = pos + capwidth * 0.5
3372: (12)                      cap_x = [cap_left, cap_right]
```

```
(12)                        cap_lo = np.full(2, stats['whislo'])
(12)                        cap_hi = np.full(2, stats['whishi'])
(12)                        box_left = pos - width * 0.5
(12)                        box_right = pos + width * 0.5
(12)                        med_y = [stats['med'], stats['med']]
(12)                        if shownotches:
(16)                            notch_left = pos - width * 0.25
(16)                            notch_right = pos + width * 0.25
(16)                            box_x = [box_left, box_right, box_right, notch_right,
(25)                                     box_right, box_right, box_left, box_left, notch_left,
(25)                                     box_left, box_left]
(16)                            box_y = [stats['q1'], stats['q1'], stats['cilo'],
(25)                                     stats['med'], stats['cihi'], stats['q3'],
(25)                                     stats['q3'], stats['cihi'], stats['med'],
(25)                                     stats['cilo'], stats['q1']]
(16)                            med_x = [notch_left, notch_right]
(12)                        else:
(16)                            box_x = [box_left, box_right, box_right, box_left, box_left]
(16)                            box_y = [stats['q1'], stats['q1'], stats['q3'], stats['q3'],
(25)                                     stats['q1']]
(16)                            med_x = [box_left, box_right]
(12)                        if showbox:
(16)                            do_box = do_patch if patch_artist else do_plot
(16)                            boxes.append(do_box(box_x, box_y, **box_kw))
(16)                            median_kw.setdefault('label', '_nolegend_')
(12)                        whisker_kw.setdefault('label', '_nolegend_')
(12)                        whiskers.append(do_plot(whis_x, whislo_y, **whisker_kw))
(12)                        whiskers.append(do_plot(whis_x, whishi_y, **whisker_kw))
(12)                        if showcaps:
(16)                            cap_kw.setdefault('label', '_nolegend_')
(16)                            caps.append(do_plot(cap_x, cap_lo, **cap_kw))
(16)                            caps.append(do_plot(cap_x, cap_hi, **cap_kw))
(12)                        medians.append(do_plot(med_x, med_y, **median_kw))
(12)                        if showmeans:
(16)                            if meanline:
(20)                                means.append(do_plot(
(24)                                    [box_left, box_right], [stats['mean'], stats['mean']],
(24)                                    **mean_kw
(20)                                ))
(16)                            else:
(20)                                means.append(do_plot([pos], [stats['mean']], **mean_kw))
(12)                        if showfliers:
(16)                            flier_kw.setdefault('label', '_nolegend_')
(16)                            flier_x = np.full(len(stats['fliers']), pos, dtype=np.float64)
(16)                            flier_y = stats['fliers']
(16)                            fliers.append(do_plot(flier_x, flier_y, **flier_kw))
(8)                     if label:
(12)                        box_or_med = boxes if showbox and patch_artist else medians
(12)                        if cbook.is_scalar_or_string(label):
(16)                            box_or_med[0].set_label(label)
(12)                        else:  # label is a sequence
(16)                            if len(box_or_med) != len(label):
(20)                                raise ValueError(datashape_message.format("label"))
(16)                            for artist, lbl in zip(box_or_med, label):
(20)                                artist.set_label(lbl)
(8)                     if manage_ticks:
(12)                        axis_name = "x" if vert else "y"
(12)                        interval = getattr(self.dataLim, f"interval{axis_name}")
(12)                        axis = self._axis_map[axis_name]
(12)                        positions = axis.convert_units(positions)
(12)                        interval[:] = (min(interval[0], min(positions) - .5),
(27)                                       max(interval[1], max(positions) + .5))
(12)                        for median, position in zip(medians, positions):
(16)                            getattr(median.sticky_edges, axis_name).extend(
(20)                                [position - .5, position + .5])
(12)                        locator = axis.get_major_locator()
(12)                        if not isinstance(axis.get_major_locator(),
(30)                                          mticker.FixedLocator):
(16)                            locator = mticker.FixedLocator([])
```

```
(16)                            axis.set_major_locator(locator)
(12)                        locator.locs = np.array([*locator.locs, *positions])
(12)                        formatter = axis.get_major_formatter()
(12)                        if not isinstance(axis.get_major_formatter(),
(30)                                          mticker.FixedFormatter):
(16)                            formatter = mticker.FixedFormatter([])
(16)                            axis.set_major_formatter(formatter)
(12)                        formatter.seq = [*formatter.seq, *datalabels]
(12)                        self._request_autoscale_view()
(8)                     return dict(whiskers=whiskers, caps=caps, boxes=boxes,
(20)                                medians=medians, fliers=fliers, means=means)
(4)                 @staticmethod
(4)                 def _parse_scatter_color_args(c, edgecolors, kwargs, xsize,
(34)                                              get_next_color_func):
(8)                     """
(8)                     Helper function to process color related arguments of `.Axes.scatter`.
(8)                     Argument precedence for facecolors:
(8)                     - c (if not None)
(8)                     - kwargs['facecolor']
(8)                     - kwargs['facecolors']
(8)                     - kwargs['color'] (==kwcolor)
(8)                     - 'b' if in classic mode else the result of ``get_next_color_func()``
(8)                     Argument precedence for edgecolors:
(8)                     - kwargs['edgecolor']
(8)                     - edgecolors (is an explicit kw argument in scatter())
(8)                     - kwargs['color'] (==kwcolor)
(8)                     - 'face' if not in classic mode else None
(8)                     Parameters
(8)                     ----------
(8)                     c : :mpltype:`color` or array-like or list of :mpltype:`color` or None
(12)                        See argument description of `.Axes.scatter`.
(8)                     edgecolors : :mpltype:`color` or sequence of color or {'face', 'none'}
or None
(12)                        See argument description of `.Axes.scatter`.
(8)                     kwargs : dict
(12)                        Additional kwargs. If these keys exist, we pop and process them:
(12)                        'facecolors', 'facecolor', 'edgecolor', 'color'
(12)                        Note: The dict is modified by this function.
(8)                     xsize : int
(12)                        The size of the x and y arrays passed to `.Axes.scatter`.
(8)                     get_next_color_func : callable
(12)                        A callable that returns a color. This color is used as facecolor
(12)                        if no other color is provided.
(12)                        Note, that this is a function rather than a fixed color value to
(12)                        support conditional evaluation of the next color.  As of the
(12)                        current implementation obtaining the next color from the
(12)                        property cycle advances the cycle. This must only happen if we
(12)                        actually use the color, which will only be decided within this
(12)                        method.
(8)                     Returns
(8)                     -------
(8)                     c
(12)                        The input *c* if it was not *None*, else a color derived from the
(12)                        other inputs or defaults.
(8)                     colors : array(N, 4) or None
(12)                        The facecolors as RGBA values, or *None* if a colormap is used.
(8)                     edgecolors
(12)                        The edgecolor.
(8)                     """
(8)                     facecolors = kwargs.pop('facecolors', None)
(8)                     facecolors = kwargs.pop('facecolor', facecolors)
(8)                     edgecolors = kwargs.pop('edgecolor', edgecolors)
(8)                     kwcolor = kwargs.pop('color', None)
(8)                     if kwcolor is not None and c is not None:
(12)                        raise ValueError("Supply a 'c' argument or a 'color'"
(29)                                         " kwarg but not both; they differ but"
(29)                                         " their functionalities overlap.")
(8)                     if kwcolor is not None:
(12)                        try:
```

```
3510: (16)                            mcolors.to_rgba_array(kwcolor)
3511: (12)                        except ValueError as err:
3512: (16)                            raise ValueError(
3513: (20)                                "'color' kwarg must be a color or sequence of color "
3514: (20)                                "specs.  For a sequence of values to be color-mapped, use
"
3515: (20)                                "the 'c' argument instead.") from err
3516: (12)                        if edgecolors is None:
3517: (16)                            edgecolors = kwcolor
3518: (12)                        if facecolors is None:
3519: (16)                            facecolors = kwcolor
3520: (8)                     if edgecolors is None and not mpl.rcParams['_internal.classic_mode']:
3521: (12)                        edgecolors = mpl.rcParams['scatter.edgecolors']
3522: (8)                     c_was_none = c is None
3523: (8)                     if c is None:
3524: (12)                        c = (facecolors if facecolors is not None
3525: (17)                             else "b" if mpl.rcParams['_internal.classic_mode']
3526: (17)                             else get_next_color_func())
3527: (8)                     c_is_string_or_strings = (
3528: (12)                        isinstance(c, str)
3529: (12)                        or (np.iterable(c) and len(c) > 0
3530: (16)                            and isinstance(cbook._safe_first_finite(c), str)))
3531: (8)                     def invalid_shape_exception(csize, xsize):
3532: (12)                        return ValueError(
3533: (16)                            f"'c' argument has {csize} elements, which is inconsistent "
3534: (16)                            f"with 'x' and 'y' with size {xsize}.")
3535: (8)                     c_is_mapped = False  # Unless proven otherwise below.
3536: (8)                     valid_shape = True  # Unless proven otherwise below.
3537: (8)                     if not c_was_none and kwcolor is None and not c_is_string_or_strings:
3538: (12)                        try:  # First, does 'c' look suitable for value-mapping?
3539: (16)                            c = np.asanyarray(c, dtype=float)
3540: (12)                        except ValueError:
3541: (16)                            pass  # Failed to convert to float array; must be color specs.
3542: (12)                        else:
3543: (16)                            if c.shape == (1, 4) or c.shape == (1, 3):
3544: (20)                                c_is_mapped = False
3545: (20)                                if c.size != xsize:
3546: (24)                                    valid_shape = False
3547: (16)                            elif c.size == xsize:
3548: (20)                                c = c.ravel()
3549: (20)                                c_is_mapped = True
3550: (16)                            else:  # Wrong size; it must not be intended for mapping.
3551: (20)                                if c.shape in ((3,), (4,)):
3552: (24)                                    _api.warn_external(
3553: (28)                                        "*c* argument looks like a single numeric RGB or "
3554: (28)                                        "RGBA sequence, which should be avoided as value-"
3555: (28)                                        "mapping will have precedence in case its length "
3556: (28)                                        "matches with *x* & *y*.  Please use the *color* "
3557: (28)                                        "keyword-argument or provide a 2D array "
3558: (28)                                        "with a single row if you intend to specify "
3559: (28)                                        "the same RGB or RGBA value for all points.")
3560: (20)                                valid_shape = False
3561: (8)                     if not c_is_mapped:
3562: (12)                        try:  # Is 'c' acceptable as PathCollection facecolors?
3563: (16)                            colors = mcolors.to_rgba_array(c)
3564: (12)                        except (TypeError, ValueError) as err:
3565: (16)                            if "RGBA values should be within 0-1 range" in str(err):
3566: (20)                                raise
3567: (16)                            else:
3568: (20)                                if not valid_shape:
3569: (24)                                    raise invalid_shape_exception(c.size, xsize) from err
3570: (20)                                raise ValueError(
3571: (24)                                    f"'c' argument must be a color, a sequence of colors,
"
3572: (24)                                    f"or a sequence of numbers, not {c!r}") from err
3573: (12)                        else:
3574: (16)                            if len(colors) not in (0, 1, xsize):
3575: (20)                                raise invalid_shape_exception(len(colors), xsize)
3576: (8)                     else:
```

```
3577: (12)                      colors = None  # use cmap, norm after collection is created
3578: (8)                   return c, colors, edgecolors
3579: (4)               @_preprocess_data(replace_names=["x", "y", "s", "linewidths",
3580: (37)                                        "edgecolors", "c", "facecolor",
3581: (37)                                        "facecolors", "color"],
3582: (22)                                 label_namer="y")
3583: (4)               @_docstring.interpd
3584: (4)               def scatter(self, x, y, s=None, c=None, marker=None, cmap=None, norm=None,
3585: (16)                          vmin=None, vmax=None, alpha=None, linewidths=None, *,
3586: (16)                          edgecolors=None, plotnonfinite=False, **kwargs):
3587: (8)                   """
3588: (8)                   A scatter plot of *y* vs. *x* with varying marker size and/or color.
3589: (8)                   Parameters
3590: (8)                   ----------
3591: (8)                   x, y : float or array-like, shape (n, )
3592: (12)                      The data positions.
3593: (8)                   s : float or array-like, shape (n, ), optional
3594: (12)                      The marker size in points**2 (typographic points are 1/72 in.).
3595: (12)                      Default is ``rcParams['lines.markersize'] ** 2``.
3596: (12)                      The linewidth and edgecolor can visually interact with the marker
3597: (12)                      size, and can lead to artifacts if the marker size is smaller than
3598: (12)                      the linewidth.
3599: (12)                      If the linewidth is greater than 0 and the edgecolor is anything
3600: (12)                      but *'none'*, then the effective size of the marker will be
3601: (12)                      increased by half the linewidth because the stroke will be
centered
3602: (12)                      on the edge of the shape.
3603: (12)                      To eliminate the marker edge either set *linewidth=0* or
3604: (12)                      *edgecolor='none'*.
3605: (8)                   c : array-like or list of :mpltype:`color` or :mpltype:`color`,
optional
3606: (12)                      The marker colors. Possible values:
3607: (12)                      - A scalar or sequence of n numbers to be mapped to colors using
3608: (14)                        *cmap* and *norm*.
3609: (12)                      - A 2D array in which the rows are RGB or RGBA.
3610: (12)                      - A sequence of colors of length n.
3611: (12)                      - A single color format string.
3612: (12)                      Note that *c* should not be a single numeric RGB or RGBA sequence
3613: (12)                      because that is indistinguishable from an array of values to be
3614: (12)                      colormapped. If you want to specify the same RGB or RGBA value for
3615: (12)                      all points, use a 2D array with a single row.  Otherwise,
3616: (12)                      value-matching will have precedence in case of a size matching
with
3617: (12)                      *x* and *y*.
3618: (12)                      If you wish to specify a single color for all points
3619: (12)                      prefer the *color* keyword argument.
3620: (12)                      Defaults to `None`. In that case the marker color is determined
3621: (12)                      by the value of *color*, *facecolor* or *facecolors*. In case
3622: (12)                      those are not specified or `None`, the marker color is determined
3623: (12)                      by the next color of the ``Axes``' current "shape and fill" color
3624: (12)                      cycle. This cycle defaults to :rc:`axes.prop_cycle`.
3625: (8)                   marker : `~.markers.MarkerStyle`, default: :rc:`scatter.marker`
3626: (12)                      The marker style. *marker* can be either an instance of the class
3627: (12)                      or the text shorthand for a particular marker.
3628: (12)                      See :mod:`matplotlib.markers` for more information about marker
3629: (12)                      styles.
3630: (8)                   %(cmap_doc)s
3631: (12)                      This parameter is ignored if *c* is RGB(A).
3632: (8)                   %(norm_doc)s
3633: (12)                      This parameter is ignored if *c* is RGB(A).
3634: (8)                   %(vmin_vmax_doc)s
3635: (12)                      This parameter is ignored if *c* is RGB(A).
3636: (8)                   alpha : float, default: None
3637: (12)                      The alpha blending value, between 0 (transparent) and 1 (opaque).
3638: (8)                   linewidths : float or array-like, default: :rc:`lines.linewidth`
3639: (12)                      The linewidth of the marker edges. Note: The default *edgecolors*
3640: (12)                      is 'face'. You may want to change this as well.
3641: (8)                   edgecolors : {'face', 'none', *None*} or :mpltype:`color` or list of \
3642: (0)            :mpltype:`color`, default: :rc:`scatter.edgecolors`
```

```
(12)                        The edge color of the marker. Possible values:
(12)                        - 'face': The edge color will always be the same as the face
color.
(12)                        - 'none': No patch boundary will be drawn.
(12)                        - A color or sequence of colors.
(12)                        For non-filled markers, *edgecolors* is ignored. Instead, the
color
(12)                        is determined like with 'face', i.e. from *c*, *colors*, or
(12)                        *facecolors*.
(8)                     plotnonfinite : bool, default: False
(12)                        Whether to plot points with nonfinite *c* (i.e. ``inf``, ``-inf``
(12)                        or ``nan``). If ``True`` the points are drawn with the *bad*
(12)                        colormap color (see `.Colormap.set_bad`).
(8)                     Returns
(8)                     -------
(8)                     `~matplotlib.collections.PathCollection`
(8)                     Other Parameters
(8)                     ----------------
(8)                     data : indexable object, optional
(12)                        DATA_PARAMETER_PLACEHOLDER
(8)                     **kwargs : `~matplotlib.collections.Collection` properties
(8)                     See Also
(8)                     --------
(8)                     plot : To plot scatter plots when markers are identical in size and
(12)                        color.
(8)                     Notes
(8)                     -----
(8)                     * The `.plot` function will be faster for scatterplots where markers
(10)                      don't vary in size or color.
(8)                     * Any or all of *x*, *y*, *s*, and *c* may be masked arrays, in which
(10)                      case all masks will be combined and only unmasked points will be
(10)                      plotted.
(8)                     * Fundamentally, scatter works with 1D arrays; *x*, *y*, *s*, and *c*
(10)                      may be input as N-D arrays, but within scatter they will be
(10)                      flattened. The exception is *c*, which will be flattened only if its
(10)                      size matches the size of *x* and *y*.
(8)                     """
(8)                     if edgecolors is not None:
(12)                        kwargs.update({'edgecolors': edgecolors})
(8)                     if linewidths is not None:
(12)                        kwargs.update({'linewidths': linewidths})
(8)                     kwargs = cbook.normalize_kwargs(kwargs, mcoll.Collection)
(8)                     linewidths = kwargs.pop('linewidth', None)
(8)                     edgecolors = kwargs.pop('edgecolor', None)
(8)                     x, y = self._process_unit_info([("x", x), ("y", y)], kwargs)
(8)                     x = np.ma.ravel(x)
(8)                     y = np.ma.ravel(y)
(8)                     if x.size != y.size:
(12)                        raise ValueError("x and y must be the same size")
(8)                     if s is None:
(12)                        s = (20 if mpl.rcParams['_internal.classic_mode'] else
(17)                             mpl.rcParams['lines.markersize'] ** 2.0)
(8)                     s = np.ma.ravel(s)
(8)                     if (len(s) not in (1, x.size) or
(16)                            (not np.issubdtype(s.dtype, np.floating) and
(17)                             not np.issubdtype(s.dtype, np.integer))):
(12)                        raise ValueError(
(16)                            "s must be a scalar, "
(16)                            "or float array-like with the same size as x and y")
(8)                     orig_edgecolor = edgecolors
(8)                     if edgecolors is None:
(12)                        orig_edgecolor = kwargs.get('edgecolor', None)
(8)                     c, colors, edgecolors = \
(12)                        self._parse_scatter_color_args(
(16)                            c, edgecolors, kwargs, x.size,
(16)                            get_next_color_func=self._get_patches_for_fill.get_next_color)
(8)                     if plotnonfinite and colors is None:
(12)                        c = np.ma.masked_invalid(c)
(12)                        x, y, s, edgecolors, linewidths = \
```

```
3710: (16)                          cbook._combine_masks(x, y, s, edgecolors, linewidths)
3711: (8)                   else:
3712: (12)                      x, y, s, c, colors, edgecolors, linewidths = \
3713: (16)                          cbook._combine_masks(
3714: (20)                              x, y, s, c, colors, edgecolors, linewidths)
3715: (8)                   if (x.size in (3, 4)
3716: (16)                          and np.ma.is_masked(edgecolors)
3717: (16)                          and not np.ma.is_masked(orig_edgecolor)):
3718: (12)                      edgecolors = edgecolors.data
3719: (8)                   scales = s   # Renamed for readability below.
3720: (8)                   if marker is None:
3721: (12)                      marker = mpl.rcParams['scatter.marker']
3722: (8)                   if isinstance(marker, mmarkers.MarkerStyle):
3723: (12)                      marker_obj = marker
3724: (8)                   else:
3725: (12)                      marker_obj = mmarkers.MarkerStyle(marker)
3726: (8)                   path = marker_obj.get_path().transformed(
3727: (12)                      marker_obj.get_transform())
3728: (8)                   if not marker_obj.is_filled():
3729: (12)                      if orig_edgecolor is not None:
3730: (16)                          _api.warn_external(
3731: (20)                              f"You passed a edgecolor/edgecolors ({orig_edgecolor!r}) "
3732: (20)                              f"for an unfilled marker ({marker!r}).  Matplotlib is "
3733: (20)                              "ignoring the edgecolor in favor of the facecolor.  This "
3734: (20)                              "behavior may change in the future."
3735: (16)                          )
3736: (12)                      if marker_obj.get_fillstyle() == 'none':
3737: (16)                          edgecolors = colors
3738: (16)                          colors = 'none'
3739: (12)                      else:
3740: (16)                          edgecolors = 'face'
3741: (12)                      if linewidths is None:
3742: (16)                          linewidths = mpl.rcParams['lines.linewidth']
3743: (12)                      elif np.iterable(linewidths):
3744: (16)                          linewidths = [
3745: (20)                              lw if lw is not None else mpl.rcParams['lines.linewidth']
3746: (20)                              for lw in linewidths]
3747: (8)                   offsets = np.ma.column_stack([x, y])
3748: (8)                   collection = mcoll.PathCollection(
3749: (12)                      (path,), scales,
3750: (12)                      facecolors=colors,
3751: (12)                      edgecolors=edgecolors,
3752: (12)                      linewidths=linewidths,
3753: (12)                      offsets=offsets,
3754: (12)                      offset_transform=kwargs.pop('transform', self.transData),
3755: (12)                      alpha=alpha,
3756: (8)                   )
3757: (8)                   collection.set_transform(mtransforms.IdentityTransform())
3758: (8)                   if colors is None:
3759: (12)                      collection.set_array(c)
3760: (12)                      collection.set_cmap(cmap)
3761: (12)                      collection.set_norm(norm)
3762: (12)                      collection._scale_norm(norm, vmin, vmax)
3763: (8)                   else:
3764: (12)                      extra_kwargs = {
3765: (20)                              'cmap': cmap, 'norm': norm, 'vmin': vmin, 'vmax': vmax
3766: (20)                              }
3767: (12)                      extra_keys = [k for k, v in extra_kwargs.items() if v is not None]
3768: (12)                      if any(extra_keys):
3769: (16)                          keys_str = ", ".join(f"'{k}'" for k in extra_keys)
3770: (16)                          _api.warn_external(
3771: (20)                              "No data for colormapping provided via 'c'. "
3772: (20)                              f"Parameters {keys_str} will be ignored")
3773: (8)                   collection._internal_update(kwargs)
3774: (8)                   if mpl.rcParams['_internal.classic_mode']:
3775: (12)                      if self._xmargin < 0.05 and x.size > 0:
3776: (16)                          self.set_xmargin(0.05)
3777: (12)                      if self._ymargin < 0.05 and x.size > 0:
3778: (16)                          self.set_ymargin(0.05)
```

```
(8)                 self.add_collection(collection)
(8)                 self._request_autoscale_view()
(8)                 return collection
(4)             @_preprocess_data(replace_names=["x", "y", "C"], label_namer="y")
(4)             @_docstring.dedent_interpd
(4)             def hexbin(self, x, y, C=None, gridsize=100, bins=None,
(15)                       xscale='linear', yscale='linear', extent=None,
(15)                       cmap=None, norm=None, vmin=None, vmax=None,
(15)                       alpha=None, linewidths=None, edgecolors='face',
(15)                       reduce_C_function=np.mean, mincnt=None, marginals=False,
(15)                       **kwargs):
(8)                 """
(8)                 Make a 2D hexagonal binning plot of points *x*, *y*.
(8)                 If *C* is *None*, the value of the hexagon is determined by the number
(8)                 of points in the hexagon. Otherwise, *C* specifies values at the
(8)                 coordinate (x[i], y[i]). For each hexagon, these values are reduced
(8)                 using *reduce_C_function*.
(8)                 Parameters
(8)                 ----------
(8)                 x, y : array-like
(12)                    The data positions. *x* and *y* must be of the same length.
(8)                 C : array-like, optional
(12)                    If given, these values are accumulated in the bins. Otherwise,
(12)                    every point has a value of 1. Must be of the same length as *x*
(12)                    and *y*.
(8)                 gridsize : int or (int, int), default: 100
(12)                    If a single int, the number of hexagons in the *x*-direction.
(12)                    The number of hexagons in the *y*-direction is chosen such that
(12)                    the hexagons are approximately regular.
(12)                    Alternatively, if a tuple (*nx*, *ny*), the number of hexagons
(12)                    in the *x*-direction and the *y*-direction. In the
(12)                    *y*-direction, counting is done along vertically aligned
(12)                    hexagons, not along the zig-zag chains of hexagons; see the
(12)                    following illustration.
(12)                    .. plot::
(15)                       import numpy
(15)                       import matplotlib.pyplot as plt
(15)                       np.random.seed(19680801)
(15)                       n= 300
(15)                       x = np.random.standard_normal(n)
(15)                       y = np.random.standard_normal(n)
(15)                       fig, ax = plt.subplots(figsize=(4, 4))
(15)                       h = ax.hexbin(x, y, gridsize=(5, 3))
(15)                       hx, hy = h.get_offsets().T
(15)                       ax.plot(hx[24::3], hy[24::3], 'ro-')
(15)                       ax.plot(hx[-3:], hy[-3:], 'ro-')
(15)                       ax.set_title('gridsize=(5, 3)')
(15)                       ax.axis('off')
(12)                    To get approximately regular hexagons, choose
(12)                    :math:`n_x = \\sqrt{3}\\,n_y`.
(8)                 bins : 'log' or int or sequence, default: None
(12)                    Discretization of the hexagon values.
(12)                    - If *None*, no binning is applied; the color of each hexagon
(14)                      directly corresponds to its count value.
(12)                    - If 'log', use a logarithmic scale for the colormap.
(14)                      Internally, :math:`log_{10}(i+1)` is used to determine the
(14)                      hexagon color. This is equivalent to ``norm=LogNorm()``.
(12)                    - If an integer, divide the counts in the specified number
(14)                      of bins, and color the hexagons accordingly.
(12)                    - If a sequence of values, the values of the lower bound of
(14)                      the bins to be used.
(8)                 xscale : {'linear', 'log'}, default: 'linear'
(12)                    Use a linear or log10 scale on the horizontal axis.
(8)                 yscale : {'linear', 'log'}, default: 'linear'
(12)                    Use a linear or log10 scale on the vertical axis.
(8)                 mincnt : int >= 0, default: *None*
(12)                    If not *None*, only display cells with at least *mincnt*
(12)                    number of points in the cell.
(8)                 marginals : bool, default: *False*
```

```
3848: (12)                      If marginals is *True*, plot the marginal density as
3849: (12)                      colormapped rectangles along the bottom of the x-axis and
3850: (12)                      left of the y-axis.
3851: (8)                   extent : 4-tuple of float, default: *None*
3852: (12)                      The limits of the bins (xmin, xmax, ymin, ymax).
3853: (12)                      The default assigns the limits based on
3854: (12)                      *gridsize*, *x*, *y*, *xscale* and *yscale*.
3855: (12)                      If *xscale* or *yscale* is set to 'log', the limits are
3856: (12)                      expected to be the exponent for a power of 10. E.g. for
3857: (12)                      x-limits of 1 and 50 in 'linear' scale and y-limits
3858: (12)                      of 10 and 1000 in 'log' scale, enter (1, 50, 1, 3).
3859: (8)                   Returns
3860: (8)                   -------
3861: (8)                   `~matplotlib.collections.PolyCollection`
3862: (12)                      A `.PolyCollection` defining the hexagonal bins.
3863: (12)                      - `.PolyCollection.get_offsets` contains a Mx2 array containing
3864: (14)                        the x, y positions of the M hexagon centers.
3865: (12)                      - `.PolyCollection.get_array` contains the values of the M
3866: (14)                        hexagons.
3867: (12)                      If *marginals* is *True*, horizontal
3868: (12)                      bar and vertical bar (both PolyCollections) will be attached
3869: (12)                      to the return collection as attributes *hbar* and *vbar*.
3870: (8)                   Other Parameters
3871: (8)                   ----------------
3872: (8)                   %(cmap_doc)s
3873: (8)                   %(norm_doc)s
3874: (8)                   %(vmin_vmax_doc)s
3875: (8)                   alpha : float between 0 and 1, optional
3876: (12)                      The alpha blending value, between 0 (transparent) and 1 (opaque).
3877: (8)                   linewidths : float, default: *None*
3878: (12)                      If *None*, defaults to :rc:`patch.linewidth`.
3879: (8)                   edgecolors : {'face', 'none', *None*} or color, default: 'face'
3880: (12)                      The color of the hexagon edges. Possible values are:
3881: (12)                      - 'face': Draw the edges in the same color as the fill color.
3882: (12)                      - 'none': No edges are drawn. This can sometimes lead to unsightly
3883: (14)                        unpainted pixels between the hexagons.
3884: (12)                      - *None*: Draw outlines in the default color.
3885: (12)                      - An explicit color.
3886: (8)                   reduce_C_function : callable, default: `numpy.mean`
3887: (12)                      The function to aggregate *C* within the bins. It is ignored if
3888: (12)                      *C* is not given. This must have the signature::
3889: (16)                          def reduce_C_function(C: array) -> float
3890: (12)                      Commonly used functions are:
3891: (12)                      - `numpy.mean`: average of the points
3892: (12)                      - `numpy.sum`: integral of the point values
3893: (12)                      - `numpy.amax`: value taken from the largest point
3894: (12)                      By default will only reduce cells with at least 1 point because
some
3895: (12)                      reduction functions (such as `numpy.amax`) will error/warn with
empty
3896: (12)                      input. Changing *mincnt* will adjust the cutoff, and if set to 0
will
3897: (12)                      pass empty input to the reduction function.
3898: (8)                   data : indexable object, optional
3899: (12)                      DATA_PARAMETER_PLACEHOLDER
3900: (8)                   **kwargs : `~matplotlib.collections.PolyCollection` properties
3901: (12)                      All other keyword arguments are passed on to `.PolyCollection`:
3902: (12)                      %(PolyCollection:kwdoc)s
3903: (8)                   See Also
3904: (8)                   --------
3905: (8)                   hist2d : 2D histogram rectangular bins
3906: (8)                   """
3907: (8)                   self._process_unit_info([("x", x), ("y", y)], kwargs, convert=False)
3908: (8)                   x, y, C = cbook.delete_masked_points(x, y, C)
3909: (8)                   if np.iterable(gridsize):
3910: (12)                      nx, ny = gridsize
3911: (8)                   else:
3912: (12)                      nx = gridsize
3913: (12)                      ny = int(nx / math.sqrt(3))
```

```
(8)                 x = np.asarray(x, float)
(8)                 y = np.asarray(y, float)
(8)                 tx = x
(8)                 ty = y
(8)                 if xscale == 'log':
(12)                    if np.any(x <= 0.0):
(16)                        raise ValueError(
(20)                            "x contains non-positive values, so cannot be log-scaled")
(12)                    tx = np.log10(tx)
(8)                 if yscale == 'log':
(12)                    if np.any(y <= 0.0):
(16)                        raise ValueError(
(20)                            "y contains non-positive values, so cannot be log-scaled")
(12)                    ty = np.log10(ty)
(8)                 if extent is not None:
(12)                    xmin, xmax, ymin, ymax = extent
(12)                    if xmin > xmax:
(16)                        raise ValueError("In extent, xmax must be greater than xmin")
(12)                    if ymin > ymax:
(16)                        raise ValueError("In extent, ymax must be greater than ymin")
(8)                 else:
(12)                    xmin, xmax = (tx.min(), tx.max()) if len(x) else (0, 1)
(12)                    ymin, ymax = (ty.min(), ty.max()) if len(y) else (0, 1)
(12)                    xmin, xmax = mtransforms.nonsingular(xmin, xmax, expander=0.1)
(12)                    ymin, ymax = mtransforms.nonsingular(ymin, ymax, expander=0.1)
(8)                 nx1 = nx + 1
(8)                 ny1 = ny + 1
(8)                 nx2 = nx
(8)                 ny2 = ny
(8)                 n = nx1 * ny1 + nx2 * ny2
(8)                 padding = 1.e-9 * (xmax - xmin)
(8)                 xmin -= padding
(8)                 xmax += padding
(8)                 sx = (xmax - xmin) / nx
(8)                 sy = (ymax - ymin) / ny
(8)                 ix = (tx - xmin) / sx
(8)                 iy = (ty - ymin) / sy
(8)                 ix1 = np.round(ix).astype(int)
(8)                 iy1 = np.round(iy).astype(int)
(8)                 ix2 = np.floor(ix).astype(int)
(8)                 iy2 = np.floor(iy).astype(int)
(8)                 i1 = np.where((0 <= ix1) & (ix1 < nx1) & (0 <= iy1) & (iy1 < ny1),
(22)                              ix1 * ny1 + iy1 + 1, 0)
(8)                 i2 = np.where((0 <= ix2) & (ix2 < nx2) & (0 <= iy2) & (iy2 < ny2),
(22)                              ix2 * ny2 + iy2 + 1, 0)
(8)                 d1 = (ix - ix1) ** 2 + 3.0 * (iy - iy1) ** 2
(8)                 d2 = (ix - ix2 - 0.5) ** 2 + 3.0 * (iy - iy2 - 0.5) ** 2
(8)                 bdist = (d1 < d2)
(8)                 if C is None:  # [1:] drops out-of-range points.
(12)                    counts1 = np.bincount(i1[bdist], minlength=1 + nx1 * ny1)[1:]
(12)                    counts2 = np.bincount(i2[~bdist], minlength=1 + nx2 * ny2)[1:]
(12)                    accum = np.concatenate([counts1, counts2]).astype(float)
(12)                    if mincnt is not None:
(16)                        accum[accum < mincnt] = np.nan
(12)                    C = np.ones(len(x))
(8)                 else:
(12)                    Cs_at_i1 = [[] for _ in range(1 + nx1 * ny1)]
(12)                    Cs_at_i2 = [[] for _ in range(1 + nx2 * ny2)]
(12)                    for i in range(len(x)):
(16)                        if bdist[i]:
(20)                            Cs_at_i1[i1[i]].append(C[i])
(16)                        else:
(20)                            Cs_at_i2[i2[i]].append(C[i])
(12)                    if mincnt is None:
(16)                        mincnt = 1
(12)                    accum = np.array(
(16)                        [reduce_C_function(acc) if len(acc) >= mincnt else np.nan
(17)                         for Cs_at_i in [Cs_at_i1, Cs_at_i2]
(17)                         for acc in Cs_at_i[1:]],  # [1:] drops out-of-range points.
```

```
(16)                            float)
(8)                     good_idxs = ~np.isnan(accum)
(8)                     offsets = np.zeros((n, 2), float)
(8)                     offsets[:nx1 * ny1, 0] = np.repeat(np.arange(nx1), ny1)
(8)                     offsets[:nx1 * ny1, 1] = np.tile(np.arange(ny1), nx1)
(8)                     offsets[nx1 * ny1:, 0] = np.repeat(np.arange(nx2) + 0.5, ny2)
(8)                     offsets[nx1 * ny1:, 1] = np.tile(np.arange(ny2), nx2) + 0.5
(8)                     offsets[:, 0] *= sx
(8)                     offsets[:, 1] *= sy
(8)                     offsets[:, 0] += xmin
(8)                     offsets[:, 1] += ymin
(8)                     offsets = offsets[good_idxs, :]
(8)                     accum = accum[good_idxs]
(8)                     polygon = [sx, sy / 3] * np.array(
(12)                        [[.5, -.5], [.5, .5], [0., 1.], [-.5, .5], [-.5, -.5], [0., -1.]])
(8)                     if linewidths is None:
(12)                        linewidths = [mpl.rcParams['patch.linewidth']]
(8)                     if xscale == 'log' or yscale == 'log':
(12)                        polygons = np.expand_dims(polygon, 0) + np.expand_dims(offsets, 1)
(12)                        if xscale == 'log':
(16)                            polygons[:, :, 0] = 10.0 ** polygons[:, :, 0]
(16)                            xmin = 10.0 ** xmin
(16)                            xmax = 10.0 ** xmax
(16)                            self.set_xscale(xscale)
(12)                        if yscale == 'log':
(16)                            polygons[:, :, 1] = 10.0 ** polygons[:, :, 1]
(16)                            ymin = 10.0 ** ymin
(16)                            ymax = 10.0 ** ymax
(16)                            self.set_yscale(yscale)
(12)                        collection = mcoll.PolyCollection(
(16)                            polygons,
(16)                            edgecolors=edgecolors,
(16)                            linewidths=linewidths,
(16)                            )
(8)                     else:
(12)                        collection = mcoll.PolyCollection(
(16)                            [polygon],
(16)                            edgecolors=edgecolors,
(16)                            linewidths=linewidths,
(16)                            offsets=offsets,
(16)                            offset_transform=mtransforms.AffineDeltaTransform(
(20)                                self.transData),
(12)                        )
(8)                     if cbook._str_equal(bins, 'log'):
(12)                        if norm is not None:
(16)                            _api.warn_external("Only one of 'bins' and 'norm' arguments "
(35)                                               f"can be supplied, ignoring {bins=}")
(12)                        else:
(16)                            norm = mcolors.LogNorm(vmin=vmin, vmax=vmax)
(16)                            vmin = vmax = None
(12)                        bins = None
(8)                     if norm is not None:
(12)                        if norm.vmin is None and norm.vmax is None:
(16)                            norm.autoscale(accum)
(8)                     if bins is not None:
(12)                        if not np.iterable(bins):
(16)                            minimum, maximum = min(accum), max(accum)
(16)                            bins -= 1  # one less edge than bins
(16)                            bins = minimum + (maximum - minimum) * np.arange(bins) / bins
(12)                        bins = np.sort(bins)
(12)                        accum = bins.searchsorted(accum)
(8)                     collection.set_array(accum)
(8)                     collection.set_cmap(cmap)
(8)                     collection.set_norm(norm)
(8)                     collection.set_alpha(alpha)
(8)                     collection._internal_update(kwargs)
(8)                     collection._scale_norm(norm, vmin, vmax)
(8)                     corners = ((xmin, ymin), (xmax, ymax))
(8)                     self.update_datalim(corners)
```

```
(8)                 self._request_autoscale_view(tight=True)
(8)                 self.add_collection(collection, autolim=False)
(8)                 if not marginals:
(12)                    return collection
(8)                 bars = []
(8)                 for zname, z, zmin, zmax, zscale, nbins in [
(16)                        ("x", x, xmin, xmax, xscale, nx),
(16)                        ("y", y, ymin, ymax, yscale, 2 * ny),
(8)                 ]:
(12)                    if zscale == "log":
(16)                        bin_edges = np.geomspace(zmin, zmax, nbins + 1)
(12)                    else:
(16)                        bin_edges = np.linspace(zmin, zmax, nbins + 1)
(12)                    verts = np.empty((nbins, 4, 2))
(12)                    verts[:, 0, 0] = verts[:, 1, 0] = bin_edges[:-1]
(12)                    verts[:, 2, 0] = verts[:, 3, 0] = bin_edges[1:]
(12)                    verts[:, 0, 1] = verts[:, 3, 1] = .00
(12)                    verts[:, 1, 1] = verts[:, 2, 1] = .05
(12)                    if zname == "y":
(16)                        verts = verts[:, :, ::-1]  # Swap x and y.
(12)                    bin_idxs = np.searchsorted(bin_edges, z) - 1
(12)                    values = np.empty(nbins)
(12)                    for i in range(nbins):
(16)                        ci = C[bin_idxs == i]
(16)                        values[i] = reduce_C_function(ci) if len(ci) > 0 else np.nan
(12)                    mask = ~np.isnan(values)
(12)                    verts = verts[mask]
(12)                    values = values[mask]
(12)                    trans = getattr(self, f"get_{zname}axis_transform")(which="grid")
(12)                    bar = mcoll.PolyCollection(
(16)                        verts, transform=trans, edgecolors="face")
(12)                    bar.set_array(values)
(12)                    bar.set_cmap(cmap)
(12)                    bar.set_norm(norm)
(12)                    bar.set_alpha(alpha)
(12)                    bar._internal_update(kwargs)
(12)                    bars.append(self.add_collection(bar, autolim=False))
(8)                 collection.hbar, collection.vbar = bars
(8)                 def on_changed(collection):
(12)                    collection.hbar.set_cmap(collection.get_cmap())
(12)                    collection.hbar.set_cmap(collection.get_cmap())
(12)                    collection.vbar.set_clim(collection.get_clim())
(12)                    collection.vbar.set_clim(collection.get_clim())
(8)                 collection.callbacks.connect('changed', on_changed)
(8)                 return collection
(4)             @_docstring.dedent_interpd
(4)             def arrow(self, x, y, dx, dy, **kwargs):
(8)                 """
(8)                 Add an arrow to the Axes.
(8)                 This draws an arrow from ``(x, y)`` to ``(x+dx, y+dy)``.
(8)                 Parameters
(8)                 ----------
(8)                 %(FancyArrow)s
(8)                 Returns
(8)                 -------
(8)                 `.FancyArrow`
(12)                    The created `.FancyArrow` object.
(8)                 Notes
(8)                 -----
(8)                 The resulting arrow is affected by the Axes aspect ratio and limits.
(8)                 This may produce an arrow whose head is not square with its stem. To
(8)                 create an arrow whose head is square with its stem,
(8)                 use :meth:`annotate` for example:
(8)                 >>> ax.annotate("", xy=(0.5, 0.5), xytext=(0, 0),
(8)                 ...             arrowprops=dict(arrowstyle="->"))
(8)                 """
(8)                 x = self.convert_xunits(x)
(8)                 y = self.convert_yunits(y)
(8)                 dx = self.convert_xunits(dx)
```

```
(8)                 dy = self.convert_yunits(dy)
(8)                 a = mpatches.FancyArrow(x, y, dx, dy, **kwargs)
(8)                 self.add_patch(a)
(8)                 self._request_autoscale_view()
(8)                 return a
(4)             @_docstring.copy(mquiver.QuiverKey.__init__)
(4)             def quiverkey(self, Q, X, Y, U, label, **kwargs):
(8)                 qk = mquiver.QuiverKey(Q, X, Y, U, label, **kwargs)
(8)                 self.add_artist(qk)
(8)                 return qk
(4)             def _quiver_units(self, args, kwargs):
(8)                 if len(args) > 3:
(12)                    x, y = args[0:2]
(12)                    x, y = self._process_unit_info([("x", x), ("y", y)], kwargs)
(12)                    return (x, y) + args[2:]
(8)                 return args
(4)             @_preprocess_data()
(4)             @_docstring.dedent_interpd
(4)             def quiver(self, *args, **kwargs):
(8)                 """%(quiver_doc)s"""
(8)                 args = self._quiver_units(args, kwargs)
(8)                 q = mquiver.Quiver(self, *args, **kwargs)
(8)                 self.add_collection(q, autolim=True)
(8)                 self._request_autoscale_view()
(8)                 return q
(4)             @_preprocess_data()
(4)             @_docstring.dedent_interpd
(4)             def barbs(self, *args, **kwargs):
(8)                 """%(barbs_doc)s"""
(8)                 args = self._quiver_units(args, kwargs)
(8)                 b = mquiver.Barbs(self, *args, **kwargs)
(8)                 self.add_collection(b, autolim=True)
(8)                 self._request_autoscale_view()
(8)                 return b
(4)             def fill(self, *args, data=None, **kwargs):
(8)                 """
(8)                 Plot filled polygons.
(8)                 Parameters
(8)                 ----------
(8)                 *args : sequence of x, y, [color]
(12)                    Each polygon is defined by the lists of *x* and *y* positions of
(12)                    its nodes, optionally followed by a *color* specifier. See
(12)                    :mod:`matplotlib.colors` for supported color specifiers. The
(12)                    standard color cycle is used for polygons without a color
(12)                    specifier.
(12)                    You can plot multiple polygons by providing multiple *x*, *y*,
(12)                    *[color]* groups.
(12)                    For example, each of the following is legal::
(16)                        ax.fill(x, y)                    # a polygon with default
color
(16)                        ax.fill(x, y, "b")               # a blue polygon
(16)                        ax.fill(x, y, x2, y2)            # two polygons
(16)                        ax.fill(x, y, "b", x2, y2, "r")  # a blue and a red polygon
(8)                 data : indexable object, optional
(12)                    An object with labelled data. If given, provide the label names to
(12)                    plot in *x* and *y*, e.g.::
(16)                        ax.fill("time", "signal",
(24)                                data={"time": [0, 1, 2], "signal": [0, 1, 0]})
(8)                 Returns
(8)                 -------
(8)                 list of `~matplotlib.patches.Polygon`
(8)                 Other Parameters
(8)                 ----------------
(8)                 **kwargs : `~matplotlib.patches.Polygon` properties
(8)                 Notes
(8)                 -----
(8)                 Use :meth:`fill_between` if you would like to fill the region between
(8)                 two curves.
(8)                 """
```

```
 4189: (8)                    kwargs = cbook.normalize_kwargs(kwargs, mlines.Line2D)
 4190: (8)                    patches = [*self._get_patches_for_fill(self, *args, data=data,
 **kwargs)]
 4191: (8)                    for poly in patches:
 4192: (12)                       self.add_patch(poly)
 4193: (8)                    self._request_autoscale_view()
 4194: (8)                    return patches
 4195: (4)                def _fill_between_x_or_y(
 4196: (12)                       self, ind_dir, ind, dep1, dep2=0, *,
 4197: (12)                       where=None, interpolate=False, step=None, **kwargs):
 4198: (8)                    """
 4199: (8)                    Fill the area between two {dir} curves.
 4200: (8)                    The curves are defined by the points (*{ind}*, *{dep}1*) and (*{ind}*,
 4201: (8)                    *{dep}2*).  This creates one or multiple polygons describing the
 filled
 4202: (8)                    area.
 4203: (8)                    You may exclude some {dir} sections from filling using *where*.
 4204: (8)                    By default, the edges connect the given points directly.  Use *step*
 4205: (8)                    if the filling should be a step function, i.e. constant in between
 4206: (8)                    *{ind}*.
 4207: (8)                    Parameters
 4208: (8)                    ----------
 4209: (8)                    {ind} : array (length N)
 4210: (12)                       The {ind} coordinates of the nodes defining the curves.
 4211: (8)                    {dep}1 : array (length N) or scalar
 4212: (12)                       The {dep} coordinates of the nodes defining the first curve.
 4213: (8)                    {dep}2 : array (length N) or scalar, default: 0
 4214: (12)                       The {dep} coordinates of the nodes defining the second curve.
 4215: (8)                    where : array of bool (length N), optional
 4216: (12)                       Define *where* to exclude some {dir} regions from being filled.
 4217: (12)                       The filled regions are defined by the coordinates ``{ind}
 [where]``.
 4218: (12)                       More precisely, fill between ``{ind}[i]`` and ``{ind}[i+1]`` if
 4219: (12)                       ``where[i] and where[i+1]``.  Note that this definition implies
 4220: (12)                       that an isolated *True* value between two *False* values in
 *where*
 4221: (12)                       will not result in filling.  Both sides of the *True* position
 4222: (12)                       remain unfilled due to the adjacent *False* values.
 4223: (8)                    interpolate : bool, default: False
 4224: (12)                       This option is only relevant if *where* is used and the two curves
 4225: (12)                       are crossing each other.
 4226: (12)                       Semantically, *where* is often used for *{dep}1* > *{dep}2* or
 4227: (12)                       similar.  By default, the nodes of the polygon defining the filled
 4228: (12)                       region will only be placed at the positions in the *{ind}* array.
 4229: (12)                       Such a polygon cannot describe the above semantics close to the
 4230: (12)                       intersection.  The {ind}-sections containing the intersection are
 4231: (12)                       simply clipped.
 4232: (12)                       Setting *interpolate* to *True* will calculate the actual
 4233: (12)                       intersection point and extend the filled region up to this point.
 4234: (8)                    step : {{'pre', 'post', 'mid'}}, optional
 4235: (12)                       Define *step* if the filling should be a step function,
 4236: (12)                       i.e. constant in between *{ind}*.  The value determines where the
 4237: (12)                       step will occur:
 4238: (12)                       - 'pre': The y value is continued constantly to the left from
 4239: (14)                         every *x* position, i.e. the interval ``(x[i-1], x[i]]`` has the
 4240: (14)                         value ``y[i]``.
 4241: (12)                       - 'post': The y value is continued constantly to the right from
 4242: (14)                         every *x* position, i.e. the interval ``[x[i], x[i+1])`` has the
 4243: (14)                         value ``y[i]``.
 4244: (12)                       - 'mid': Steps occur half-way between the *x* positions.
 4245: (8)                    Returns
 4246: (8)                    -------
 4247: (8)                    `.PolyCollection`
 4248: (12)                       A `.PolyCollection` containing the plotted polygons.
 4249: (8)                    Other Parameters
 4250: (8)                    ----------------
 4251: (8)                    data : indexable object, optional
 4252: (12)                       DATA_PARAMETER_PLACEHOLDER
 4253: (8)                    **kwargs
```

```
(12)                        All other keyword arguments are passed on to `.PolyCollection`.
(12)                        They control the `.Polygon` properties:
(12)                        %(PolyCollection:kwdoc)s
(8)                     See Also
(8)                     --------
(8)                     fill_between : Fill between two sets of y-values.
(8)                     fill_betweenx : Fill between two sets of x-values.
(8)                     """
(8)                     dep_dir = {"x": "y", "y": "x"}[ind_dir]
(8)                     if not mpl.rcParams["_internal.classic_mode"]:
(12)                        kwargs = cbook.normalize_kwargs(kwargs, mcoll.Collection)
(12)                        if not any(c in kwargs for c in ("color", "facecolor")):
(16)                            kwargs["facecolor"] = \
(20)                                self._get_patches_for_fill.get_next_color()
(8)                     ind, dep1, dep2 = map(
(12)                        ma.masked_invalid, self._process_unit_info(
(16)                            [(ind_dir, ind), (dep_dir, dep1), (dep_dir, dep2)], kwargs))
(8)                     for name, array in [
(16)                            (ind_dir, ind), (f"{dep_dir}1", dep1), (f"{dep_dir}2", dep2)]:
(12)                        if array.ndim > 1:
(16)                            raise ValueError(f"{name!r} is not 1-dimensional")
(8)                     if where is None:
(12)                        where = True
(8)                     else:
(12)                        where = np.asarray(where, dtype=bool)
(12)                        if where.size != ind.size:
(16)                            raise ValueError(f"where size ({where.size}) does not match "
(33)                                             f"{ind_dir} size ({ind.size})")
(8)                     where = where & ~functools.reduce(
(12)                        np.logical_or, map(np.ma.getmaskarray, [ind, dep1, dep2]))
(8)                     ind, dep1, dep2 = np.broadcast_arrays(
(12)                        np.atleast_1d(ind), dep1, dep2, subok=True)
(8)                     polys = []
(8)                     for idx0, idx1 in cbook.contiguous_regions(where):
(12)                        indslice = ind[idx0:idx1]
(12)                        dep1slice = dep1[idx0:idx1]
(12)                        dep2slice = dep2[idx0:idx1]
(12)                        if step is not None:
(16)                            step_func = cbook.STEP_LOOKUP_MAP["steps-" + step]
(16)                            indslice, dep1slice, dep2slice = \
(20)                                step_func(indslice, dep1slice, dep2slice)
(12)                        if not len(indslice):
(16)                            continue
(12)                        N = len(indslice)
(12)                        pts = np.zeros((2 * N + 2, 2))
(12)                        if interpolate:
(16)                            def get_interp_point(idx):
(20)                                im1 = max(idx - 1, 0)
(20)                                ind_values = ind[im1:idx+1]
(20)                                diff_values = dep1[im1:idx+1] - dep2[im1:idx+1]
(20)                                dep1_values = dep1[im1:idx+1]
(20)                                if len(diff_values) == 2:
(24)                                    if np.ma.is_masked(diff_values[1]):
(28)                                        return ind[im1], dep1[im1]
(24)                                    elif np.ma.is_masked(diff_values[0]):
(28)                                        return ind[idx], dep1[idx]
(20)                                diff_order = diff_values.argsort()
(20)                                diff_root_ind = np.interp(
(24)                                    0, diff_values[diff_order], ind_values[diff_order])
(20)                                ind_order = ind_values.argsort()
(20)                                diff_root_dep = np.interp(
(24)                                    diff_root_ind,
(24)                                    ind_values[ind_order], dep1_values[ind_order])
(20)                                return diff_root_ind, diff_root_dep
(16)                            start = get_interp_point(idx0)
(16)                            end = get_interp_point(idx1)
(12)                        else:
(16)                            start = indslice[0], dep2slice[0]
(16)                            end = indslice[-1], dep2slice[-1]
```

```
4323: (12)                      pts[0] = start
4324: (12)                      pts[N + 1] = end
4325: (12)                      pts[1:N+1, 0] = indslice
4326: (12)                      pts[1:N+1, 1] = dep1slice
4327: (12)                      pts[N+2:, 0] = indslice[::-1]
4328: (12)                      pts[N+2:, 1] = dep2slice[::-1]
4329: (12)                      if ind_dir == "y":
4330: (16)                          pts = pts[:, ::-1]
4331: (12)                      polys.append(pts)
4332: (8)                   collection = mcoll.PolyCollection(polys, **kwargs)
4333: (8)                   pts = np.vstack([np.hstack([ind[where, None], dep1[where, None]]),
4334: (25)                                   np.hstack([ind[where, None], dep2[where, None]])])
4335: (8)                   if ind_dir == "y":
4336: (12)                      pts = pts[:, ::-1]
4337: (8)                   up_x = up_y = True
4338: (8)                   if "transform" in kwargs:
4339: (12)                      up_x, up_y =
kwargs["transform"].contains_branch_seperately(self.transData)
4340: (8)                   self.update_datalim(pts, updatex=up_x, updatey=up_y)
4341: (8)                   self.add_collection(collection, autolim=False)
4342: (8)                   self._request_autoscale_view()
4343: (8)                   return collection
4344: (4)               def fill_between(self, x, y1, y2=0, where=None, interpolate=False,
4345: (21)                               step=None, **kwargs):
4346: (8)                   return self._fill_between_x_or_y(
4347: (12)                      "x", x, y1, y2,
4348: (12)                      where=where, interpolate=interpolate, step=step, **kwargs)
4349: (4)               if _fill_between_x_or_y.__doc__:
4350: (8)                   fill_between.__doc__ = _fill_between_x_or_y.__doc__.format(
4351: (12)                      dir="horizontal", ind="x", dep="y"
4352: (8)                   )
4353: (4)               fill_between = _preprocess_data(
4354: (8)                   _docstring.dedent_interpd(fill_between),
4355: (8)                   replace_names=["x", "y1", "y2", "where"])
4356: (4)               def fill_betweenx(self, y, x1, x2=0, where=None,
4357: (22)                                step=None, interpolate=False, **kwargs):
4358: (8)                   return self._fill_between_x_or_y(
4359: (12)                      "y", y, x1, x2,
4360: (12)                      where=where, interpolate=interpolate, step=step, **kwargs)
4361: (4)               if _fill_between_x_or_y.__doc__:
4362: (8)                   fill_betweenx.__doc__ = _fill_between_x_or_y.__doc__.format(
4363: (12)                      dir="vertical", ind="y", dep="x"
4364: (8)                   )
4365: (4)               fill_betweenx = _preprocess_data(
4366: (8)                   _docstring.dedent_interpd(fill_betweenx),
4367: (8)                   replace_names=["y", "x1", "x2", "where"])
4368: (4)               @_preprocess_data()
4369: (4)               @_docstring.interpd
4370: (4)               def imshow(self, X, cmap=None, norm=None, *, aspect=None,
4371: (15)                         interpolation=None, alpha=None,
4372: (15)                         vmin=None, vmax=None, origin=None, extent=None,
4373: (15)                         interpolation_stage=None, filternorm=True, filterrad=4.0,
4374: (15)                         resample=None, url=None, **kwargs):
4375: (8)                   """
4376: (8)                   Display data as an image, i.e., on a 2D regular raster.
4377: (8)                   The input may either be actual RGB(A) data, or 2D scalar data, which
4378: (8)                   will be rendered as a pseudocolor image. For displaying a grayscale
4379: (8)                   image, set up the colormapping using the parameters
4380: (8)                   ``cmap='gray', vmin=0, vmax=255``.
4381: (8)                   The number of pixels used to render an image is set by the Axes size
4382: (8)                   and the figure *dpi*. This can lead to aliasing artifacts when
4383: (8)                   the image is resampled, because the displayed image size will usually
4384: (8)                   not match the size of *X* (see
4385: (8)                   :doc:`/gallery/images_contours_and_fields/image_antialiasing`).
4386: (8)                   The resampling can be controlled via the *interpolation* parameter
4387: (8)                   and/or :rc:`image.interpolation`.
4388: (8)                   Parameters
4389: (8)                   ----------
4390: (8)                   X : array-like or PIL image
```

```
(12)                        The image data. Supported array shapes are:
(12)                        - (M, N): an image with scalar data. The values are mapped to
(14)                          colors using normalization and a colormap. See parameters
*norm*,
(14)                          *cmap*, *vmin*, *vmax*.
(12)                        - (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
(12)                        - (M, N, 4): an image with RGBA values (0-1 float or 0-255 int),
(14)                          i.e. including transparency.
(12)                        The first two dimensions (M, N) define the rows and columns of
(12)                        the image.
(12)                        Out-of-range RGB(A) values are clipped.
(8)                     %(cmap_doc)s
(12)                        This parameter is ignored if *X* is RGB(A).
(8)                     %(norm_doc)s
(12)                        This parameter is ignored if *X* is RGB(A).
(8)                     %(vmin_vmax_doc)s
(12)                        This parameter is ignored if *X* is RGB(A).
(8)                     aspect : {'equal', 'auto'} or float or None, default: None
(12)                        The aspect ratio of the Axes.  This parameter is particularly
(12)                        relevant for images since it determines whether data pixels are
(12)                        square.
(12)                        This parameter is a shortcut for explicitly calling
(12)                        `.Axes.set_aspect`. See there for further details.
(12)                        - 'equal': Ensures an aspect ratio of 1. Pixels will be square
(14)                          (unless pixel sizes are explicitly made non-square in data
(14)                          coordinates using *extent*).
(12)                        - 'auto': The Axes is kept fixed and the aspect is adjusted so
(14)                          that the data fit in the Axes. In general, this will result in
(14)                          non-square pixels.
(12)                        Normally, None (the default) means to use :rc:`image.aspect`.
However, if
(12)                        the image uses a transform that does not contain the axes data
transform,
(12)                        then None means to not modify the axes aspect at all (in that
case, directly
(12)                        call `.Axes.set_aspect` if desired).
(8)                     interpolation : str, default: :rc:`image.interpolation`
(12)                        The interpolation method used.
(12)                        Supported values are 'none', 'antialiased', 'nearest', 'bilinear',
(12)                        'bicubic', 'spline16', 'spline36', 'hanning', 'hamming',
'hermite',
(12)                        'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell',
(12)                        'sinc', 'lanczos', 'blackman'.
(12)                        The data *X* is resampled to the pixel size of the image on the
(12)                        figure canvas, using the interpolation method to either up- or
(12)                        downsample the data.
(12)                        If *interpolation* is 'none', then for the ps, pdf, and svg
(12)                        backends no down- or upsampling occurs, and the image data is
(12)                        passed to the backend as a native image.  Note that different ps,
(12)                        pdf, and svg viewers may display these raw pixels differently. On
(12)                        other backends, 'none' is the same as 'nearest'.
(12)                        If *interpolation* is the default 'antialiased', then 'nearest'
(12)                        interpolation is used if the image is upsampled by more than a
(12)                        factor of three (i.e. the number of display pixels is at least
(12)                        three times the size of the data array).  If the upsampling rate
is
(12)                        smaller than 3, or the image is downsampled, then 'hanning'
(12)                        interpolation is used to act as an anti-aliasing filter, unless
the
(12)                        image happens to be upsampled by exactly a factor of two or one.
(12)                        See
(12)                        :doc:`/gallery/images_contours_and_fields/interpolation_methods`
(12)                        for an overview of the supported interpolation methods, and
(12)                        :doc:`/gallery/images_contours_and_fields/image_antialiasing` for
(12)                        a discussion of image antialiasing.
(12)                        Some interpolation methods require an additional radius parameter,
(12)                        which can be set by *filterrad*. Additionally, the antigrain image
(12)                        resize filter is controlled by the parameter *filternorm*.
(8)                     interpolation_stage : {'data', 'rgba'}, default: 'data'
```

```
(12)                        If 'data', interpolation
(12)                        is carried out on the data provided by the user.  If 'rgba', the
(12)                        interpolation is carried out after the colormapping has been
(12)                        applied (visual interpolation).
(8)                     alpha : float or array-like, optional
(12)                        The alpha blending value, between 0 (transparent) and 1 (opaque).
(12)                        If *alpha* is an array, the alpha blending values are applied
pixel
(12)                        by pixel, and *alpha* must have the same shape as *X*.
(8)                     origin : {'upper', 'lower'}, default: :rc:`image.origin`
(12)                        Place the [0, 0] index of the array in the upper left or lower
(12)                        left corner of the Axes. The convention (the default) 'upper' is
(12)                        typically used for matrices and images.
(12)                        Note that the vertical axis points upward for 'lower'
(12)                        but downward for 'upper'.
(12)                        See the :ref:`imshow_extent` tutorial for
(12)                        examples and a more detailed description.
(8)                     extent : floats (left, right, bottom, top), optional
(12)                        The bounding box in data coordinates that the image will fill.
(12)                        These values may be unitful and match the units of the Axes.
(12)                        The image is stretched individually along x and y to fill the box.
(12)                        The default extent is determined by the following conditions.
(12)                        Pixels have unit size in data coordinates. Their centers are on
(12)                        integer coordinates, and their center coordinates range from 0 to
(12)                        columns-1 horizontally and from 0 to rows-1 vertically.
(12)                        Note that the direction of the vertical axis and thus the default
(12)                        values for top and bottom depend on *origin*:
(12)                        - For ``origin == 'upper'`` the default is
(14)                          ``(-0.5, numcols-0.5, numrows-0.5, -0.5)``.
(12)                        - For ``origin == 'lower'`` the default is
(14)                          ``(-0.5, numcols-0.5, -0.5, numrows-0.5)``.
(12)                        See the :ref:`imshow_extent` tutorial for
(12)                        examples and a more detailed description.
(8)                     filternorm : bool, default: True
(12)                        A parameter for the antigrain image resize filter (see the
(12)                        antigrain documentation).  If *filternorm* is set, the filter
(12)                        normalizes integer values and corrects the rounding errors. It
(12)                        doesn't do anything with the source floating point values, it
(12)                        corrects only integers according to the rule of 1.0 which means
(12)                        that any sum of pixel weights must be equal to 1.0.  So, the
(12)                        filter function must produce a graph of the proper shape.
(8)                     filterrad : float > 0, default: 4.0
(12)                        The filter radius for filters that have a radius parameter, i.e.
(12)                        when interpolation is one of: 'sinc', 'lanczos' or 'blackman'.
(8)                     resample : bool, default: :rc:`image.resample`
(12)                        When *True*, use a full resampling method.  When *False*, only
(12)                        resample when the output image is larger than the input image.
(8)                     url : str, optional
(12)                        Set the url of the created `.AxesImage`. See `.Artist.set_url`.
(8)                     Returns
(8)                     -------
(8)                     `~matplotlib.image.AxesImage`
(8)                     Other Parameters
(8)                     ----------------
(8)                     data : indexable object, optional
(12)                        DATA_PARAMETER_PLACEHOLDER
(8)                     **kwargs : `~matplotlib.artist.Artist` properties
(12)                        These parameters are passed on to the constructor of the
(12)                        `.AxesImage` artist.
(8)                     See Also
(8)                     --------
(8)                     matshow : Plot a matrix or an array as an image.
(8)                     Notes
(8)                     -----
(8)                     Unless *extent* is used, pixel centers will be located at integer
(8)                     coordinates. In other words: the origin will coincide with the center
(8)                     of pixel (0, 0).
(8)                     There are two common representations for RGB images with an alpha
(8)                     channel:
```

```
4521: (8)                   -   Straight (unassociated) alpha: R, G, and B channels represent the
4522: (12)                      color of the pixel, disregarding its opacity.
4523: (8)                   -   Premultiplied (associated) alpha: R, G, and B channels represent
4524: (12)                      the color of the pixel, adjusted for its opacity by
multiplication.
4525: (8)                   `~matplotlib.pyplot.imshow` expects RGB images adopting the straight
4526: (8)                   (unassociated) alpha representation.
4527: (8)                   """
4528: (8)                   im = mimage.AxesImage(self, cmap=cmap, norm=norm,
4529: (30)                                        interpolation=interpolation, origin=origin,
4530: (30)                                        extent=extent, filternorm=filternorm,
4531: (30)                                        filterrad=filterrad, resample=resample,
4532: (30)                                        interpolation_stage=interpolation_stage,
4533: (30)                                        **kwargs)
4534: (8)                   if aspect is None and not (
4535: (16)                          im.is_transform_set()
4536: (16)                          and not im.get_transform().contains_branch(self.transData)):
4537: (12)                      aspect = mpl.rcParams['image.aspect']
4538: (8)                   if aspect is not None:
4539: (12)                      self.set_aspect(aspect)
4540: (8)                   im.set_data(X)
4541: (8)                   im.set_alpha(alpha)
4542: (8)                   if im.get_clip_path() is None:
4543: (12)                      im.set_clip_path(self.patch)
4544: (8)                   im._scale_norm(norm, vmin, vmax)
4545: (8)                   im.set_url(url)
4546: (8)                   im.set_extent(im.get_extent())
4547: (8)                   self.add_image(im)
4548: (8)                   return im
4549: (4)               def _pcolorargs(self, funcname, *args, shading='auto', **kwargs):
4550: (8)                   _valid_shading = ['gouraud', 'nearest', 'flat', 'auto']
4551: (8)                   try:
4552: (12)                      _api.check_in_list(_valid_shading, shading=shading)
4553: (8)                   except ValueError:
4554: (12)                      _api.warn_external(f"shading value '{shading}' not in list of "
4555: (31)                                         f"valid values {_valid_shading}. Setting "
4556: (31)                                         "shading='auto'.")
4557: (12)                      shading = 'auto'
4558: (8)                   if len(args) == 1:
4559: (12)                      C = np.asanyarray(args[0])
4560: (12)                      nrows, ncols = C.shape[:2]
4561: (12)                      if shading in ['gouraud', 'nearest']:
4562: (16)                          X, Y = np.meshgrid(np.arange(ncols), np.arange(nrows))
4563: (12)                      else:
4564: (16)                          X, Y = np.meshgrid(np.arange(ncols + 1), np.arange(nrows + 1))
4565: (16)                          shading = 'flat'
4566: (12)                      C = cbook.safe_masked_invalid(C, copy=True)
4567: (12)                      return X, Y, C, shading
4568: (8)                   if len(args) == 3:
4569: (12)                      C = np.asanyarray(args[2])
4570: (12)                      X, Y = args[:2]
4571: (12)                      X, Y = self._process_unit_info([("x", X), ("y", Y)], kwargs)
4572: (12)                      X, Y = [cbook.safe_masked_invalid(a, copy=True) for a in [X, Y]]
4573: (12)                      if funcname == 'pcolormesh':
4574: (16)                          if np.ma.is_masked(X) or np.ma.is_masked(Y):
4575: (20)                              raise ValueError(
4576: (24)                                  'x and y arguments to pcolormesh cannot have '
4577: (24)                                  'non-finite values or be of type '
4578: (24)                                  'numpy.ma.MaskedArray with masked values')
4579: (12)                      nrows, ncols = C.shape[:2]
4580: (8)                   else:
4581: (12)                      raise _api.nargs_error(funcname, takes="1 or 3", given=len(args))
4582: (8)                   Nx = X.shape[-1]
4583: (8)                   Ny = Y.shape[0]
4584: (8)                   if X.ndim != 2 or X.shape[0] == 1:
4585: (12)                      x = X.reshape(1, Nx)
4586: (12)                      X = x.repeat(Ny, axis=0)
4587: (8)                   if Y.ndim != 2 or Y.shape[1] == 1:
4588: (12)                      y = Y.reshape(Ny, 1)
```

```
4589: (12)                        Y = y.repeat(Nx, axis=1)
4590: (8)                     if X.shape != Y.shape:
4591: (12)                        raise TypeError(f'Incompatible X, Y inputs to {funcname}; '
4592: (28)                                        f'see help({funcname})')
4593: (8)                     if shading == 'auto':
4594: (12)                        if ncols == Nx and nrows == Ny:
4595: (16)                            shading = 'nearest'
4596: (12)                        else:
4597: (16)                            shading = 'flat'
4598: (8)                     if shading == 'flat':
4599: (12)                        if (Nx, Ny) != (ncols + 1, nrows + 1):
4600: (16)                            raise TypeError(f"Dimensions of C {C.shape} should"
4601: (32)                                            f" be one smaller than X({Nx}) and Y({Ny})"
4602: (32)                                            f" while using shading='flat'"
4603: (32)                                            f" see help({funcname})")
4604: (8)                     else:    # ['nearest', 'gouraud']:
4605: (12)                        if (Nx, Ny) != (ncols, nrows):
4606: (16)                            raise TypeError('Dimensions of C %s are incompatible with'
4607: (32)                                            ' X (%d) and/or Y (%d); see help(%s)' % (
4608: (36)                                                C.shape, Nx, Ny, funcname))
4609: (12)                        if shading == 'nearest':
4610: (16)                            def _interp_grid(X):
4611: (20)                                if np.shape(X)[1] > 1:
4612: (24)                                    dX = np.diff(X, axis=1) * 0.5
4613: (24)                                    if not (np.all(dX >= 0) or np.all(dX <= 0)):
4614: (28)                                        _api.warn_external(
4615: (32)                                            f"The input coordinates to {funcname} are "
4616: (32)                                            "interpreted as cell centers, but are not "
4617: (32)                                            "monotonically increasing or decreasing. "
4618: (32)                                            "This may lead to incorrectly calculated cell
"
4619: (32)                                            "edges, in which case, please supply "
4620: (32)                                            f"explicit cell edges to {funcname}.")
4621: (24)                                    hstack = np.ma.hstack if np.ma.isMA(X) else np.hstack
4622: (24)                                    X = hstack((X[:, [0]] - dX[:, [0]],
4623: (36)                                                X[:, :-1] + dX,
4624: (36)                                                X[:, [-1]] + dX[:, [-1]]))
4625: (20)                                else:
4626: (24)                                    X = np.hstack((X, X))
4627: (20)                                return X
4628: (16)                            if ncols == Nx:
4629: (20)                                X = _interp_grid(X)
4630: (20)                                Y = _interp_grid(Y)
4631: (16)                            if nrows == Ny:
4632: (20)                                X = _interp_grid(X.T).T
4633: (20)                                Y = _interp_grid(Y.T).T
4634: (16)                            shading = 'flat'
4635: (8)                     C = cbook.safe_masked_invalid(C, copy=True)
4636: (8)                     return X, Y, C, shading
4637: (4)                 @_preprocess_data()
4638: (4)                 @_docstring.dedent_interpd
4639: (4)                 def pcolor(self, *args, shading=None, alpha=None, norm=None, cmap=None,
4640: (15)                           vmin=None, vmax=None, **kwargs):
4641: (8)                     r"""
4642: (8)                     Create a pseudocolor plot with a non-regular rectangular grid.
4643: (8)                     Call signature::
4644: (12)                        pcolor([X, Y,] C, **kwargs)
4645: (8)                     *X* and *Y* can be used to specify the corners of the quadrilaterals.
4646: (8)                     .. hint::
4647: (12)                        ``pcolor()`` can be very slow for large arrays. In most
4648: (12)                        cases you should use the similar but much faster
4649: (12)                        `~.Axes.pcolormesh` instead. See
4650: (12)                        :ref:`Differences between pcolor() and pcolormesh()
4651: (12)                        <differences-pcolor-pcolormesh>` for a discussion of the
4652: (12)                        differences.
4653: (8)                     Parameters
4654: (8)                     ----------
4655: (8)                     C : 2D array-like
4656: (12)                        The color-mapped values.  Color-mapping is controlled by *cmap*,
```

```
4657: (12)                      *norm*, *vmin*, and *vmax*.
4658: (8)                   X, Y : array-like, optional
4659: (12)                      The coordinates of the corners of quadrilaterals of a pcolormesh::
4660: (16)                          (X[i+1, j], Y[i+1, j])       (X[i+1, j+1], Y[i+1, j+1])
4661: (38)                                                 ●────●
4662: (38)                                                 │    │
4663: (38)                                                 ●────●
4664: (20)                                  (X[i, j], Y[i, j])       (X[i, j+1], Y[i, j+1])
4665: (12)                      Note that the column index corresponds to the x-coordinate, and
4666: (12)                      the row index corresponds to y. For details, see the
4667: (12)                      :ref:`Notes <axes-pcolormesh-grid-orientation>` section below.
4668: (12)                      If ``shading='flat'`` the dimensions of *X* and *Y* should be one
4669: (12)                      greater than those of *C*, and the quadrilateral is colored due
4670: (12)                      to the value at ``C[i, j]``.  If *X*, *Y* and *C* have equal
4671: (12)                      dimensions, a warning will be raised and the last row and column
4672: (12)                      of *C* will be ignored.
4673: (12)                      If ``shading='nearest'``, the dimensions of *X* and *Y* should be
4674: (12)                      the same as those of *C* (if not, a ValueError will be raised).
The
4675: (12)                      color ``C[i, j]`` will be centered on ``(X[i, j], Y[i, j])``.
4676: (12)                      If *X* and/or *Y* are 1-D arrays or column vectors they will be
4677: (12)                      expanded as needed into the appropriate 2D arrays, making a
4678: (12)                      rectangular grid.
4679: (8)                   shading : {'flat', 'nearest', 'auto'}, default: :rc:`pcolor.shading`
4680: (12)                      The fill style for the quadrilateral. Possible values:
4681: (12)                      - 'flat': A solid color is used for each quad. The color of the
4682: (14)                        quad (i, j), (i+1, j), (i, j+1), (i+1, j+1) is given by
4683: (14)                        ``C[i, j]``. The dimensions of *X* and *Y* should be
4684: (14)                        one greater than those of *C*; if they are the same as *C*,
4685: (14)                        then a deprecation warning is raised, and the last row
4686: (14)                        and column of *C* are dropped.
4687: (12)                      - 'nearest': Each grid point will have a color centered on it,
4688: (14)                        extending halfway between the adjacent grid centers.  The
4689: (14)                        dimensions of *X* and *Y* must be the same as *C*.
4690: (12)                      - 'auto': Choose 'flat' if dimensions of *X* and *Y* are one
4691: (14)                        larger than *C*.  Choose 'nearest' if dimensions are the same.
4692: (12)                      See :doc:`/gallery/images_contours_and_fields/pcolormesh_grids`
4693: (12)                      for more description.
4694: (8)                   %(cmap_doc)s
4695: (8)                   %(norm_doc)s
4696: (8)                   %(vmin_vmax_doc)s
4697: (8)                   edgecolors : {'none', None, 'face', color, color sequence}, optional
4698: (12)                      The color of the edges. Defaults to 'none'. Possible values:
4699: (12)                      - 'none' or '': No edge.
4700: (12)                      - *None*: :rc:`patch.edgecolor` will be used. Note that currently
4701: (14)                        :rc:`patch.force_edgecolor` has to be True for this to work.
4702: (12)                      - 'face': Use the adjacent face color.
4703: (12)                      - A color or sequence of colors will set the edge color.
4704: (12)                      The singular form *edgecolor* works as an alias.
4705: (8)                   alpha : float, default: None
4706: (12)                      The alpha blending value of the face color, between 0
(transparent)
4707: (12)                      and 1 (opaque). Note: The edgecolor is currently not affected by
4708: (12)                      this.
4709: (8)                   snap : bool, default: False
4710: (12)                      Whether to snap the mesh to pixel boundaries.
4711: (8)                   Returns
4712: (8)                   -------
4713: (8)                   `matplotlib.collections.PolyQuadMesh`
4714: (8)                   Other Parameters
4715: (8)                   ----------------
4716: (8)                   antialiaseds : bool, default: False
4717: (12)                      The default *antialiaseds* is False if the default
4718: (12)                      *edgecolors*\ ="none" is used.  This eliminates artificial lines
4719: (12)                      at patch boundaries, and works regardless of the value of alpha.
4720: (12)                      If *edgecolors* is not "none", then the default *antialiaseds*
4721: (12)                      is taken from :rc:`patch.antialiased`.
4722: (12)                      Stroking the edges may be preferred if *alpha* is 1, but will
4723: (12)                      cause artifacts otherwise.
```

```
(8)                 data : indexable object, optional
(12)                    DATA_PARAMETER_PLACEHOLDER
(8)                 **kwargs
(12)                    Additionally, the following arguments are allowed. They are passed
(12)                    along to the `~matplotlib.collections.PolyQuadMesh` constructor:
(8)                 %(PolyCollection:kwdoc)s
(8)                 See Also
(8)                 --------
(8)                 pcolormesh : for an explanation of the differences between
(12)                    pcolor and pcolormesh.
(8)                 imshow : If *X* and *Y* are each equidistant, `~.Axes.imshow` can be a
(12)                    faster alternative.
(8)                 Notes
(8)                 -----
(8)                 **Masked arrays**
(8)                 *X*, *Y* and *C* may be masked arrays. If either ``C[i, j]``, or one
(8)                 of the vertices surrounding ``C[i, j]`` (*X* or *Y* at
(8)                 ``[i, j], [i+1, j], [i, j+1], [i+1, j+1]``) is masked, nothing is
(8)                 plotted.
(8)                 .. _axes-pcolor-grid-orientation:
(8)                 **Grid orientation**
(8)                 The grid orientation follows the standard matrix convention: An array
(8)                 *C* with shape (nrows, ncolumns) is plotted with the column number as
(8)                 *X* and the row number as *Y*.
(8)                 """
(8)                 if shading is None:
(12)                    shading = mpl.rcParams['pcolor.shading']
(8)                 shading = shading.lower()
(8)                 X, Y, C, shading = self._pcolorargs('pcolor', *args, shading=shading,
(44)                                                    kwargs=kwargs)
(8)                 linewidths = (0.25,)
(8)                 if 'linewidth' in kwargs:
(12)                    kwargs['linewidths'] = kwargs.pop('linewidth')
(8)                 kwargs.setdefault('linewidths', linewidths)
(8)                 if 'edgecolor' in kwargs:
(12)                    kwargs['edgecolors'] = kwargs.pop('edgecolor')
(8)                 ec = kwargs.setdefault('edgecolors', 'none')
(8)                 if 'antialiaseds' in kwargs:
(12)                    kwargs['antialiased'] = kwargs.pop('antialiaseds')
(8)                 if 'antialiased' not in kwargs and cbook._str_lower_equal(ec, "none"):
(12)                    kwargs['antialiased'] = False
(8)                 kwargs.setdefault('snap', False)
(8)                 if np.ma.isMaskedArray(X) or np.ma.isMaskedArray(Y):
(12)                    stack = np.ma.stack
(12)                    X = np.ma.asarray(X)
(12)                    Y = np.ma.asarray(Y)
(12)                    x = X.compressed()
(12)                    y = Y.compressed()
(8)                 else:
(12)                    stack = np.stack
(12)                    x = X
(12)                    y = Y
(8)                 coords = stack([X, Y], axis=-1)
(8)                 collection = mcoll.PolyQuadMesh(
(12)                    coords, array=C, cmap=cmap, norm=norm, alpha=alpha, **kwargs)
(8)                 collection._scale_norm(norm, vmin, vmax)
(8)                 t = collection._transform
(8)                 if (not isinstance(t, mtransforms.Transform) and
(16)                        hasattr(t, '_as_mpl_transform')):
(12)                    t = t._as_mpl_transform(self.axes)
(8)                 if t and any(t.contains_branch_seperately(self.transData)):
(12)                    trans_to_data = t - self.transData
(12)                    pts = np.vstack([x, y]).T.astype(float)
(12)                    transformed_pts = trans_to_data.transform(pts)
(12)                    x = transformed_pts[..., 0]
(12)                    y = transformed_pts[..., 1]
(8)                 self.add_collection(collection, autolim=False)
(8)                 minx = np.min(x)
(8)                 maxx = np.max(x)
```

```
        miny = np.min(y)
        maxy = np.max(y)
        collection.sticky_edges.x[:] = [minx, maxx]
        collection.sticky_edges.y[:] = [miny, maxy]
        corners = (minx, miny), (maxx, maxy)
        self.update_datalim(corners)
        self._request_autoscale_view()
        return collection
    @_preprocess_data()
    @_docstring.dedent_interpd
    def pcolormesh(self, *args, alpha=None, norm=None, cmap=None, vmin=None,
                   vmax=None, shading=None, antialiased=False, **kwargs):
        """
        Create a pseudocolor plot with a non-regular rectangular grid.
        Call signature::
            pcolormesh([X, Y,] C, **kwargs)
        *X* and *Y* can be used to specify the corners of the quadrilaterals.
        .. hint::
           `~.Axes.pcolormesh` is similar to `~.Axes.pcolor`. It is much
faster

           and preferred in most cases. For a detailed discussion on the
           differences see :ref:`Differences between pcolor() and pcolormesh()
           <differences-pcolor-pcolormesh>`.
        Parameters
        ----------
        C : array-like
            The mesh data. Supported array shapes are:
            - (M, N) or M*N: a mesh with scalar data. The values are mapped to
              colors using normalization and a colormap. See parameters
*norm*,

              *cmap*, *vmin*, *vmax*.
            - (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
            - (M, N, 4): an image with RGBA values (0-1 float or 0-255 int),
              i.e. including transparency.
            The first two dimensions (M, N) define the rows and columns of
            the mesh data.
        X, Y : array-like, optional
            The coordinates of the corners of quadrilaterals of a pcolormesh::
                (X[i+1, j], Y[i+1, j])       (X[i+1, j+1], Y[i+1, j+1])
                                              ●╶╶╶╶●
                                              ╎     ╎
                                              ●╶╶╶╶●
                        (X[i, j], Y[i, j])       (X[i, j+1], Y[i, j+1])
            Note that the column index corresponds to the x-coordinate, and
            the row index corresponds to y. For details, see the
            :ref:`Notes <axes-pcolormesh-grid-orientation>` section below.
            If ``shading='flat'`` the dimensions of *X* and *Y* should be one
            greater than those of *C*, and the quadrilateral is colored due
            to the value at ``C[i, j]``.  If *X*, *Y* and *C* have equal
            dimensions, a warning will be raised and the last row and column
            of *C* will be ignored.
            If ``shading='nearest'`` or ``'gouraud'``, the dimensions of *X*
            and *Y* should be the same as those of *C* (if not, a ValueError
            will be raised).  For ``'nearest'`` the color ``C[i, j]`` is
            centered on ``(X[i, j], Y[i, j])``.  For ``'gouraud'``, a smooth
            interpolation is carried out between the quadrilateral corners.
            If *X* and/or *Y* are 1-D arrays or column vectors they will be
            expanded as needed into the appropriate 2D arrays, making a
            rectangular grid.
        %(cmap_doc)s
        %(norm_doc)s
        %(vmin_vmax_doc)s
        edgecolors : {'none', None, 'face', color, color sequence}, optional
            The color of the edges. Defaults to 'none'. Possible values:
            - 'none' or '': No edge.
            - *None*: :rc:`patch.edgecolor` will be used. Note that currently
              :rc:`patch.force_edgecolor` has to be True for this to work.
            - 'face': Use the adjacent face color.
            - A color or sequence of colors will set the edge color.
```

```
(12)                    The singular form *edgecolor* works as an alias.
(8)                 alpha : float, default: None
(12)                    The alpha blending value, between 0 (transparent) and 1 (opaque).
(8)                 shading : {'flat', 'nearest', 'gouraud', 'auto'}, optional
(12)                    The fill style for the quadrilateral; defaults to
(12)                    :rc:`pcolor.shading`. Possible values:
(12)                    - 'flat': A solid color is used for each quad. The color of the
(14)                      quad (i, j), (i+1, j), (i, j+1), (i+1, j+1) is given by
(14)                      ``C[i, j]``. The dimensions of *X* and *Y* should be
(14)                      one greater than those of *C*; if they are the same as *C*,
(14)                      then a deprecation warning is raised, and the last row
(14)                      and column of *C* are dropped.
(12)                    - 'nearest': Each grid point will have a color centered on it,
(14)                      extending halfway between the adjacent grid centers.  The
(14)                      dimensions of *X* and *Y* must be the same as *C*.
(12)                    - 'gouraud': Each quad will be Gouraud shaded: The color of the
(14)                      corners (i', j') are given by ``C[i', j']``. The color values of
(14)                      the area in between is interpolated from the corner values.
(14)                      The dimensions of *X* and *Y* must be the same as *C*. When
(14)                      Gouraud shading is used, *edgecolors* is ignored.
(12)                    - 'auto': Choose 'flat' if dimensions of *X* and *Y* are one
(14)                      larger than *C*.  Choose 'nearest' if dimensions are the same.
(12)                    See :doc:`/gallery/images_contours_and_fields/pcolormesh_grids`
(12)                    for more description.
(8)                 snap : bool, default: False
(12)                    Whether to snap the mesh to pixel boundaries.
(8)                 rasterized : bool, optional
(12)                    Rasterize the pcolormesh when drawing vector graphics.  This can
(12)                    speed up rendering and produce smaller files for large data sets.
(12)                    See also :doc:`/gallery/misc/rasterization_demo`.
(8)                 Returns
(8)                 -------
(8)                 `matplotlib.collections.QuadMesh`
(8)                 Other Parameters
(8)                 ----------------
(8)                 data : indexable object, optional
(12)                    DATA_PARAMETER_PLACEHOLDER
(8)                 **kwargs
(12)                    Additionally, the following arguments are allowed. They are passed
(12)                    along to the `~matplotlib.collections.QuadMesh` constructor:
(8)                 %(QuadMesh:kwdoc)s
(8)                 See Also
(8)                 --------
(8)                 pcolor : An alternative implementation with slightly different
(12)                    features. For a detailed discussion on the differences see
(12)                    :ref:`Differences between pcolor() and pcolormesh()
(12)                    <differences-pcolor-pcolormesh>`.
(8)                 imshow : If *X* and *Y* are each equidistant, `~.Axes.imshow` can be a
(12)                    faster alternative.
(8)                 Notes
(8)                 -----
(8)                 **Masked arrays**
(8)                 *C* may be a masked array. If ``C[i, j]`` is masked, the corresponding
(8)                 quadrilateral will be transparent. Masking of *X* and *Y* is not
(8)                 supported. Use `~.Axes.pcolor` if you need this functionality.
(8)                 .. _axes-pcolormesh-grid-orientation:
(8)                 **Grid orientation**
(8)                 The grid orientation follows the standard matrix convention: An array
(8)                 *C* with shape (nrows, ncolumns) is plotted with the column number as
(8)                 *X* and the row number as *Y*.
(8)                 .. _differences-pcolor-pcolormesh:
(8)                 **Differences between pcolor() and pcolormesh()**
(8)                 Both methods are used to create a pseudocolor plot of a 2D array
(8)                 using quadrilaterals.
(8)                 The main difference lies in the created object and internal data
(8)                 handling:
(8)                 While `~.Axes.pcolor` returns a `.PolyQuadMesh`, `~.Axes.pcolormesh`
(8)                 returns a `.QuadMesh`. The latter is more specialized for the given
(8)                 purpose and thus is faster. It should almost always be preferred.
```

```
(8)                 There is also a slight difference in the handling of masked arrays.
(8)                 Both `~.Axes.pcolor` and `~.Axes.pcolormesh` support masked arrays
(8)                 for *C*. However, only `~.Axes.pcolor` supports masked arrays for *X*
(8)                 and *Y*. The reason lies in the internal handling of the masked values.
(8)                 `~.Axes.pcolor` leaves out the respective polygons from the
(8)                 PolyQuadMesh. `~.Axes.pcolormesh` sets the facecolor of the masked
(8)                 elements to transparent. You can see the difference when using
(8)                 edgecolors. While all edges are drawn irrespective of masking in a
(8)                 QuadMesh, the edge between two adjacent masked quadrilaterals in
(8)                 `~.Axes.pcolor` is not drawn as the corresponding polygons do not
(8)                 exist in the PolyQuadMesh. Because PolyQuadMesh draws each individual
(8)                 polygon, it also supports applying hatches and linestyles to the collection.
(8)                 Another difference is the support of Gouraud shading in
(8)                 `~.Axes.pcolormesh`, which is not available with `~.Axes.pcolor`.
(8)                 """
(8)                 if shading is None:
(12)                    shading = mpl.rcParams['pcolor.shading']
(8)                 shading = shading.lower()
(8)                 kwargs.setdefault('edgecolors', 'none')
(8)                 X, Y, C, shading = self._pcolorargs('pcolormesh', *args,
(44)                                                    shading=shading, kwargs=kwargs)
(8)                 coords = np.stack([X, Y], axis=-1)
(8)                 kwargs.setdefault('snap', mpl.rcParams['pcolormesh.snap'])
(8)                 collection = mcoll.QuadMesh(
(12)                    coords, antialiased=antialiased, shading=shading,
(12)                    array=C, cmap=cmap, norm=norm, alpha=alpha, **kwargs)
(8)                 collection._scale_norm(norm, vmin, vmax)
(8)                 coords = coords.reshape(-1, 2)  # flatten the grid structure; keep x, y
(8)                 t = collection._transform
(8)                 if (not isinstance(t, mtransforms.Transform) and
(16)                        hasattr(t, '_as_mpl_transform')):
(12)                    t = t._as_mpl_transform(self.axes)
(8)                 if t and any(t.contains_branch_seperately(self.transData)):
(12)                    trans_to_data = t - self.transData
(12)                    coords = trans_to_data.transform(coords)
(8)                 self.add_collection(collection, autolim=False)
(8)                 minx, miny = np.min(coords, axis=0)
(8)                 maxx, maxy = np.max(coords, axis=0)
(8)                 collection.sticky_edges.x[:] = [minx, maxx]
(8)                 collection.sticky_edges.y[:] = [miny, maxy]
(8)                 corners = (minx, miny), (maxx, maxy)
(8)                 self.update_datalim(corners)
(8)                 self._request_autoscale_view()
(8)                 return collection
(4)             @_preprocess_data()
(4)             @_docstring.dedent_interpd
(4)             def pcolorfast(self, *args, alpha=None, norm=None, cmap=None, vmin=None,
(19)                           vmax=None, **kwargs):
(8)                 """
(8)                 Create a pseudocolor plot with a non-regular rectangular grid.
(8)                 Call signature::
(10)                  ax.pcolorfast([X, Y], C, /, **kwargs)
(8)                 This method is similar to `~.Axes.pcolor` and `~.Axes.pcolormesh`.
(8)                 It's designed to provide the fastest pcolor-type plotting with the
(8)                 Agg backend. To achieve this, it uses different algorithms internally
(8)                 depending on the complexity of the input grid (regular rectangular,
(8)                 non-regular rectangular or arbitrary quadrilateral).
(8)                 .. warning::
(11)                   This method is experimental. Compared to `~.Axes.pcolor` or
(11)                   `~.Axes.pcolormesh` it has some limitations:
(11)                   - It supports only flat shading (no outlines)
(11)                   - It lacks support for log scaling of the axes.
(11)                   - It does not have a pyplot wrapper.
(8)                 Parameters
(8)                 ----------
(8)                 C : array-like
```

```
                        The image data. Supported array shapes are:
                        - (M, N): an image with scalar data.  Color-mapping is controlled
                          by *cmap*, *norm*, *vmin*, and *vmax*.
                        - (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
                        - (M, N, 4): an image with RGBA values (0-1 float or 0-255 int),
                          i.e. including transparency.
                        The first two dimensions (M, N) define the rows and columns of
                        the image.
                        This parameter can only be passed positionally.
                    X, Y : tuple or array-like, default: ``(0, N)``, ``(0, M)``
                        *X* and *Y* are used to specify the coordinates of the
                        quadrilaterals. There are different ways to do this:
                        - Use tuples ``X=(xmin, xmax)`` and ``Y=(ymin, ymax)`` to define
                          a *uniform rectangular grid*.
                          The tuples define the outer edges of the grid. All individual
                          quadrilaterals will be of the same size. This is the fastest
                          version.
                        - Use 1D arrays *X*, *Y* to specify a *non-uniform rectangular
                          grid*.
                          In this case *X* and *Y* have to be monotonic 1D arrays of
length

                          *N+1* and *M+1*, specifying the x and y boundaries of the cells.
                          The speed is intermediate. Note: The grid is checked, and if
                          found to be uniform the fast version is used.
                        - Use 2D arrays *X*, *Y* if you need an *arbitrary quadrilateral
                          grid* (i.e. if the quadrilaterals are not rectangular).
                          In this case *X* and *Y* are 2D arrays with shape (M + 1, N +
1),

                          specifying the x and y coordinates of the corners of the colored
                          quadrilaterals.
                          This is the most general, but the slowest to render.  It may
                          produce faster and more compact output using ps, pdf, and
                          svg backends, however.
                        These arguments can only be passed positionally.
                    %(cmap_doc)s
                        This parameter is ignored if *C* is RGB(A).
                    %(norm_doc)s
                        This parameter is ignored if *C* is RGB(A).
                    %(vmin_vmax_doc)s
                        This parameter is ignored if *C* is RGB(A).
                    alpha : float, default: None
                        The alpha blending value, between 0 (transparent) and 1 (opaque).
                    snap : bool, default: False
                        Whether to snap the mesh to pixel boundaries.
                    Returns
                    -------
                    `.AxesImage` or `.PcolorImage` or `.QuadMesh`
                        The return type depends on the type of grid:
                        - `.AxesImage` for a regular rectangular grid.
                        - `.PcolorImage` for a non-regular rectangular grid.
                        - `.QuadMesh` for a non-rectangular grid.
                    Other Parameters
                    ----------------
                    data : indexable object, optional
                        DATA_PARAMETER_PLACEHOLDER
                    **kwargs
                        Supported additional parameters depend on the type of grid.
                        See return types of *image* for further description.
                    """
                    C = args[-1]
                    nr, nc = np.shape(C)[:2]
                    if len(args) == 1:
                        style = "image"
                        x = [0, nc]
                        y = [0, nr]
                    elif len(args) == 3:
                        x, y = args[:2]
                        x = np.asarray(x)
                        y = np.asarray(y)
```

```
(12)                        if x.ndim == 1 and y.ndim == 1:
(16)                            if x.size == 2 and y.size == 2:
(20)                                style = "image"
(16)                            else:
(20)                                dx = np.diff(x)
(20)                                dy = np.diff(y)
(20)                                if (np.ptp(dx) < 0.01 * abs(dx.mean()) and
(28)                                        np.ptp(dy) < 0.01 * abs(dy.mean())):
(24)                                    style = "image"
(20)                                else:
(24)                                    style = "pcolorimage"
(12)                        elif x.ndim == 2 and y.ndim == 2:
(16)                            style = "quadmesh"
(12)                        else:
(16)                            raise TypeError(
(20)                                f"When 3 positional parameters are passed to pcolorfast, the first "
(20)                                f"two (X and Y) must be both 1D or both 2D; the given X was "
(20)                                f"{x.ndim}D and the given Y was {y.ndim}D")
(8)                     else:
(12)                        raise _api.nargs_error('pcolorfast', '1 or 3', len(args))
(8)                     if style == "quadmesh":
(12)                        coords = np.stack([x, y], axis=-1)
(12)                        if np.ndim(C) not in {2, 3}:
(16)                            raise ValueError("C must be 2D or 3D")
(12)                        collection = mcoll.QuadMesh(
(16)                            coords, array=C,
(16)                            alpha=alpha, cmap=cmap, norm=norm,
(16)                            antialiased=False, edgecolors="none")
(12)                        self.add_collection(collection, autolim=False)
(12)                        xl, xr, yb, yt = x.min(), x.max(), y.min(), y.max()
(12)                        ret = collection
(8)                     else:  # It's one of the two image styles.
(12)                        extent = xl, xr, yb, yt = x[0], x[-1], y[0], y[-1]
(12)                        if style == "image":
(16)                            im = mimage.AxesImage(
(20)                                self, cmap=cmap, norm=norm,
(20)                                data=C, alpha=alpha, extent=extent,
(20)                                interpolation='nearest', origin='lower',
(20)                                **kwargs)
(12)                        elif style == "pcolorimage":
(16)                            im = mimage.PcolorImage(
(20)                                self, x, y, C,
(20)                                cmap=cmap, norm=norm, alpha=alpha, extent=extent,
(20)                                **kwargs)
(12)                        self.add_image(im)
(12)                        ret = im
(8)                     if np.ndim(C) == 2:  # C.ndim == 3 is RGB(A) so doesn't need scaling.
(12)                        ret._scale_norm(norm, vmin, vmax)
(8)                     if ret.get_clip_path() is None:
(12)                        ret.set_clip_path(self.patch)
(8)                     ret.sticky_edges.x[:] = [xl, xr]
(8)                     ret.sticky_edges.y[:] = [yb, yt]
(8)                     self.update_datalim(np.array([[xl, yb], [xr, yt]]))
(8)                     self._request_autoscale_view(tight=True)
(8)                     return ret
(4)                 @_preprocess_data()
(4)                 @_docstring.dedent_interpd
(4)                 def contour(self, *args, **kwargs):
(8)                     """
(8)                     Plot contour lines.
(8)                     Call signature::
(12)                        contour([X, Y,] Z, [levels], **kwargs)
(8)                     %(contour_doc)s
(8)                     """
(8)                     kwargs['filled'] = False
(8)                     contours = mcontour.QuadContourSet(self, *args, **kwargs)
(8)                     self._request_autoscale_view()
```

```
(8)                 return contours
(4)             @_preprocess_data()
(4)             @_docstring.dedent_interpd
(4)             def contourf(self, *args, **kwargs):
(8)                 """
(8)                 Plot filled contours.
(8)                 Call signature::
(12)                    contourf([X, Y,] Z, [levels], **kwargs)
(8)                 %(contour_doc)s
(8)                 """
(8)                 kwargs['filled'] = True
(8)                 contours = mcontour.QuadContourSet(self, *args, **kwargs)
(8)                 self._request_autoscale_view()
(8)                 return contours
(4)             def clabel(self, CS, levels=None, **kwargs):
(8)                 """
(8)                 Label a contour plot.
(8)                 Adds labels to line contours in given `.ContourSet`.
(8)                 Parameters
(8)                 ----------
(8)                 CS : `.ContourSet` instance
(12)                    Line contours to label.
(8)                 levels : array-like, optional
(12)                    A list of level values, that should be labeled. The list must be
(12)                    a subset of ``CS.levels``. If not given, all levels are labeled.
(8)                 **kwargs
(12)                    All other parameters are documented in `~.ContourLabeler.clabel`.
(8)                 """
(8)                 return CS.clabel(levels, **kwargs)
(4)             @_preprocess_data(replace_names=["x", 'weights'], label_namer="x")
(4)             def hist(self, x, bins=None, range=None, density=False, weights=None,
(13)                     cumulative=False, bottom=None, histtype='bar', align='mid',
(13)                     orientation='vertical', rwidth=None, log=False,
(13)                     color=None, label=None, stacked=False, **kwargs):
(8)                 """
(8)                 Compute and plot a histogram.
(8)                 This method uses `numpy.histogram` to bin the data in *x* and count
the
(8)                 number of values in each bin, then draws the distribution either as a
(8)                 `.BarContainer` or `.Polygon`. The *bins*, *range*, *density*, and
(8)                 *weights* parameters are forwarded to `numpy.histogram`.
(8)                 If the data has already been binned and counted, use `~.bar` or
(8)                 `~.stairs` to plot the distribution::
(12)                    counts, bins = np.histogram(x)
(12)                    plt.stairs(counts, bins)
(8)                 Alternatively, plot pre-computed bins and counts using ``hist()`` by
(8)                 treating each bin as a single point with a weight equal to its count::
(12)                    plt.hist(bins[:-1], bins, weights=counts)
(8)                 The data input *x* can be a singular array, a list of datasets of
(8)                 potentially different lengths ([*x0*, *x1*, ...]), or a 2D ndarray in
(8)                 which each column is a dataset. Note that the ndarray form is
(8)                 transposed relative to the list form. If the input is an array, then
(8)                 the return value is a tuple (*n*, *bins*, *patches*); if the input is
a
(8)                 sequence of arrays, then the return value is a tuple
(8)                 ([*n0*, *n1*, ...], *bins*, [*patches0*, *patches1*, ...]).
(8)                 Masked arrays are not supported.
(8)                 Parameters
(8)                 ----------
(8)                 x : (n,) array or sequence of (n,) arrays
(12)                    Input values, this takes either a single array or a sequence of
(12)                    arrays which are not required to be of the same length.
(8)                 bins : int or sequence or str, default: :rc:`hist.bins`
(12)                    If *bins* is an integer, it defines the number of equal-width bins
(12)                    in the range.
(12)                    If *bins* is a sequence, it defines the bin edges, including the
(12)                    left edge of the first bin and the right edge of the last bin;
(12)                    in this case, bins may be unequally spaced.  All but the last
(12)                    (righthand-most) bin is half-open.  In other words, if *bins* is::
```

```
(16)                        [1, 2, 3, 4]
(12)                    then the first bin is ``[1, 2)`` (including 1, but excluding 2)
and
(12)                    the second ``[2, 3)``.  The last bin, however, is ``[3, 4]``,
which
(12)                    *includes* 4.
(12)                    If *bins* is a string, it is one of the binning strategies
(12)                    supported by `numpy.histogram_bin_edges`: 'auto', 'fd', 'doane',
(12)                    'scott', 'stone', 'rice', 'sturges', or 'sqrt'.
(8)                 range : tuple or None, default: None
(12)                    The lower and upper range of the bins. Lower and upper outliers
(12)                    are ignored. If not provided, *range* is ``(x.min(), x.max())``.
(12)                    Range has no effect if *bins* is a sequence.
(12)                    If *bins* is a sequence or *range* is specified, autoscaling
(12)                    is based on the specified bin range instead of the
(12)                    range of x.
(8)                 density : bool, default: False
(12)                    If ``True``, draw and return a probability density: each bin
(12)                    will display the bin's raw count divided by the total number of
(12)                    counts *and the bin width*
(12)                    (``density = counts / (sum(counts) * np.diff(bins))``),
(12)                    so that the area under the histogram integrates to 1
(12)                    (``np.sum(density * np.diff(bins)) == 1``).
(12)                    If *stacked* is also ``True``, the sum of the histograms is
(12)                    normalized to 1.
(8)                 weights : (n,) array-like or None, default: None
(12)                    An array of weights, of the same shape as *x*.  Each value in
(12)                    *x* only contributes its associated weight towards the bin count
(12)                    (instead of 1).  If *density* is ``True``, the weights are
(12)                    normalized, so that the integral of the density over the range
(12)                    remains 1.
(8)                 cumulative : bool or -1, default: False
(12)                    If ``True``, then a histogram is computed where each bin gives the
(12)                    counts in that bin plus all bins for smaller values. The last bin
(12)                    gives the total number of datapoints.
(12)                    If *density* is also ``True`` then the histogram is normalized
such
(12)                    that the last bin equals 1.
(12)                    If *cumulative* is a number less than 0 (e.g., -1), the direction
(12)                    of accumulation is reversed.  In this case, if *density* is also
(12)                    ``True``, then the histogram is normalized such that the first bin
(12)                    equals 1.
(8)                 bottom : array-like, scalar, or None, default: None
(12)                    Location of the bottom of each bin, i.e. bins are drawn from
(12)                    ``bottom`` to ``bottom + hist(x, bins)`` If a scalar, the bottom
(12)                    of each bin is shifted by the same amount. If an array, each bin
(12)                    is shifted independently and the length of bottom must match the
(12)                    number of bins. If None, defaults to 0.
(8)                 histtype : {'bar', 'barstacked', 'step', 'stepfilled'}, default: 'bar'
(12)                    The type of histogram to draw.
(12)                    - 'bar' is a traditional bar-type histogram.  If multiple data
(14)                      are given the bars are arranged side by side.
(12)                    - 'barstacked' is a bar-type histogram where multiple
(14)                      data are stacked on top of each other.
(12)                    - 'step' generates a lineplot that is by default unfilled.
(12)                    - 'stepfilled' generates a lineplot that is by default filled.
(8)                 align : {'left', 'mid', 'right'}, default: 'mid'
(12)                    The horizontal alignment of the histogram bars.
(12)                    - 'left': bars are centered on the left bin edges.
(12)                    - 'mid': bars are centered between the bin edges.
(12)                    - 'right': bars are centered on the right bin edges.
(8)                 orientation : {'vertical', 'horizontal'}, default: 'vertical'
(12)                    If 'horizontal', `~.Axes.barh` will be used for bar-type
histograms
(12)                    and the *bottom* kwarg will be the left edges.
(8)                 rwidth : float or None, default: None
(12)                    The relative width of the bars as a fraction of the bin width.  If
(12)                    ``None``, automatically compute the width.
(12)                    Ignored if *histtype* is 'step' or 'stepfilled'.
```

```
(8)                     log : bool, default: False
(12)                        If ``True``, the histogram axis will be set to a log scale.
(8)                     color : :mpltype:`color` or list of :mpltype:`color` or None, default:
None
(12)                        Color or sequence of colors, one per dataset.  Default (``None``)
(12)                        uses the standard line color sequence.
(8)                     label : str or list of str, optional
(12)                        String, or sequence of strings to match multiple datasets.  Bar
(12)                        charts yield multiple patches per dataset, but only the first gets
(12)                        the label, so that `~.Axes.legend` will work as expected.
(8)                     stacked : bool, default: False
(12)                        If ``True``, multiple data are stacked on top of each other If
(12)                        ``False`` multiple data are arranged side by side if histtype is
(12)                        'bar' or on top of each other if histtype is 'step'
(8)                     Returns
(8)                     -------
(8)                     n : array or list of arrays
(12)                        The values of the histogram bins. See *density* and *weights* for
a
(12)                        description of the possible semantics.  If input *x* is an array,
(12)                        then this is an array of length *nbins*. If input is a sequence of
(12)                        arrays ``[data1, data2, ...]``, then this is a list of arrays with
(12)                        the values of the histograms for each of the arrays in the same
(12)                        order.  The dtype of the array *n* (or of its element arrays) will
(12)                        always be float even if no weighting or normalization is used.
(8)                     bins : array
(12)                        The edges of the bins. Length nbins + 1 (nbins left edges and
right
(12)                        edge of last bin).  Always a single array even when multiple data
(12)                        sets are passed in.
(8)                     patches : `.BarContainer` or list of a single `.Polygon` or list of \
(0)             such objects
(12)                        Container of individual artists used to create the histogram
(12)                        or list of such containers if there are multiple input datasets.
(8)                     Other Parameters
(8)                     ----------------
(8)                     data : indexable object, optional
(12)                        DATA_PARAMETER_PLACEHOLDER
(8)                     **kwargs
(12)                        `~matplotlib.patches.Patch` properties
(8)                     See Also
(8)                     --------
(8)                     hist2d : 2D histogram with rectangular bins
(8)                     hexbin : 2D histogram with hexagonal bins
(8)                     stairs : Plot a pre-computed histogram
(8)                     bar : Plot a pre-computed histogram
(8)                     Notes
(8)                     -----
(8)                     For large numbers of bins (>1000), plotting can be significantly
(8)                     accelerated by using `~.Axes.stairs` to plot a pre-computed histogram
(8)                     (``plt.stairs(*np.histogram(data))``), or by setting *histtype* to
(8)                     'step' or 'stepfilled' rather than 'bar' or 'barstacked'.
(8)                     """
(8)                     bin_range = range
(8)                     from builtins import range
(8)                     if np.isscalar(x):
(12)                        x = [x]
(8)                     if bins is None:
(12)                        bins = mpl.rcParams['hist.bins']
(8)                     _api.check_in_list(['bar', 'barstacked', 'step', 'stepfilled'],
(27)                                        histtype=histtype)
(8)                     _api.check_in_list(['left', 'mid', 'right'], align=align)
(8)                     _api.check_in_list(['horizontal', 'vertical'],
orientation=orientation)
(8)                     if histtype == 'barstacked' and not stacked:
(12)                        stacked = True
(8)                     x = cbook._reshape_2D(x, 'x')
(8)                     nx = len(x)  # number of datasets
(8)                     if orientation == "vertical":
```

```
(12)                        convert_units = self.convert_xunits
(12)                        x = [*self._process_unit_info([("x", x[0])], kwargs),
(17)                             *map(convert_units, x[1:])]
(8)                     else:  # horizontal
(12)                        convert_units = self.convert_yunits
(12)                        x = [*self._process_unit_info([("y", x[0])], kwargs),
(17)                             *map(convert_units, x[1:])]
(8)                     if bin_range is not None:
(12)                        bin_range = convert_units(bin_range)
(8)                     if not cbook.is_scalar_or_string(bins):
(12)                        bins = convert_units(bins)
(8)                     if weights is not None:
(12)                        w = cbook._reshape_2D(weights, 'weights')
(8)                     else:
(12)                        w = [None] * nx
(8)                     if len(w) != nx:
(12)                        raise ValueError('weights should have the same shape as x')
(8)                     input_empty = True
(8)                     for xi, wi in zip(x, w):
(12)                        len_xi = len(xi)
(12)                        if wi is not None and len(wi) != len_xi:
(16)                            raise ValueError('weights should have the same shape as x')
(12)                        if len_xi:
(16)                            input_empty = False
(8)                     if color is None:
(12)                        colors = [self._get_lines.get_next_color() for i in range(nx)]
(8)                     else:
(12)                        colors = mcolors.to_rgba_array(color)
(12)                        if len(colors) != nx:
(16)                            raise ValueError(f"The 'color' keyword argument must have one
"
(33)                                                f"color per dataset, but {nx} datasets and "
(33)                                                f"{len(colors)} colors were provided")
(8)                     hist_kwargs = dict()
(8)                     if bin_range is None:
(12)                        xmin = np.inf
(12)                        xmax = -np.inf
(12)                        for xi in x:
(16)                            if len(xi):
(20)                                xmin = min(xmin, np.nanmin(xi))
(20)                                xmax = max(xmax, np.nanmax(xi))
(12)                        if xmin <= xmax:  # Only happens if we have seen a finite value.
(16)                            bin_range = (xmin, xmax)
(8)                     if not input_empty and len(x) > 1:
(12)                        if weights is not None:
(16)                            _w = np.concatenate(w)
(12)                        else:
(16)                            _w = None
(12)                        bins = np.histogram_bin_edges(
(16)                            np.concatenate(x), bins, bin_range, _w)
(8)                     else:
(12)                        hist_kwargs['range'] = bin_range
(8)                     density = bool(density)
(8)                     if density and not stacked:
(12)                        hist_kwargs['density'] = density
(8)                     tops = []  # Will have shape (n_datasets, n_bins).
(8)                     for i in range(nx):
(12)                        m, bins = np.histogram(x[i], bins, weights=w[i], **hist_kwargs)
(12)                        tops.append(m)
(8)                     tops = np.array(tops, float)  # causes problems later if it's an int
(8)                     bins = np.array(bins, float)  # causes problems if float16
(8)                     if stacked:
(12)                        tops = tops.cumsum(axis=0)
(12)                        if density:
(16)                            tops = (tops / np.diff(bins)) / tops[-1].sum()
(8)                     if cumulative:
(12)                        slc = slice(None)
(12)                        if isinstance(cumulative, Number) and cumulative < 0:
(16)                            slc = slice(None, None, -1)
```

```
(12)                        if density:
(16)                            tops = (tops * np.diff(bins))[:, slc].cumsum(axis=1)[:, slc]
(12)                        else:
(16)                            tops = tops[:, slc].cumsum(axis=1)[:, slc]
(8)                     patches = []
(8)                     if histtype.startswith('bar'):
(12)                        totwidth = np.diff(bins)
(12)                        if rwidth is not None:
(16)                            dr = np.clip(rwidth, 0, 1)
(12)                        elif (len(tops) > 1 and
(18)                              ((not stacked) or mpl.rcParams['_internal.classic_mode'])):
(16)                            dr = 0.8
(12)                        else:
(16)                            dr = 1.0
(12)                        if histtype == 'bar' and not stacked:
(16)                            width = dr * totwidth / nx
(16)                            dw = width
(16)                            boffset = -0.5 * dr * totwidth * (1 - 1 / nx)
(12)                        elif histtype == 'barstacked' or stacked:
(16)                            width = dr * totwidth
(16)                            boffset, dw = 0.0, 0.0
(12)                        if align == 'mid':
(16)                            boffset += 0.5 * totwidth
(12)                        elif align == 'right':
(16)                            boffset += totwidth
(12)                        if orientation == 'horizontal':
(16)                            _barfunc = self.barh
(16)                            bottom_kwarg = 'left'
(12)                        else:  # orientation == 'vertical'
(16)                            _barfunc = self.bar
(16)                            bottom_kwarg = 'bottom'
(12)                        for top, color in zip(tops, colors):
(16)                            if bottom is None:
(20)                                bottom = np.zeros(len(top))
(16)                            if stacked:
(20)                                height = top - bottom
(16)                            else:
(20)                                height = top
(16)                            bars = _barfunc(bins[:-1]+boffset, height, width,
(32)                                            align='center', log=log,
(32)                                            color=color, **{bottom_kwarg: bottom})
(16)                            patches.append(bars)
(16)                            if stacked:
(20)                                bottom = top
(16)                            boffset += dw
(12)                        for bars in patches[1:]:
(16)                            for patch in bars:
(20)                                patch.sticky_edges.x[:] = patch.sticky_edges.y[:] = []
(8)                     elif histtype.startswith('step'):
(12)                        x = np.zeros(4 * len(bins) - 3)
(12)                        y = np.zeros(4 * len(bins) - 3)
(12)                        x[0:2*len(bins)-1:2], x[1:2*len(bins)-1:2] = bins, bins[:-1]
(12)                        x[2*len(bins)-1:] = x[1:2*len(bins)-1][::-1]
(12)                        if bottom is None:
(16)                            bottom = 0
(12)                        y[1:2*len(bins)-1:2] = y[2:2*len(bins):2] = bottom
(12)                        y[2*len(bins)-1:] = y[1:2*len(bins)-1][::-1]
(12)                        if log:
(16)                            if orientation == 'horizontal':
(20)                                self.set_xscale('log', nonpositive='clip')
(16)                            else:  # orientation == 'vertical'
(20)                                self.set_yscale('log', nonpositive='clip')
(12)                        if align == 'left':
(16)                            x -= 0.5*(bins[1]-bins[0])
(12)                        elif align == 'right':
(16)                            x += 0.5*(bins[1]-bins[0])
(12)                        fill = (histtype == 'stepfilled')
(12)                        xvals, yvals = [], []
(12)                        for top in tops:
```

```
(16)                        if stacked:
(20)                            y[2*len(bins)-1:] = y[1:2*len(bins)-1][::-1]
(16)                        y[1:2*len(bins)-1:2] = y[2:2*len(bins):2] = top + bottom
(16)                        y[0] = y[-1]
(16)                        if orientation == 'horizontal':
(20)                            xvals.append(y.copy())
(20)                            yvals.append(x.copy())
(16)                        else:
(20)                            xvals.append(x.copy())
(20)                            yvals.append(y.copy())
(12)                    split = -1 if fill else 2 * len(bins)
(12)                    for x, y, color in reversed(list(zip(xvals, yvals, colors))):
(16)                        patches.append(self.fill(
(20)                            x[:split], y[:split],
(20)                            closed=True if fill else None,
(20)                            facecolor=color,
(20)                            edgecolor=None if fill else color,
(20)                            fill=fill if fill else None,
(20)                            zorder=None if fill else mlines.Line2D.zorder))
(12)                    for patch_list in patches:
(16)                        for patch in patch_list:
(20)                            if orientation == 'vertical':
(24)                                patch.sticky_edges.y.append(0)
(20)                            elif orientation == 'horizontal':
(24)                                patch.sticky_edges.x.append(0)
(12)                    patches.reverse()
(8)                 labels = [] if label is None else np.atleast_1d(np.asarray(label,
str))
(8)                 for patch, lbl in itertools.zip_longest(patches, labels):
(12)                    if patch:
(16)                        p = patch[0]
(16)                        p._internal_update(kwargs)
(16)                        if lbl is not None:
(20)                            p.set_label(lbl)
(16)                        for p in patch[1:]:
(20)                            p._internal_update(kwargs)
(20)                            p.set_label('_nolegend_')
(8)                 if nx == 1:
(12)                    return tops[0], bins, patches[0]
(8)                 else:
(12)                    patch_type = ("BarContainer" if histtype.startswith("bar")
(26)                                  else "list[Polygon]")
(12)                    return tops, bins, cbook.silent_list(patch_type, patches)
(4)             @_preprocess_data()
(4)             def stairs(self, values, edges=None, *,
(15)                       orientation='vertical', baseline=0, fill=False, **kwargs):
(8)                 """
(8)                 Draw a stepwise constant function as a line or a filled plot.
(8)                 *edges* define the x-axis positions of the steps. *values* the
function values
(8)                 between these steps. Depending on *fill*, the function is drawn either
as a
(8)                 continuous line with vertical segments at the edges, or as a filled
area.
(8)                 Parameters
(8)                 ----------
(8)                 values : array-like
(12)                    The step heights.
(8)                 edges : array-like
(12)                    The step positions, with ``len(edges) == len(vals) + 1``,
(12)                    between which the curve takes on vals values.
(8)                 orientation : {'vertical', 'horizontal'}, default: 'vertical'
(12)                    The direction of the steps. Vertical means that *values* are along
(12)                    the y-axis, and edges are along the x-axis.
(8)                 baseline : float, array-like or None, default: 0
(12)                    The bottom value of the bounding edges or when
(12)                    ``fill=True``, position of lower edge. If *fill* is
(12)                    True or an array is passed to *baseline*, a closed
(12)                    path is drawn.
```

```
(8)                 fill : bool, default: False
(12)                    Whether the area under the step curve should be filled.
(8)                 Returns
(8)                 -------
(8)                 StepPatch : `~matplotlib.patches.StepPatch`
(8)                 Other Parameters
(8)                 ----------------
(8)                 data : indexable object, optional
(12)                    DATA_PARAMETER_PLACEHOLDER
(8)                 **kwargs
(12)                    `~matplotlib.patches.StepPatch` properties
(8)                 """
(8)                 if 'color' in kwargs:
(12)                    _color = kwargs.pop('color')
(8)                 else:
(12)                    _color = self._get_lines.get_next_color()
(8)                 if fill:
(12)                    kwargs.setdefault('linewidth', 0)
(12)                    kwargs.setdefault('facecolor', _color)
(8)                 else:
(12)                    kwargs.setdefault('edgecolor', _color)
(8)                 if edges is None:
(12)                    edges = np.arange(len(values) + 1)
(8)                 edges, values, baseline = self._process_unit_info(
(12)                    [("x", edges), ("y", values), ("y", baseline)], kwargs)
(8)                 patch = mpatches.StepPatch(values,
(35)                                           edges,
(35)                                           baseline=baseline,
(35)                                           orientation=orientation,
(35)                                           fill=fill,
(35)                                           **kwargs)
(8)                 self.add_patch(patch)
(8)                 if baseline is None:
(12)                    baseline = 0
(8)                 if orientation == 'vertical':
(12)                    patch.sticky_edges.y.append(np.min(baseline))
(12)                    self.update_datalim([(edges[0], np.min(baseline))])
(8)                 else:
(12)                    patch.sticky_edges.x.append(np.min(baseline))
(12)                    self.update_datalim([(np.min(baseline), edges[0])])
(8)                 self._request_autoscale_view()
(8)                 return patch
(4)             @_preprocess_data(replace_names=["x", "y", "weights"])
(4)             @_docstring.dedent_interpd
(4)             def hist2d(self, x, y, bins=10, range=None, density=False, weights=None,
(15)                       cmin=None, cmax=None, **kwargs):
(8)                 """
(8)                 Make a 2D histogram plot.
(8)                 Parameters
(8)                 ----------
(8)                 x, y : array-like, shape (n, )
(12)                    Input values
(8)                 bins : None or int or [int, int] or array-like or [array, array]
(12)                    The bin specification:
(12)                    - If int, the number of bins for the two dimensions
(14)                      (``nx = ny = bins``).
(12)                    - If ``[int, int]``, the number of bins in each dimension
(14)                      (``nx, ny = bins``).
(12)                    - If array-like, the bin edges for the two dimensions
(14)                      (``x_edges = y_edges = bins``).
(12)                    - If ``[array, array]``, the bin edges in each dimension
(14)                      (``x_edges, y_edges = bins``).
(12)                    The default value is 10.
(8)                 range : array-like shape(2, 2), optional
(12)                    The leftmost and rightmost edges of the bins along each dimension
(12)                    (if not specified explicitly in the bins parameters): ``[[xmin,
(12)                    xmax], [ymin, ymax]]``. All values outside of this range will be
(12)                    considered outliers and not tallied in the histogram.
(8)                 density : bool, default: False
```

```
5597: (12)                      Normalize histogram.  See the documentation for the *density*
5598: (12)                      parameter of `~.Axes.hist` for more details.
5599: (8)                   weights : array-like, shape (n, ), optional
5600: (12)                      An array of values w_i weighing each sample (x_i, y_i).
5601: (8)                   cmin, cmax : float, default: None
5602: (12)                      All bins that has count less than *cmin* or more than *cmax* will
not be
5603: (12)                      displayed (set to NaN before passing to `~.Axes.pcolormesh`) and
these count
5604: (12)                      values in the return value count histogram will also be set to nan
upon
5605: (12)                      return.
5606: (8)                   Returns
5607: (8)                   -------
5608: (8)                   h : 2D array
5609: (12)                      The bi-dimensional histogram of samples x and y. Values in x are
5610: (12)                      histogrammed along the first dimension and values in y are
5611: (12)                      histogrammed along the second dimension.
5612: (8)                   xedges : 1D array
5613: (12)                      The bin edges along the x-axis.
5614: (8)                   yedges : 1D array
5615: (12)                      The bin edges along the y-axis.
5616: (8)                   image : `~.matplotlib.collections.QuadMesh`
5617: (8)                   Other Parameters
5618: (8)                   ----------------
5619: (8)                   %(cmap_doc)s
5620: (8)                   %(norm_doc)s
5621: (8)                   %(vmin_vmax_doc)s
5622: (8)                   alpha : ``0 <= scalar <= 1`` or ``None``, optional
5623: (12)                      The alpha blending value.
5624: (8)                   data : indexable object, optional
5625: (12)                      DATA_PARAMETER_PLACEHOLDER
5626: (8)                   **kwargs
5627: (12)                      Additional parameters are passed along to the
5628: (12)                      `~.Axes.pcolormesh` method and `~matplotlib.collections.QuadMesh`
5629: (12)                      constructor.
5630: (8)                   See Also
5631: (8)                   --------
5632: (8)                   hist : 1D histogram plotting
5633: (8)                   hexbin : 2D histogram with hexagonal bins
5634: (8)                   Notes
5635: (8)                   -----
5636: (8)                   - Currently ``hist2d`` calculates its own axis limits, and any limits
5637: (10)                    previously set are ignored.
5638: (8)                   - Rendering the histogram with a logarithmic color scale is
5639: (10)                    accomplished by passing a `.colors.LogNorm` instance to the *norm*
5640: (10)                    keyword argument. Likewise, power-law normalization (similar
5641: (10)                    in effect to gamma correction) can be accomplished with
5642: (10)                    `.colors.PowerNorm`.
5643: (8)                   """
5644: (8)                   h, xedges, yedges = np.histogram2d(x, y, bins=bins, range=range,
5645: (43)                                          density=density, weights=weights)
5646: (8)                   if cmin is not None:
5647: (12)                      h[h < cmin] = None
5648: (8)                   if cmax is not None:
5649: (12)                      h[h > cmax] = None
5650: (8)                   pc = self.pcolormesh(xedges, yedges, h.T, **kwargs)
5651: (8)                   self.set_xlim(xedges[0], xedges[-1])
5652: (8)                   self.set_ylim(yedges[0], yedges[-1])
5653: (8)                   return h, xedges, yedges, pc
5654: (4)               @_preprocess_data(replace_names=["x", "weights"], label_namer="x")
5655: (4)               @_docstring.dedent_interpd
5656: (4)               def ecdf(self, x, weights=None, *, complementary=False,
5657: (13)                       orientation="vertical", compress=False, **kwargs):
5658: (8)                   """
5659: (8)                   Compute and plot the empirical cumulative distribution function of
*x*.
5660: (8)                   .. versionadded:: 3.8
5661: (8)                   Parameters
```

```
    ----------
    x : 1d array-like
        The input data.  Infinite entries are kept (and move the relevant
        end of the ecdf from 0/1), but NaNs and masked values are errors.
    weights : 1d array-like or None, default: None
        The weights of the entries; must have the same shape as *x*.
        Weights corresponding to NaN data points are dropped, and then the
        remaining weights are normalized to sum to 1.  If unset, all
        entries have the same weight.
    complementary : bool, default: False
        Whether to plot a cumulative distribution function, which increases

        from 0 to 1 (the default), or a complementary cumulative
        distribution function, which decreases from 1 to 0.
    orientation : {"vertical", "horizontal"}, default: "vertical"
        Whether the entries are plotted along the x-axis ("vertical", the
        default) or the y-axis ("horizontal").  This parameter takes the
        same values as in `~.Axes.hist`.
    compress : bool, default: False
        Whether multiple entries with the same values are grouped together
        (with a summed weight) before plotting.  This is mainly useful if
        *x* contains many identical data points, to decrease the rendering
        complexity of the plot. If *x* contains no duplicate points, this
        has no effect and just uses some time and memory.
    Other Parameters
    ----------------
    data : indexable object, optional
        DATA_PARAMETER_PLACEHOLDER
    **kwargs
        Keyword arguments control the `.Line2D` properties:
        %(Line2D:kwdoc)s
    Returns
    -------
    `.Line2D`
    Notes
    -----
    The ecdf plot can be thought of as a cumulative histogram with one bin
    per data entry; i.e. it reports on the entire dataset without any
    arbitrary binning.
    If *x* contains NaNs or masked entries, either remove them first from
    the array (if they should not taken into account), or replace them by
    -inf or +inf (if they should be sorted at the beginning or the end of
    the array).
    """
    _api.check_in_list(["horizontal", "vertical"], orientation=orientation)
    if "drawstyle" in kwargs or "ds" in kwargs:
        raise TypeError("Cannot pass 'drawstyle' or 'ds' to ecdf()")
    if np.ma.getmask(x).any():
        raise ValueError("ecdf() does not support masked entries")
    x = np.asarray(x)
    if np.isnan(x).any():
        raise ValueError("ecdf() does not support NaNs")
    argsort = np.argsort(x)
    x = x[argsort]
    if weights is None:
        cum_weights = (1 + np.arange(len(x))) / len(x)
    else:
        weights = np.take(weights, argsort)   # Reorder weights like we reordered x.
        cum_weights = np.cumsum(weights / np.sum(weights))
    if compress:
        compress_idxs = [0, *(x[:-1] != x[1:]).nonzero()[0] + 1]
        x = x[compress_idxs]
        cum_weights = cum_weights[compress_idxs]
    if orientation == "vertical":
        if not complementary:
            line, = self.plot([x[0], *x], [0, *cum_weights],
                              drawstyle="steps-post", **kwargs)
```

```
(12)                        else:
(16)                            line, = self.plot([*x, x[-1]], [1, *1 - cum_weights],
(34)                                              drawstyle="steps-pre", **kwargs)
(12)                        line.sticky_edges.y[:] = [0, 1]
(8)                     else:  # orientation == "horizontal":
(12)                        if not complementary:
(16)                            line, = self.plot([0, *cum_weights], [x[0], *x],
(34)                                              drawstyle="steps-pre", **kwargs)
(12)                        else:
(16)                            line, = self.plot([1, *1 - cum_weights], [*x, x[-1]],
(34)                                              drawstyle="steps-post", **kwargs)
(12)                        line.sticky_edges.x[:] = [0, 1]
(8)                     return line
(4)                 @_preprocess_data(replace_names=["x"])
(4)                 @_docstring.dedent_interpd
(4)                 def psd(self, x, NFFT=None, Fs=None, Fc=None, detrend=None,
(12)                        window=None, noverlap=None, pad_to=None,
(12)                        sides=None, scale_by_freq=None, return_line=None, **kwargs):
(8)                     r"""
(8)                     Plot the power spectral density.
(8)                     The power spectral density :math:`P_{xx}` by Welch's average
(8)                     periodogram method.  The vector *x* is divided into *NFFT* length
(8)                     segments.  Each segment is detrended by function *detrend* and
(8)                     windowed by function *window*.  *noverlap* gives the length of
(8)                     the overlap between segments.  The :math:`|\mathrm{fft}(i)|^2`
(8)                     of each segment :math:`i` are averaged to compute :math:`P_{xx}`,
(8)                     with a scaling to correct for power loss due to windowing.
(8)                     If len(*x*) < *NFFT*, it will be zero padded to *NFFT*.
(8)                     Parameters
(8)                     ----------
(8)                     x : 1-D array or sequence
(12)                        Array or sequence containing the data
(8)                     %(Spectral)s
(8)                     %(PSD)s
(8)                     noverlap : int, default: 0 (no overlap)
(12)                        The number of points of overlap between segments.
(8)                     Fc : int, default: 0
(12)                        The center frequency of *x*, which offsets the x extents of the
(12)                        plot to reflect the frequency range used when a signal is acquired
(12)                        and then filtered and downsampled to baseband.
(8)                     return_line : bool, default: False
(12)                        Whether to include the line object plotted in the returned values.
(8)                     Returns
(8)                     -------
(8)                     Pxx : 1-D array
(12)                        The values for the power spectrum :math:`P_{xx}` before scaling
(12)                        (real valued).
(8)                     freqs : 1-D array
(12)                        The frequencies corresponding to the elements in *Pxx*.
(8)                     line : `~matplotlib.lines.Line2D`
(12)                        The line created by this function.
(12)                        Only returned if *return_line* is True.
(8)                     Other Parameters
(8)                     ----------------
(8)                     data : indexable object, optional
(12)                        DATA_PARAMETER_PLACEHOLDER
(8)                     **kwargs
(12)                        Keyword arguments control the `.Line2D` properties:
(12)                        %(Line2D:kwdoc)s
(8)                     See Also
(8)                     --------
(8)                     specgram
(12)                        Differs in the default overlap; in not returning the mean of the
(12)                        segment periodograms; in returning the times of the segments; and
(12)                        in plotting a colormap instead of a line.
(8)                     magnitude_spectrum
(12)                        Plots the magnitude spectrum.
(8)                     csd
(12)                        Plots the spectral density between two signals.
```

```
(8)                 Notes
(8)                 -----
(8)                 For plotting, the power is plotted as
(8)                 :math:`10\log_{10}(P_{xx})` for decibels, though *Pxx* itself
(8)                 is returned.
(8)                 References
(8)                 ----------
(8)                 Bendat & Piersol -- Random Data: Analysis and Measurement Procedures,
(8)                 John Wiley & Sons (1986)
(8)                 """
(8)                 if Fc is None:
(12)                    Fc = 0
(8)                 pxx, freqs = mlab.psd(x=x, NFFT=NFFT, Fs=Fs, detrend=detrend,
(30)                                      window=window, noverlap=noverlap, pad_to=pad_to,
(30)                                      sides=sides, scale_by_freq=scale_by_freq)
(8)                 freqs += Fc
(8)                 if scale_by_freq in (None, True):
(12)                    psd_units = 'dB/Hz'
(8)                 else:
(12)                    psd_units = 'dB'
(8)                 line = self.plot(freqs, 10 * np.log10(pxx), **kwargs)
(8)                 self.set_xlabel('Frequency')
(8)                 self.set_ylabel('Power Spectral Density (%s)' % psd_units)
(8)                 self.grid(True)
(8)                 vmin, vmax = self.get_ybound()
(8)                 step = max(10 * int(np.log10(vmax - vmin)), 1)
(8)                 ticks = np.arange(math.floor(vmin), math.ceil(vmax) + 1, step)
(8)                 self.set_yticks(ticks)
(8)                 if return_line is None or not return_line:
(12)                    return pxx, freqs
(8)                 else:
(12)                    return pxx, freqs, line
(4)             @_preprocess_data(replace_names=["x", "y"], label_namer="y")
(4)             @_docstring.dedent_interpd
(4)             def csd(self, x, y, NFFT=None, Fs=None, Fc=None, detrend=None,
(12)                    window=None, noverlap=None, pad_to=None,
(12)                    sides=None, scale_by_freq=None, return_line=None, **kwargs):
(8)                 r"""
(8)                 Plot the cross-spectral density.
(8)                 The cross spectral density :math:`P_{xy}` by Welch's average
(8)                 periodogram method.  The vectors *x* and *y* are divided into
(8)                 *NFFT* length segments.  Each segment is detrended by function
(8)                 *detrend* and windowed by function *window*.  *noverlap* gives
(8)                 the length of the overlap between segments.  The product of
(8)                 the direct FFTs of *x* and *y* are averaged over each segment
(8)                 to compute :math:`P_{xy}`, with a scaling to correct for power
(8)                 loss due to windowing.
(8)                 If len(*x*) < *NFFT* or len(*y*) < *NFFT*, they will be zero
(8)                 padded to *NFFT*.
(8)                 Parameters
(8)                 ----------
(8)                 x, y : 1-D arrays or sequences
(12)                    Arrays or sequences containing the data.
(8)                 %(Spectral)s
(8)                 %(PSD)s
(8)                 noverlap : int, default: 0 (no overlap)
(12)                    The number of points of overlap between segments.
(8)                 Fc : int, default: 0
(12)                    The center frequency of *x*, which offsets the x extents of the
(12)                    plot to reflect the frequency range used when a signal is acquired
(12)                    and then filtered and downsampled to baseband.
(8)                 return_line : bool, default: False
(12)                    Whether to include the line object plotted in the returned values.
(8)                 Returns
(8)                 -------
(8)                 Pxy : 1-D array
(12)                    The values for the cross spectrum :math:`P_{xy}` before scaling
(12)                    (complex valued).
(8)                 freqs : 1-D array
```

```
(12)                    The frequencies corresponding to the elements in *Pxy*.
(8)                 line : `~matplotlib.lines.Line2D`
(12)                    The line created by this function.
(12)                    Only returned if *return_line* is True.
(8)                 Other Parameters
(8)                 ----------------
(8)                 data : indexable object, optional
(12)                    DATA_PARAMETER_PLACEHOLDER
(8)                 **kwargs
(12)                    Keyword arguments control the `.Line2D` properties:
(12)                    %(Line2D:kwdoc)s
(8)                 See Also
(8)                 --------
(8)                 psd : is equivalent to setting ``y = x``.
(8)                 Notes
(8)                 -----
(8)                 For plotting, the power is plotted as
(8)                 :math:`10 \log_{10}(P_{xy})` for decibels, though :math:`P_{xy}`
itself
(8)                 is returned.
(8)                 References
(8)                 ----------
(8)                 Bendat & Piersol -- Random Data: Analysis and Measurement Procedures,
(8)                 John Wiley & Sons (1986)
(8)                 """
(8)                 if Fc is None:
(12)                    Fc = 0
(8)                 pxy, freqs = mlab.csd(x=x, y=y, NFFT=NFFT, Fs=Fs, detrend=detrend,
(30)                                      window=window, noverlap=noverlap, pad_to=pad_to,
(30)                                      sides=sides, scale_by_freq=scale_by_freq)
(8)                 freqs += Fc
(8)                 line = self.plot(freqs, 10 * np.log10(np.abs(pxy)), **kwargs)
(8)                 self.set_xlabel('Frequency')
(8)                 self.set_ylabel('Cross Spectrum Magnitude (dB)')
(8)                 self.grid(True)
(8)                 vmin, vmax = self.get_ybound()
(8)                 step = max(10 * int(np.log10(vmax - vmin)), 1)
(8)                 ticks = np.arange(math.floor(vmin), math.ceil(vmax) + 1, step)
(8)                 self.set_yticks(ticks)
(8)                 if return_line is None or not return_line:
(12)                    return pxy, freqs
(8)                 else:
(12)                    return pxy, freqs, line
(4)             @_preprocess_data(replace_names=["x"])
(4)             @_docstring.dedent_interpd
(4)             def magnitude_spectrum(self, x, Fs=None, Fc=None, window=None,
(27)                                   pad_to=None, sides=None, scale=None,
(27)                                   **kwargs):
(8)                 """
(8)                 Plot the magnitude spectrum.
(8)                 Compute the magnitude spectrum of *x*.  Data is padded to a
(8)                 length of *pad_to* and the windowing function *window* is applied to
(8)                 the signal.
(8)                 Parameters
(8)                 ----------
(8)                 x : 1-D array or sequence
(12)                    Array or sequence containing the data.
(8)                 %(Spectral)s
(8)                 %(Single_Spectrum)s
(8)                 scale : {'default', 'linear', 'dB'}
(12)                    The scaling of the values in the *spec*.  'linear' is no scaling.
(12)                    'dB' returns the values in dB scale, i.e., the dB amplitude
(12)                    (20 * log10). 'default' is 'linear'.
(8)                 Fc : int, default: 0
(12)                    The center frequency of *x*, which offsets the x extents of the
(12)                    plot to reflect the frequency range used when a signal is acquired
(12)                    and then filtered and downsampled to baseband.
(8)                 Returns
(8)                 -------
```

```
(8)                 spectrum : 1-D array
(12)                    The values for the magnitude spectrum before scaling (real
valued).
(8)                 freqs : 1-D array
(12)                    The frequencies corresponding to the elements in *spectrum*.
(8)                 line : `~matplotlib.lines.Line2D`
(12)                    The line created by this function.
(8)                 Other Parameters
(8)                 ----------------
(8)                 data : indexable object, optional
(12)                    DATA_PARAMETER_PLACEHOLDER
(8)                 **kwargs
(12)                    Keyword arguments control the `.Line2D` properties:
(12)                    %(Line2D:kwdoc)s
(8)                 See Also
(8)                 --------
(8)                 psd
(12)                    Plots the power spectral density.
(8)                 angle_spectrum
(12)                    Plots the angles of the corresponding frequencies.
(8)                 phase_spectrum
(12)                    Plots the phase (unwrapped angle) of the corresponding
frequencies.
(8)                 specgram
(12)                    Can plot the magnitude spectrum of segments within the signal in a
(12)                    colormap.
(8)                 """
(8)                 if Fc is None:
(12)                    Fc = 0
(8)                 spec, freqs = mlab.magnitude_spectrum(x=x, Fs=Fs, window=window,
(46)                                                      pad_to=pad_to, sides=sides)
(8)                 freqs += Fc
(8)                 yunits = _api.check_getitem(
(12)                    {None: 'energy', 'default': 'energy', 'linear': 'energy',
(13)                     'dB': 'dB'},
(12)                    scale=scale)
(8)                 if yunits == 'energy':
(12)                    Z = spec
(8)                 else:  # yunits == 'dB'
(12)                    Z = 20. * np.log10(spec)
(8)                 line, = self.plot(freqs, Z, **kwargs)
(8)                 self.set_xlabel('Frequency')
(8)                 self.set_ylabel('Magnitude (%s)' % yunits)
(8)                 return spec, freqs, line
(4)             @_preprocess_data(replace_names=["x"])
(4)             @_docstring.dedent_interpd
(4)             def angle_spectrum(self, x, Fs=None, Fc=None, window=None,
(23)                               pad_to=None, sides=None, **kwargs):
(8)                 """
(8)                 Plot the angle spectrum.
(8)                 Compute the angle spectrum (wrapped phase spectrum) of *x*.
(8)                 Data is padded to a length of *pad_to* and the windowing function
(8)                 *window* is applied to the signal.
(8)                 Parameters
(8)                 ----------
(8)                 x : 1-D array or sequence
(12)                    Array or sequence containing the data.
(8)                 %(Spectral)s
(8)                 %(Single_Spectrum)s
(8)                 Fc : int, default: 0
(12)                    The center frequency of *x*, which offsets the x extents of the
(12)                    plot to reflect the frequency range used when a signal is acquired
(12)                    and then filtered and downsampled to baseband.
(8)                 Returns
(8)                 -------
(8)                 spectrum : 1-D array
(12)                    The values for the angle spectrum in radians (real valued).
(8)                 freqs : 1-D array
(12)                    The frequencies corresponding to the elements in *spectrum*.
```

```
(8)                 line : `~matplotlib.lines.Line2D`
(12)                    The line created by this function.
(8)                 Other Parameters
(8)                 ----------------
(8)                 data : indexable object, optional
(12)                    DATA_PARAMETER_PLACEHOLDER
(8)                 **kwargs
(12)                    Keyword arguments control the `.Line2D` properties:
(12)                    %(Line2D:kwdoc)s
(8)                 See Also
(8)                 --------
(8)                 magnitude_spectrum
(12)                    Plots the magnitudes of the corresponding frequencies.
(8)                 phase_spectrum
(12)                    Plots the unwrapped version of this function.
(8)                 specgram
(12)                    Can plot the angle spectrum of segments within the signal in a
(12)                    colormap.
(8)                 """
(8)                 if Fc is None:
(12)                    Fc = 0
(8)                 spec, freqs = mlab.angle_spectrum(x=x, Fs=Fs, window=window,
(42)                                                  pad_to=pad_to, sides=sides)
(8)                 freqs += Fc
(8)                 lines = self.plot(freqs, spec, **kwargs)
(8)                 self.set_xlabel('Frequency')
(8)                 self.set_ylabel('Angle (radians)')
(8)                 return spec, freqs, lines[0]
(4)             @_preprocess_data(replace_names=["x"])
(4)             @_docstring.dedent_interpd
(4)             def phase_spectrum(self, x, Fs=None, Fc=None, window=None,
(23)                               pad_to=None, sides=None, **kwargs):
(8)                 """
(8)                 Plot the phase spectrum.
(8)                 Compute the phase spectrum (unwrapped angle spectrum) of *x*.
(8)                 Data is padded to a length of *pad_to* and the windowing function
(8)                 *window* is applied to the signal.
(8)                 Parameters
(8)                 ----------
(8)                 x : 1-D array or sequence
(12)                    Array or sequence containing the data
(8)                 %(Spectral)s
(8)                 %(Single_Spectrum)s
(8)                 Fc : int, default: 0
(12)                    The center frequency of *x*, which offsets the x extents of the
(12)                    plot to reflect the frequency range used when a signal is acquired
(12)                    and then filtered and downsampled to baseband.
(8)                 Returns
(8)                 -------
(8)                 spectrum : 1-D array
(12)                    The values for the phase spectrum in radians (real valued).
(8)                 freqs : 1-D array
(12)                    The frequencies corresponding to the elements in *spectrum*.
(8)                 line : `~matplotlib.lines.Line2D`
(12)                    The line created by this function.
(8)                 Other Parameters
(8)                 ----------------
(8)                 data : indexable object, optional
(12)                    DATA_PARAMETER_PLACEHOLDER
(8)                 **kwargs
(12)                    Keyword arguments control the `.Line2D` properties:
(12)                    %(Line2D:kwdoc)s
(8)                 See Also
(8)                 --------
(8)                 magnitude_spectrum
(12)                    Plots the magnitudes of the corresponding frequencies.
(8)                 angle_spectrum
(12)                    Plots the wrapped version of this function.
(8)                 specgram
```

```
(12)                    Can plot the phase spectrum of segments within the signal in a
(12)                    colormap.
(8)                 """
(8)                 if Fc is None:
(12)                    Fc = 0
(8)                 spec, freqs = mlab.phase_spectrum(x=x, Fs=Fs, window=window,
(42)                                                  pad_to=pad_to, sides=sides)
(8)                 freqs += Fc
(8)                 lines = self.plot(freqs, spec, **kwargs)
(8)                 self.set_xlabel('Frequency')
(8)                 self.set_ylabel('Phase (radians)')
(8)                 return spec, freqs, lines[0]
(4)             @_preprocess_data(replace_names=["x", "y"])
(4)             @_docstring.dedent_interpd
(4)             def cohere(self, x, y, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
(15)                       window=mlab.window_hanning, noverlap=0, pad_to=None,
(15)                       sides='default', scale_by_freq=None, **kwargs):
(8)                 r"""
(8)                 Plot the coherence between *x* and *y*.
(8)                 Coherence is the normalized cross spectral density:
(8)                 .. math::
(10)                  C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}
(8)                 Parameters
(8)                 ----------
(8)                 %(Spectral)s
(8)                 %(PSD)s
(8)                 noverlap : int, default: 0 (no overlap)
(12)                    The number of points of overlap between blocks.
(8)                 Fc : int, default: 0
(12)                    The center frequency of *x*, which offsets the x extents of the
(12)                    plot to reflect the frequency range used when a signal is acquired
(12)                    and then filtered and downsampled to baseband.
(8)                 Returns
(8)                 -------
(8)                 Cxy : 1-D array
(12)                    The coherence vector.
(8)                 freqs : 1-D array
(12)                    The frequencies for the elements in *Cxy*.
(8)                 Other Parameters
(8)                 ----------------
(8)                 data : indexable object, optional
(12)                    DATA_PARAMETER_PLACEHOLDER
(8)                 **kwargs
(12)                    Keyword arguments control the `.Line2D` properties:
(12)                    %(Line2D:kwdoc)s
(8)                 References
(8)                 ----------
(8)                 Bendat & Piersol -- Random Data: Analysis and Measurement Procedures,
(8)                 John Wiley & Sons (1986)
(8)                 """
(8)                 cxy, freqs = mlab.cohere(x=x, y=y, NFFT=NFFT, Fs=Fs, detrend=detrend,
(33)                                         window=window, noverlap=noverlap,
(33)                                         scale_by_freq=scale_by_freq, sides=sides,
(33)                                         pad_to=pad_to)
(8)                 freqs += Fc
(8)                 self.plot(freqs, cxy, **kwargs)
(8)                 self.set_xlabel('Frequency')
(8)                 self.set_ylabel('Coherence')
(8)                 self.grid(True)
(8)                 return cxy, freqs
(4)             @_preprocess_data(replace_names=["x"])
(4)             @_docstring.dedent_interpd
(4)             def specgram(self, x, NFFT=None, Fs=None, Fc=None, detrend=None,
(17)                         window=None, noverlap=None,
(17)                         cmap=None, xextent=None, pad_to=None, sides=None,
(17)                         scale_by_freq=None, mode=None, scale=None,
(17)                         vmin=None, vmax=None, **kwargs):
(8)                 """
(8)                 Plot a spectrogram.
```

```
(8)                     Compute and plot a spectrogram of data in *x*.  Data are split into
(8)                     *NFFT* length segments and the spectrum of each section is
(8)                     computed.  The windowing function *window* is applied to each
(8)                     segment, and the amount of overlap of each segment is
(8)                     specified with *noverlap*. The spectrogram is plotted as a colormap
(8)                     (using imshow).
(8)                     Parameters
(8)                     ----------
(8)                     x : 1-D array or sequence
(12)                        Array or sequence containing the data.
(8)                     %(Spectral)s
(8)                     %(PSD)s
(8)                     mode : {'default', 'psd', 'magnitude', 'angle', 'phase'}
(12)                        What sort of spectrum to use.  Default is 'psd', which takes the
(12)                        power spectral density.  'magnitude' returns the magnitude
(12)                        spectrum.  'angle' returns the phase spectrum without unwrapping.
(12)                        'phase' returns the phase spectrum with unwrapping.
(8)                     noverlap : int, default: 128
(12)                        The number of points of overlap between blocks.
(8)                     scale : {'default', 'linear', 'dB'}
(12)                        The scaling of the values in the *spec*.  'linear' is no scaling.
(12)                        'dB' returns the values in dB scale.  When *mode* is 'psd',
(12)                        this is dB power (10 * log10).  Otherwise, this is dB amplitude
(12)                        (20 * log10). 'default' is 'dB' if *mode* is 'psd' or
(12)                        'magnitude' and 'linear' otherwise.  This must be 'linear'
(12)                        if *mode* is 'angle' or 'phase'.
(8)                     Fc : int, default: 0
(12)                        The center frequency of *x*, which offsets the x extents of the
(12)                        plot to reflect the frequency range used when a signal is acquired
(12)                        and then filtered and downsampled to baseband.
(8)                     cmap : `.Colormap`, default: :rc:`image.cmap`
(8)                     xextent : *None* or (xmin, xmax)
(12)                        The image extent along the x-axis. The default sets *xmin* to the
(12)                        left border of the first bin (*spectrum* column) and *xmax* to the
(12)                        right border of the last bin. Note that for *noverlap>0* the width
(12)                        of the bins is smaller than those of the segments.
(8)                     data : indexable object, optional
(12)                        DATA_PARAMETER_PLACEHOLDER
(8)                     **kwargs
(12)                        Additional keyword arguments are passed on to `~.axes.Axes.imshow`
(12)                        which makes the specgram image. The origin keyword argument
(12)                        is not supported.
(8)                     Returns
(8)                     -------
(8)                     spectrum : 2D array
(12)                        Columns are the periodograms of successive segments.
(8)                     freqs : 1-D array
(12)                        The frequencies corresponding to the rows in *spectrum*.
(8)                     t : 1-D array
(12)                        The times corresponding to midpoints of segments (i.e., the
columns

(12)                        in *spectrum*).
(8)                     im : `.AxesImage`
(12)                        The image created by imshow containing the spectrogram.
(8)                     See Also
(8)                     --------
(8)                     psd
(12)                        Differs in the default overlap; in returning the mean of the
(12)                        segment periodograms; in not returning times; and in generating a
(12)                        line plot instead of colormap.
(8)                     magnitude_spectrum
(12)                        A single spectrum, similar to having a single segment when *mode*
(12)                        is 'magnitude'. Plots a line instead of a colormap.
(8)                     angle_spectrum
(12)                        A single spectrum, similar to having a single segment when *mode*
(12)                        is 'angle'. Plots a line instead of a colormap.
(8)                     phase_spectrum
(12)                        A single spectrum, similar to having a single segment when *mode*
(12)                        is 'phase'. Plots a line instead of a colormap.
```

```
(8)                     Notes
(8)                     -----
(8)                     The parameters *detrend* and *scale_by_freq* do only apply when *mode*
(8)                     is set to 'psd'.
(8)                     """
(8)                     if NFFT is None:
(12)                        NFFT = 256  # same default as in mlab.specgram()
(8)                     if Fc is None:
(12)                        Fc = 0  # same default as in mlab._spectral_helper()
(8)                     if noverlap is None:
(12)                        noverlap = 128  # same default as in mlab.specgram()
(8)                     if Fs is None:
(12)                        Fs = 2  # same default as in mlab._spectral_helper()
(8)                     if mode == 'complex':
(12)                        raise ValueError('Cannot plot a complex specgram')
(8)                     if scale is None or scale == 'default':
(12)                        if mode in ['angle', 'phase']:
(16)                            scale = 'linear'
(12)                        else:
(16)                            scale = 'dB'
(8)                     elif mode in ['angle', 'phase'] and scale == 'dB':
(12)                        raise ValueError('Cannot use dB scale with angle or phase mode')
(8)                     spec, freqs, t = mlab.specgram(x=x, NFFT=NFFT, Fs=Fs,
(39)                                                   detrend=detrend, window=window,
(39)                                                   noverlap=noverlap, pad_to=pad_to,
(39)                                                   sides=sides,
(39)                                                   scale_by_freq=scale_by_freq,
(39)                                                   mode=mode)
(8)                     if scale == 'linear':
(12)                        Z = spec
(8)                     elif scale == 'dB':
(12)                        if mode is None or mode == 'default' or mode == 'psd':
(16)                            Z = 10. * np.log10(spec)
(12)                        else:
(16)                            Z = 20. * np.log10(spec)
(8)                     else:
(12)                        raise ValueError(f'Unknown scale {scale!r}')
(8)                     Z = np.flipud(Z)
(8)                     if xextent is None:
(12)                        pad_xextent = (NFFT-noverlap) / Fs / 2
(12)                        xextent = np.min(t) - pad_xextent, np.max(t) + pad_xextent
(8)                     xmin, xmax = xextent
(8)                     freqs += Fc
(8)                     extent = xmin, xmax, freqs[0], freqs[-1]
(8)                     if 'origin' in kwargs:
(12)                        raise _api.kwarg_error("specgram", "origin")
(8)                     im = self.imshow(Z, cmap, extent=extent, vmin=vmin, vmax=vmax,
(25)                                     origin='upper', **kwargs)
(8)                     self.axis('auto')
(8)                     return spec, freqs, t, im
(4)                 @_docstring.dedent_interpd
(4)                 def spy(self, Z, precision=0, marker=None, markersize=None,
(12)                        aspect='equal', origin="upper", **kwargs):
(8)                     """
(8)                     Plot the sparsity pattern of a 2D array.
(8)                     This visualizes the non-zero values of the array.
(8)                     Two plotting styles are available: image and marker. Both
(8)                     are available for full arrays, but only the marker style
(8)                     works for `scipy.sparse.spmatrix` instances.
(8)                     **Image style**
(8)                     If *marker* and *markersize* are *None*, `~.Axes.imshow` is used. Any
(8)                     extra remaining keyword arguments are passed to this method.
(8)                     **Marker style**
(8)                     If *Z* is a `scipy.sparse.spmatrix` or *marker* or *markersize* are
(8)                     *None*, a `.Line2D` object will be returned with the value of marker
(8)                     determining the marker type, and any remaining keyword arguments
(8)                     passed to `~.Axes.plot`.
(8)                     Parameters
(8)                     ----------
```

```
(8)                 Z : (M, N) array-like
(12)                    The array to be plotted.
(8)                 precision : float or 'present', default: 0
(12)                    If *precision* is 0, any non-zero value will be plotted.
Otherwise,

(12)                    values of :math:`|Z| > precision` will be plotted.
(12)                    For `scipy.sparse.spmatrix` instances, you can also
(12)                    pass 'present'. In this case any value present in the array
(12)                    will be plotted, even if it is identically zero.
(8)                 aspect : {'equal', 'auto', None} or float, default: 'equal'
(12)                    The aspect ratio of the Axes.  This parameter is particularly
(12)                    relevant for images since it determines whether data pixels are
(12)                    square.
(12)                    This parameter is a shortcut for explicitly calling
(12)                    `.Axes.set_aspect`. See there for further details.
(12)                    - 'equal': Ensures an aspect ratio of 1. Pixels will be square.
(12)                    - 'auto': The Axes is kept fixed and the aspect is adjusted so
(14)                      that the data fit in the Axes. In general, this will result in
(14)                      non-square pixels.
(12)                    - *None*: Use :rc:`image.aspect`.
(8)                 origin : {'upper', 'lower'}, default: :rc:`image.origin`
(12)                    Place the [0, 0] index of the array in the upper left or lower
left

(12)                    corner of the Axes. The convention 'upper' is typically used for
(12)                    matrices and images.
(8)                 Returns
(8)                 -------
(8)                 `~matplotlib.image.AxesImage` or `.Line2D`
(12)                    The return type depends on the plotting style (see above).
(8)                 Other Parameters
(8)                 ----------------
(8)                 **kwargs
(12)                    The supported additional parameters depend on the plotting style.
(12)                    For the image style, you can pass the following additional
(12)                    parameters of `~.Axes.imshow`:
(12)                    - *cmap*
(12)                    - *alpha*
(12)                    - *url*
(12)                    - any `.Artist` properties (passed on to the `.AxesImage`)
(12)                    For the marker style, you can pass any `.Line2D` property except
(12)                    for *linestyle*:
(12)                    %(Line2D:kwdoc)s
(8)                 """
(8)                 if marker is None and markersize is None and hasattr(Z, 'tocoo'):
(12)                    marker = 's'
(8)                 _api.check_in_list(["upper", "lower"], origin=origin)
(8)                 if marker is None and markersize is None:
(12)                    Z = np.asarray(Z)
(12)                    mask = np.abs(Z) > precision
(12)                    if 'cmap' not in kwargs:
(16)                        kwargs['cmap'] = mcolors.ListedColormap(['w', 'k'],
(56)                                                                name='binary')
(12)                    if 'interpolation' in kwargs:
(16)                        raise _api.kwarg_error("spy", "interpolation")
(12)                    if 'norm' not in kwargs:
(16)                        kwargs['norm'] = mcolors.NoNorm()
(12)                    ret = self.imshow(mask, interpolation='nearest',
(30)                                      aspect=aspect, origin=origin,
(30)                                      **kwargs)
(8)                 else:
(12)                    if hasattr(Z, 'tocoo'):
(16)                        c = Z.tocoo()
(16)                        if precision == 'present':
(20)                            y = c.row
(20)                            x = c.col
(16)                        else:
(20)                            nonzero = np.abs(c.data) > precision
(20)                            y = c.row[nonzero]
(20)                            x = c.col[nonzero]
```

```
(12)                        else:
(16)                            Z = np.asarray(Z)
(16)                            nonzero = np.abs(Z) > precision
(16)                            y, x = np.nonzero(nonzero)
(12)                        if marker is None:
(16)                            marker = 's'
(12)                        if markersize is None:
(16)                            markersize = 10
(12)                        if 'linestyle' in kwargs:
(16)                            raise _api.kwarg_error("spy", "linestyle")
(12)                        ret = mlines.Line2D(
(16)                            x, y, linestyle='None', marker=marker, markersize=markersize,
(16)                            **kwargs)
(12)                        self.add_line(ret)
(12)                        nr, nc = Z.shape
(12)                        self.set_xlim(-0.5, nc - 0.5)
(12)                        if origin == "upper":
(16)                            self.set_ylim(nr - 0.5, -0.5)
(12)                        else:
(16)                            self.set_ylim(-0.5, nr - 0.5)
(12)                        self.set_aspect(aspect)
(8)                     self.title.set_y(1.05)
(8)                     if origin == "upper":
(12)                        self.xaxis.tick_top()
(8)                     else:  # lower
(12)                        self.xaxis.tick_bottom()
(8)                     self.xaxis.set_ticks_position('both')
(8)                     self.xaxis.set_major_locator(
(12)                        mticker.MaxNLocator(nbins=9, steps=[1, 2, 5, 10], integer=True))
(8)                     self.yaxis.set_major_locator(
(12)                        mticker.MaxNLocator(nbins=9, steps=[1, 2, 5, 10], integer=True))
(8)                     return ret
(4)                 def matshow(self, Z, **kwargs):
(8)                     """
(8)                     Plot the values of a 2D matrix or array as color-coded image.
(8)                     The matrix will be shown the way it would be printed, with the first
(8)                     row at the top.  Row and column numbering is zero-based.
(8)                     Parameters
(8)                     ----------
(8)                     Z : (M, N) array-like
(12)                        The matrix to be displayed.
(8)                     Returns
(8)                     -------
(8)                     `~matplotlib.image.AxesImage`
(8)                     Other Parameters
(8)                     ----------------
(8)                     **kwargs : `~matplotlib.axes.Axes.imshow` arguments
(8)                     See Also
(8)                     --------
(8)                     imshow : More general function to plot data on a 2D regular raster.
(8)                     Notes
(8)                     -----
(8)                     This is just a convenience function wrapping `.imshow` to set useful
(8)                     defaults for displaying a matrix. In particular:
(8)                     - Set ``origin='upper'``.
(8)                     - Set ``interpolation='nearest'``.
(8)                     - Set ``aspect='equal'``.
(8)                     - Ticks are placed to the left and above.
(8)                     - Ticks are formatted to show integer indices.
(8)                     """
(8)                     Z = np.asanyarray(Z)
(8)                     kw = {'origin': 'upper',
(14)                          'interpolation': 'nearest',
(14)                          'aspect': 'equal',          # (already the imshow default)
(14)                          **kwargs}
(8)                     im = self.imshow(Z, **kw)
(8)                     self.title.set_y(1.05)
(8)                     self.xaxis.tick_top()
(8)                     self.xaxis.set_ticks_position('both')
```

```
(8)                 self.xaxis.set_major_locator(
(12)                    mticker.MaxNLocator(nbins=9, steps=[1, 2, 5, 10], integer=True))
(8)                 self.yaxis.set_major_locator(
(12)                    mticker.MaxNLocator(nbins=9, steps=[1, 2, 5, 10], integer=True))
(8)                 return im
(4)             @_preprocess_data(replace_names=["dataset"])
(4)             def violinplot(self, dataset, positions=None, vert=True, widths=0.5,
(19)                           showmeans=False, showextrema=True, showmedians=False,
(19)                           quantiles=None, points=100, bw_method=None, side='both'):
(8)                 """
(8)                 Make a violin plot.
(8)                 Make a violin plot for each column of *dataset* or each vector in
(8)                 sequence *dataset*.  Each filled area extends to represent the
(8)                 entire data range, with optional lines at the mean, the median,
(8)                 the minimum, the maximum, and user-specified quantiles.
(8)                 Parameters
(8)                 ----------
(8)                 dataset : Array or a sequence of vectors.
(10)                  The input data.
(8)                 positions : array-like, default: [1, 2, ..., n]
(10)                  The positions of the violins; i.e. coordinates on the x-axis for
(10)                  vertical violins (or y-axis for horizontal violins).
(8)                 vert : bool, default: True.
(10)                  If true, creates a vertical violin plot.
(10)                  Otherwise, creates a horizontal violin plot.
(8)                 widths : float or array-like, default: 0.5
(10)                  The maximum width of each violin in units of the *positions* axis.
(10)                  The default is 0.5, which is half the available space when using
default
(10)                  *positions*.
(8)                 showmeans : bool, default: False
(10)                  Whether to show the mean with a line.
(8)                 showextrema : bool, default: True
(10)                  Whether to show extrema with a line.
(8)                 showmedians : bool, default: False
(10)                  Whether to show the median with a line.
(8)                 quantiles : array-like, default: None
(10)                  If not None, set a list of floats in interval [0, 1] for each
violin,
(10)                  which stands for the quantiles that will be rendered for that
(10)                  violin.
(8)                 points : int, default: 100
(10)                  The number of points to evaluate each of the gaussian kernel density
(10)                  estimations at.
(8)                 bw_method : {'scott', 'silverman'} or float or callable, default:
'scott'
(10)                  The method used to calculate the estimator bandwidth.  If a
(10)                  float, this will be used directly as `kde.factor`.  If a
(10)                  callable, it should take a `matplotlib.mlab.GaussianKDE` instance as
(10)                  its only parameter and return a float.
(8)                 side : {'both', 'low', 'high'}, default: 'both'
(12)                    'both' plots standard violins. 'low'/'high' only
(12)                    plots the side below/above the positions value.
(8)                 data : indexable object, optional
(12)                    DATA_PARAMETER_PLACEHOLDER
(8)                 Returns
(8)                 -------
(8)                 dict
(10)                  A dictionary mapping each component of the violinplot to a
(10)                  list of the corresponding collection instances created. The
(10)                  dictionary has the following keys:
(10)                  - ``bodies``: A list of the `~.collections.PolyCollection`
(12)                    instances containing the filled area of each violin.
(10)                  - ``cmeans``: A `~.collections.LineCollection` instance that marks
(12)                    the mean values of each of the violin's distribution.
(10)                  - ``cmins``: A `~.collections.LineCollection` instance that marks
(12)                    the bottom of each violin's distribution.
(10)                  - ``cmaxes``: A `~.collections.LineCollection` instance that marks
(12)                    the top of each violin's distribution.
```

```
  6478: (10)                    - ``cbars``: A `~.collections.LineCollection` instance that marks
  6479: (12)                      the centers of each violin's distribution.
  6480: (10)                    - ``cmedians``: A `~.collections.LineCollection` instance that
  6481: (12)                      marks the median values of each of the violin's distribution.
  6482: (10)                    - ``cquantiles``: A `~.collections.LineCollection` instance created
  6483: (12)                      to identify the quantile values of each of the violin's
  6484: (12)                      distribution.
  6485: (8)                 See Also
  6486: (8)                 --------
  6487: (8)                 .Axes.violin : Draw a violin from pre-computed statistics.
  6488: (8)                 boxplot : Draw a box and whisker plot.
  6489: (8)                 """
  6490: (8)                 def _kde_method(X, coords):
  6491: (12)                    X = cbook._unpack_to_numpy(X)
  6492: (12)                    if np.all(X[0] == X):
  6493: (16)                        return (X[0] == coords).astype(float)
  6494: (12)                    kde = mlab.GaussianKDE(X, bw_method)
  6495: (12)                    return kde.evaluate(coords)
  6496: (8)                 vpstats = cbook.violin_stats(dataset, _kde_method, points=points,
  6497: (37)                                             quantiles=quantiles)
  6498: (8)                 return self.violin(vpstats, positions=positions, vert=vert,
  6499: (27)                                   widths=widths, showmeans=showmeans,
  6500: (27)                                   showextrema=showextrema, showmedians=showmedians,
  side=side)
  6501: (4)             def violin(self, vpstats, positions=None, vert=True, widths=0.5,
  6502: (15)                     showmeans=False, showextrema=True, showmedians=False,
  side='both'):
  6503: (8)                 """
  6504: (8)                 Draw a violin plot from pre-computed statistics.
  6505: (8)                 Draw a violin plot for each column of *vpstats*. Each filled area
  6506: (8)                 extends to represent the entire data range, with optional lines at the
  6507: (8)                 mean, the median, the minimum, the maximum, and the quantiles values.
  6508: (8)                 Parameters
  6509: (8)                 ----------
  6510: (8)                 vpstats : list of dicts
  6511: (10)                  A list of dictionaries containing stats for each violin plot.
  6512: (10)                  Required keys are:
  6513: (10)                  - ``coords``: A list of scalars containing the coordinates that
  6514: (12)                    the violin's kernel density estimate were evaluated at.
  6515: (10)                  - ``vals``: A list of scalars containing the values of the
  6516: (12)                    kernel density estimate at each of the coordinates given
  6517: (12)                    in *coords*.
  6518: (10)                  - ``mean``: The mean value for this violin's dataset.
  6519: (10)                  - ``median``: The median value for this violin's dataset.
  6520: (10)                  - ``min``: The minimum value for this violin's dataset.
  6521: (10)                  - ``max``: The maximum value for this violin's dataset.
  6522: (10)                  Optional keys are:
  6523: (10)                  - ``quantiles``: A list of scalars containing the quantile values
  6524: (12)                    for this violin's dataset.
  6525: (8)                 positions : array-like, default: [1, 2, ..., n]
  6526: (10)                  The positions of the violins; i.e. coordinates on the x-axis for
  6527: (10)                  vertical violins (or y-axis for horizontal violins).
  6528: (8)                 vert : bool, default: True.
  6529: (10)                  If true, plots the violins vertically.
  6530: (10)                  Otherwise, plots the violins horizontally.
  6531: (8)                 widths : float or array-like, default: 0.5
  6532: (10)                  The maximum width of each violin in units of the *positions* axis.
  6533: (10)                  The default is 0.5, which is half available space when using default
  6534: (10)                  *positions*.
  6535: (8)                 showmeans : bool, default: False
  6536: (10)                  Whether to show the mean with a line.
  6537: (8)                 showextrema : bool, default: True
  6538: (10)                  Whether to show extrema with a line.
  6539: (8)                 showmedians : bool, default: False
  6540: (10)                  Whether to show the median with a line.
  6541: (8)                 side : {'both', 'low', 'high'}, default: 'both'
  6542: (12)                    'both' plots standard violins. 'low'/'high' only
  6543: (12)                    plots the side below/above the positions value.
  6544: (8)                 Returns
```

```
        -------
        dict
          A dictionary mapping each component of the violinplot to a
          list of the corresponding collection instances created. The
          dictionary has the following keys:
          - ``bodies``: A list of the `~.collections.PolyCollection`
            instances containing the filled area of each violin.
          - ``cmeans``: A `~.collections.LineCollection` instance that marks
            the mean values of each of the violin's distribution.
          - ``cmins``: A `~.collections.LineCollection` instance that marks
            the bottom of each violin's distribution.
          - ``cmaxes``: A `~.collections.LineCollection` instance that marks
            the top of each violin's distribution.
          - ``cbars``: A `~.collections.LineCollection` instance that marks
            the centers of each violin's distribution.
          - ``cmedians``: A `~.collections.LineCollection` instance that
            marks the median values of each of the violin's distribution.
          - ``cquantiles``: A `~.collections.LineCollection` instance created
            to identify the quantiles values of each of the violin's
            distribution.
        See Also
        --------
        violin :
            Draw a violin plot from data instead of pre-computed statistics.
        """
        means = []
        mins = []
        maxes = []
        medians = []
        quantiles = []
        qlens = []  # Number of quantiles in each dataset.
        artists = {}  # Collections to be returned
        N = len(vpstats)
        datashape_message = ("List of violinplot statistics and `{0}` "
                             "values must have the same length")
        if positions is None:
            positions = range(1, N + 1)
        elif len(positions) != N:
            raise ValueError(datashape_message.format("positions"))
        if np.isscalar(widths):
            widths = [widths] * N
        elif len(widths) != N:
            raise ValueError(datashape_message.format("widths"))
        _api.check_in_list(["both", "low", "high"], side=side)
        line_ends = [[-0.25 if side in ['both', 'low'] else 0],
                     [0.25 if side in ['both', 'high'] else 0]] \
                         * np.array(widths) + positions
        if mpl.rcParams['_internal.classic_mode']:
            fillcolor = 'y'
            linecolor = 'r'
        else:
            fillcolor = linecolor = self._get_lines.get_next_color()
        if vert:
            fill = self.fill_betweenx
            if side in ['low', 'high']:
                perp_lines = functools.partial(self.hlines, colors=linecolor,
                                               capstyle='projecting')
                par_lines = functools.partial(self.vlines, colors=linecolor,
                                              capstyle='projecting')
            else:
                perp_lines = functools.partial(self.hlines, colors=linecolor)
                par_lines = functools.partial(self.vlines, colors=linecolor)
        else:
            fill = self.fill_between
            if side in ['low', 'high']:
                perp_lines = functools.partial(self.vlines, colors=linecolor,
                                               capstyle='projecting')
                par_lines = functools.partial(self.hlines, colors=linecolor,
                                              capstyle='projecting')
```

```
6614: (12)                      else:
6615: (16)                          perp_lines = functools.partial(self.vlines, colors=linecolor)
6616: (16)                          par_lines = functools.partial(self.hlines, colors=linecolor)
6617: (8)                   bodies = []
6618: (8)                   for stats, pos, width in zip(vpstats, positions, widths):
6619: (12)                      vals = np.array(stats['vals'])
6620: (12)                      vals = 0.5 * width * vals / vals.max()
6621: (12)                      bodies += [fill(stats['coords'],
6622: (28)                                      -vals + pos if side in ['both', 'low'] else pos,
6623: (28)                                      vals + pos if side in ['both', 'high'] else pos,
6624: (28)                                      facecolor=fillcolor, alpha=0.3)]
6625: (12)                      means.append(stats['mean'])
6626: (12)                      mins.append(stats['min'])
6627: (12)                      maxes.append(stats['max'])
6628: (12)                      medians.append(stats['median'])
6629: (12)                      q = stats.get('quantiles')  # a list of floats, or None
6630: (12)                      if q is None:
6631: (16)                          q = []
6632: (12)                      quantiles.extend(q)
6633: (12)                      qlens.append(len(q))
6634: (8)                   artists['bodies'] = bodies
6635: (8)                   if showmeans:  # Render means
6636: (12)                      artists['cmeans'] = perp_lines(means, *line_ends)
6637: (8)                   if showextrema:  # Render extrema
6638: (12)                      artists['cmaxes'] = perp_lines(maxes, *line_ends)
6639: (12)                      artists['cmins'] = perp_lines(mins, *line_ends)
6640: (12)                      artists['cbars'] = par_lines(positions, mins, maxes)
6641: (8)                   if showmedians:  # Render medians
6642: (12)                      artists['cmedians'] = perp_lines(medians, *line_ends)
6643: (8)                   if quantiles:  # Render quantiles: each width is repeated qlen times.
6644: (12)                      artists['cquantiles'] = perp_lines(
6645: (16)                          quantiles, *np.repeat(line_ends, qlens, axis=1))
6646: (8)                   return artists
6647: (4)               table = _make_axes_method(mtable.table)
6648: (4)               stackplot = _preprocess_data()(_make_axes_method(mstack.stackplot))
6649: (4)               streamplot = _preprocess_data(
6650: (12)                      replace_names=["x", "y", "u", "v", "start_points"])(
6651: (8)                   _make_axes_method(mstream.streamplot))
6652: (4)               tricontour = _make_axes_method(mtri.tricontour)
6653: (4)               tricontourf = _make_axes_method(mtri.tricontourf)
6654: (4)               tripcolor = _make_axes_method(mtri.tripcolor)
6655: (4)               triplot = _make_axes_method(mtri.triplot)
6656: (4)               def _get_aspect_ratio(self):
6657: (8)                   """
6658: (8)                   Convenience method to calculate the aspect ratio of the Axes in
6659: (8)                   the display coordinate system.
6660: (8)                   """
6661: (8)                   figure_size = self.get_figure().get_size_inches()
6662: (8)                   ll, ur = self.get_position() * figure_size
6663: (8)                   width, height = ur - ll
6664: (8)                   return height / (width * self.get_data_ratio())
```

File 76 - _base.py:

```
1: (0)              from collections.abc import Iterable, Sequence
2: (0)              from contextlib import ExitStack
3: (0)              import functools
4: (0)              import inspect
5: (0)              import logging
6: (0)              from numbers import Real
7: (0)              from operator import attrgetter
8: (0)              import re
9: (0)              import types
10: (0)             import numpy as np
11: (0)             import matplotlib as mpl
12: (0)             from matplotlib import _api, cbook, _docstring, offsetbox
13: (0)             import matplotlib.artist as martist
```

```
 14: (0)             import matplotlib.axis as maxis
 15: (0)             from matplotlib.cbook import _OrderedSet, _check_1d, index_of
 16: (0)             import matplotlib.collections as mcoll
 17: (0)             import matplotlib.colors as mcolors
 18: (0)             import matplotlib.font_manager as font_manager
 19: (0)             from matplotlib.gridspec import SubplotSpec
 20: (0)             import matplotlib.image as mimage
 21: (0)             import matplotlib.lines as mlines
 22: (0)             import matplotlib.patches as mpatches
 23: (0)             from matplotlib.rcsetup import cycler, validate_axisbelow
 24: (0)             import matplotlib.spines as mspines
 25: (0)             import matplotlib.table as mtable
 26: (0)             import matplotlib.text as mtext
 27: (0)             import matplotlib.ticker as mticker
 28: (0)             import matplotlib.transforms as mtransforms
 29: (0)             _log = logging.getLogger(__name__)
 30: (0)             class _axis_method_wrapper:
 31: (4)                 """
 32: (4)                 Helper to generate Axes methods wrapping Axis methods.
 33: (4)                 After ::
 34: (8)                     get_foo = _axis_method_wrapper("xaxis", "get_bar")
 35: (4)                 (in the body of a class) ``get_foo`` is a method that forwards it
arguments
 36: (4)                 to the ``get_bar`` method of the ``xaxis`` attribute, and gets its
 37: (4)                 signature and docstring from ``Axis.get_bar``.
 38: (4)                 The docstring of ``get_foo`` is built by replacing "this Axis" by "the
 39: (4)                 {attr_name}" (i.e., "the xaxis", "the yaxis") in the wrapped method's
 40: (4)                 dedented docstring; additional replacements can be given in *doc_sub*.
 41: (4)                 """
 42: (4)                 def __init__(self, attr_name, method_name, *, doc_sub=None):
 43: (8)                     self.attr_name = attr_name
 44: (8)                     self.method_name = method_name
 45: (8)                     doc = inspect.getdoc(getattr(maxis.Axis, method_name))
 46: (8)                     self._missing_subs = []
 47: (8)                     if doc:
 48: (12)                        doc_sub = {"this Axis": f"the {self.attr_name}", **(doc_sub or
{})}
 49: (12)                        for k, v in doc_sub.items():
 50: (16)                            if k not in doc:  # Delay raising error until we know
qualname.
 51: (20)                                self._missing_subs.append(k)
 52: (16)                            doc = doc.replace(k, v)
 53: (8)                     self.__doc__ = doc
 54: (4)                 def __set_name__(self, owner, name):
 55: (8)                     get_method = attrgetter(f"{self.attr_name}.{self.method_name}")
 56: (8)                     def wrapper(self, *args, **kwargs):
 57: (12)                        return get_method(self)(*args, **kwargs)
 58: (8)                     wrapper.__module__ = owner.__module__
 59: (8)                     wrapper.__name__ = name
 60: (8)                     wrapper.__qualname__ = f"{owner.__qualname__}.{name}"
 61: (8)                     wrapper.__doc__ = self.__doc__
 62: (8)                     wrapper.__signature__ = inspect.signature(
 63: (12)                        getattr(maxis.Axis, self.method_name))
 64: (8)                     if self._missing_subs:
 65: (12)                        raise ValueError(
 66: (16)                            "The definition of {} expected that the docstring of Axis.{} "
 67: (16)                            "contains {!r} as substrings".format(
 68: (20)                                wrapper.__qualname__, self.method_name,
 69: (20)                                ", ".join(map(repr, self._missing_subs))))
 70: (8)                     setattr(owner, name, wrapper)
 71: (0)             class _TransformedBoundsLocator:
 72: (4)                 """
 73: (4)                 Axes locator for `.Axes.inset_axes` and similarly positioned Axes.
 74: (4)                 The locator is a callable object used in `.Axes.set_aspect` to compute the
 75: (4)                 Axes location depending on the renderer.
 76: (4)                 """
 77: (4)                 def __init__(self, bounds, transform):
 78: (8)                     """
 79: (8)                     *bounds* (a ``[l, b, w, h]`` rectangle) and *transform* together
```

```
 80: (8)                    specify the position of the inset Axes.
 81: (8)                    """
 82: (8)                    self._bounds = bounds
 83: (8)                    self._transform = transform
 84: (4)                def __call__(self, ax, renderer):
 85: (8)                    return mtransforms.TransformedBbox(
 86: (12)                       mtransforms.Bbox.from_bounds(*self._bounds),
 87: (12)                       self._transform - ax.figure.transSubfigure)
 88: (0)            def _process_plot_format(fmt, *, ambiguous_fmt_datakey=False):
 89: (4)                """
 90: (4)                Convert a MATLAB style color/line style format string to a (*linestyle*,
 91: (4)                *marker*, *color*) tuple.
 92: (4)                Example format strings include:
 93: (4)                * 'ko': black circles
 94: (4)                * '.b': blue dots
 95: (4)                * 'r--': red dashed lines
 96: (4)                * 'C2--': the third color in the color cycle, dashed lines
 97: (4)                The format is absolute in the sense that if a linestyle or marker is not
 98: (4)                defined in *fmt*, there is no line or marker. This is expressed by
 99: (4)                returning 'None' for the respective quantity.
 100: (4)               See Also
 101: (4)               --------
 102: (4)               matplotlib.Line2D.lineStyles, matplotlib.colors.cnames
 103: (8)                   All possible styles and color format strings.
 104: (4)               """
 105: (4)               linestyle = None
 106: (4)               marker = None
 107: (4)               color = None
 108: (4)               if fmt not in ["0", "1"]:
 109: (8)                   try:
 110: (12)                      color = mcolors.to_rgba(fmt)
 111: (12)                      return linestyle, marker, color
 112: (8)                   except ValueError:
 113: (12)                      pass
 114: (4)               errfmt = ("{!r} is neither a data key nor a valid format string ({})"
 115: (14)                         if ambiguous_fmt_datakey else
 116: (14)                         "{!r} is not a valid format string ({})")
 117: (4)               i = 0
 118: (4)               while i < len(fmt):
 119: (8)                   c = fmt[i]
 120: (8)                   if fmt[i:i+2] in mlines.lineStyles:  # First, the two-char styles.
 121: (12)                      if linestyle is not None:
 122: (16)                          raise ValueError(errfmt.format(fmt, "two linestyle symbols"))
 123: (12)                      linestyle = fmt[i:i+2]
 124: (12)                      i += 2
 125: (8)                   elif c in mlines.lineStyles:
 126: (12)                      if linestyle is not None:
 127: (16)                          raise ValueError(errfmt.format(fmt, "two linestyle symbols"))
 128: (12)                      linestyle = c
 129: (12)                      i += 1
 130: (8)                   elif c in mlines.lineMarkers:
 131: (12)                      if marker is not None:
 132: (16)                          raise ValueError(errfmt.format(fmt, "two marker symbols"))
 133: (12)                      marker = c
 134: (12)                      i += 1
 135: (8)                   elif c in mcolors.get_named_colors_mapping():
 136: (12)                      if color is not None:
 137: (16)                          raise ValueError(errfmt.format(fmt, "two color symbols"))
 138: (12)                      color = c
 139: (12)                      i += 1
 140: (8)                   elif c == "C":
 141: (12)                      cn_color = re.match(r"C\d+", fmt[i:])
 142: (12)                      if not cn_color:
 143: (16)                          raise ValueError(errfmt.format(fmt, "'C' must be followed by a
 number"))
 144: (12)                      color = mcolors.to_rgba(cn_color[0])
 145: (12)                      i += len(cn_color[0])
 146: (8)                   else:
 147: (12)                      raise ValueError(errfmt.format(fmt, f"unrecognized character
```

```
{c!r}"))
148: (4)                if linestyle is None and marker is None:
149: (8)                    linestyle = mpl.rcParams['lines.linestyle']
150: (4)                if linestyle is None:
151: (8)                    linestyle = 'None'
152: (4)                if marker is None:
153: (8)                    marker = 'None'
154: (4)                return linestyle, marker, color
155: (0)            class _process_plot_var_args:
156: (4)                """
157: (4)                Process variable length arguments to `~.Axes.plot`, to support ::
158: (6)                  plot(t, s)
159: (6)                  plot(t1, s1, t2, s2)
160: (6)                  plot(t1, s1, 'ko', t2, s2)
161: (6)                  plot(t1, s1, 'ko', t2, s2, 'r--', t3, e3)
162: (4)                an arbitrary number of *x*, *y*, *fmt* are allowed
163: (4)                """
164: (4)                def __init__(self, command='plot'):
165: (8)                    self.command = command
166: (8)                    self.set_prop_cycle(None)
167: (4)                def set_prop_cycle(self, cycler):
168: (8)                    if cycler is None:
169: (12)                       cycler = mpl.rcParams['axes.prop_cycle']
170: (8)                    self._idx = 0
171: (8)                    self._cycler_items = [*cycler]
172: (4)                def __call__(self, axes, *args, data=None, **kwargs):
173: (8)                    axes._process_unit_info(kwargs=kwargs)
174: (8)                    for pos_only in "xy":
175: (12)                       if pos_only in kwargs:
176: (16)                           raise _api.kwarg_error(self.command, pos_only)
177: (8)                    if not args:
178: (12)                       return
179: (8)                    if data is None:  # Process dict views
180: (12)                       args = [cbook.sanitize_sequence(a) for a in args]
181: (8)                    else:  # Process the 'data' kwarg.
182: (12)                       replaced = [mpl._replacer(data, arg) for arg in args]
183: (12)                       if len(args) == 1:
184: (16)                           label_namer_idx = 0
185: (12)                       elif len(args) == 2:  # Can be x, y or y, c.
186: (16)                           try:
187: (20)                               _process_plot_format(args[1])
188: (16)                           except ValueError:  # case 1)
189: (20)                               label_namer_idx = 1
190: (16)                           else:
191: (20)                               if replaced[1] is not args[1]:  # case 2a)
192: (24)                                   _api.warn_external(
193: (28)                                       f"Second argument {args[1]!r} is ambiguous: could
"
194: (28)                                       f"be a format string but is in 'data'; using as "
195: (28)                                       f"data.  If it was intended as data, set the "
196: (28)                                       f"format string to an empty string to suppress "
197: (28)                                       f"this warning.  If it was intended as a format "
198: (28)                                       f"string, explicitly pass the x-values as well.  "
199: (28)                                       f"Alternatively, rename the entry in 'data'.",
200: (28)                                       RuntimeWarning)
201: (24)                                   label_namer_idx = 1
202: (20)                               else:  # case 2b)
203: (24)                                   label_namer_idx = 0
204: (12)                       elif len(args) == 3:
205: (16)                           label_namer_idx = 1
206: (12)                       else:
207: (16)                           raise ValueError(
208: (20)                               "Using arbitrary long args with data is not supported due
"
209: (20)                               "to ambiguity of arguments; use multiple plotting calls "
210: (20)                               "instead")
211: (12)                       if kwargs.get("label") is None:
212: (16)                           kwargs["label"] = mpl._label_from_arg(
213: (20)                               replaced[label_namer_idx], args[label_namer_idx])
```

```
214: (12)                        args = replaced
215: (8)                    ambiguous_fmt_datakey = data is not None and len(args) == 2
216: (8)                    if len(args) >= 4 and not cbook.is_scalar_or_string(
217: (16)                           kwargs.get("label")):
218: (12)                       raise ValueError("plot() with multiple groups of data (i.e., "
219: (29)                                        "pairs of x and y) does not support multiple "
220: (29)                                        "labels")
221: (8)                    while args:
222: (12)                       this, args = args[:2], args[2:]
223: (12)                       if args and isinstance(args[0], str):
224: (16)                           this += args[0],
225: (16)                           args = args[1:]
226: (12)                       yield from self._plot_args(
227: (16)                           axes, this, kwargs,
ambiguous_fmt_datakey=ambiguous_fmt_datakey)
228: (4)                def get_next_color(self):
229: (8)                    """Return the next color in the cycle."""
230: (8)                    entry = self._cycler_items[self._idx]
231: (8)                    if "color" in entry:
232: (12)                       self._idx = (self._idx + 1) % len(self._cycler_items)  # Advance
cycler.
233: (12)                       return entry["color"]
234: (8)                    else:
235: (12)                       return "k"
236: (4)                def _getdefaults(self, kw, ignore=frozenset()):
237: (8)                    """
238: (8)                    If some keys in the property cycle (excluding those in the set
239: (8)                    *ignore*) are absent or set to None in the dict *kw*, return a copy
240: (8)                    of the next entry in the property cycle, excluding keys in *ignore*.
241: (8)                    Otherwise, don't advance the property cycle, and return an empty dict.
242: (8)                    """
243: (8)                    defaults = self._cycler_items[self._idx]
244: (8)                    if any(kw.get(k, None) is None for k in {*defaults} - ignore):
245: (12)                       self._idx = (self._idx + 1) % len(self._cycler_items)  # Advance
cycler.
246: (12)                       return {k: v for k, v in defaults.items() if k not in ignore}
247: (8)                    else:
248: (12)                       return {}
249: (4)                def _setdefaults(self, defaults, kw):
250: (8)                    """
251: (8)                    Add to the dict *kw* the entries in the dict *default* that are absent
252: (8)                    or set to None in *kw*.
253: (8)                    """
254: (8)                    for k in defaults:
255: (12)                       if kw.get(k, None) is None:
256: (16)                           kw[k] = defaults[k]
257: (4)                def _makeline(self, axes, x, y, kw, kwargs):
258: (8)                    kw = {**kw, **kwargs}  # Don't modify the original kw.
259: (8)                    self._setdefaults(self._getdefaults(kw), kw)
260: (8)                    seg = mlines.Line2D(x, y, **kw)
261: (8)                    return seg, kw
262: (4)                def _makefill(self, axes, x, y, kw, kwargs):
263: (8)                    x = axes.convert_xunits(x)
264: (8)                    y = axes.convert_yunits(y)
265: (8)                    kw = kw.copy()  # Don't modify the original kw.
266: (8)                    kwargs = kwargs.copy()
267: (8)                    ignores = ({'marker', 'markersize', 'markeredgecolor',
268: (20)                               'markerfacecolor', 'markeredgewidth'}
269: (19)                              | {k for k, v in kwargs.items() if v is not None})
270: (8)                    default_dict = self._getdefaults(kw, ignores)
271: (8)                    self._setdefaults(default_dict, kw)
272: (8)                    facecolor = kw.get('color', None)
273: (8)                    default_dict.pop('color', None)
274: (8)                    self._setdefaults(default_dict, kwargs)
275: (8)                    seg = mpatches.Polygon(np.column_stack((x, y)),
276: (31)                                          facecolor=facecolor,
277: (31)                                          fill=kwargs.get('fill', True),
278: (31)                                          closed=kw['closed'])
279: (8)                    seg.set(**kwargs)
```

```
(8)                     return seg, kwargs
(4)                 def _plot_args(self, axes, tup, kwargs, *,
(19)                                   return_kwargs=False, ambiguous_fmt_datakey=False):
(8)                     """
(8)                     Process the arguments of ``plot([x], y, [fmt], **kwargs)`` calls.
(8)                     This processes a single set of ([x], y, [fmt]) parameters; i.e. for
(8)                     ``plot(x, y, x2, y2)`` it will be called twice. Once for (x, y) and
(8)                     once for (x2, y2).
(8)                     x and y may be 2D and thus can still represent multiple datasets.
(8)                     For multiple datasets, if the keyword argument *label* is a list, this
(8)                     will unpack the list and assign the individual labels to the datasets.
(8)                     Parameters
(8)                     ----------
(8)                     tup : tuple
(12)                        A tuple of the positional parameters. This can be one of
(12)                        - (y,)
(12)                        - (x, y)
(12)                        - (y, fmt)
(12)                        - (x, y, fmt)
(8)                     kwargs : dict
(12)                        The keyword arguments passed to ``plot()``.
(8)                     return_kwargs : bool
(12)                        Whether to also return the effective keyword arguments after label
(12)                        unpacking as well.
(8)                     ambiguous_fmt_datakey : bool
(12)                        Whether the format string in *tup* could also have been a
(12)                        misspelled data key.
(8)                     Returns
(8)                     -------
(8)                     result
(12)                        If *return_kwargs* is false, a list of Artists representing the
(12)                        dataset(s).
(12)                        If *return_kwargs* is true, a list of (Artist, effective_kwargs)
(12)                        representing the dataset(s). See *return_kwargs*.
(12)                        The Artist is either `.Line2D` (if called from ``plot()``) or
(12)                        `.Polygon` otherwise.
(8)                     """
(8)                     if len(tup) > 1 and isinstance(tup[-1], str):
(12)                        *xy, fmt = tup
(12)                        linestyle, marker, color = _process_plot_format(
(16)                            fmt, ambiguous_fmt_datakey=ambiguous_fmt_datakey)
(8)                     elif len(tup) == 3:
(12)                        raise ValueError('third arg must be a format string')
(8)                     else:
(12)                        xy = tup
(12)                        linestyle, marker, color = None, None, None
(8)                     if any(v is None for v in tup):
(12)                        raise ValueError("x, y, and format string must not be None")
(8)                     kw = {}
(8)                     for prop_name, val in zip(('linestyle', 'marker', 'color'),
(34)                                              (linestyle, marker, color)):
(12)                        if val is not None:
(16)                            if (fmt.lower() != 'none'
(24)                                    and prop_name in kwargs
(24)                                    and val != 'None'):
(20)                                _api.warn_external(
(24)                                    f"{prop_name} is redundantly defined by the "
(24)                                    f"'{prop_name}' keyword argument and the fmt string "
(24)                                    f'"{fmt}" (-> {prop_name}={val!r}). The keyword '
(24)                                    f"argument will take precedence.")
(16)                            kw[prop_name] = val
(8)                     if len(xy) == 2:
(12)                        x = _check_1d(xy[0])
(12)                        y = _check_1d(xy[1])
(8)                     else:
(12)                        x, y = index_of(xy[-1])
(8)                     if axes.xaxis is not None:
(12)                        axes.xaxis.update_units(x)
(8)                     if axes.yaxis is not None:
```

```
349: (12)                        axes.yaxis.update_units(y)
350: (8)                    if x.shape[0] != y.shape[0]:
351: (12)                        raise ValueError(f"x and y must have same first dimension, but "
352: (29)                                         f"have shapes {x.shape} and {y.shape}")
353: (8)                    if x.ndim > 2 or y.ndim > 2:
354: (12)                        raise ValueError(f"x and y can be no greater than 2D, but have "
355: (29)                                         f"shapes {x.shape} and {y.shape}")
356: (8)                    if x.ndim == 1:
357: (12)                        x = x[:, np.newaxis]
358: (8)                    if y.ndim == 1:
359: (12)                        y = y[:, np.newaxis]
360: (8)                    if self.command == 'plot':
361: (12)                        make_artist = self._makeline
362: (8)                    else:
363: (12)                        kw['closed'] = kwargs.get('closed', True)
364: (12)                        make_artist = self._makefill
365: (8)                    ncx, ncy = x.shape[1], y.shape[1]
366: (8)                    if ncx > 1 and ncy > 1 and ncx != ncy:
367: (12)                        raise ValueError(f"x has {ncx} columns but y has {ncy} columns")
368: (8)                    if ncx == 0 or ncy == 0:
369: (12)                        return []
370: (8)                    label = kwargs.get('label')
371: (8)                    n_datasets = max(ncx, ncy)
372: (8)                    if cbook.is_scalar_or_string(label):
373: (12)                        labels = [label] * n_datasets
374: (8)                    elif len(label) == n_datasets:
375: (12)                        labels = label
376: (8)                    elif n_datasets == 1:
377: (12)                        msg = (f'Passing label as a length {len(label)} sequence when '
378: (20)                               'plotting a single dataset is deprecated in Matplotlib 3.9
'
379: (20)                               'and will error in 3.11.  To keep the current behavior, '
380: (20)                               'cast the sequence to string before passing.')
381: (12)                        _api.warn_deprecated('3.9', message=msg)
382: (12)                        labels = [label]
383: (8)                    else:
384: (12)                        raise ValueError(
385: (16)                            f"label must be scalar or have the same length as the input "
386: (16)                            f"data, but found {len(label)} for {n_datasets} datasets.")
387: (8)                    result = (make_artist(axes, x[:, j % ncx], y[:, j % ncy], kw,
388: (30)                                          {**kwargs, 'label': label})
389: (18)                             for j, label in enumerate(labels))
390: (8)                    if return_kwargs:
391: (12)                        return list(result)
392: (8)                    else:
393: (12)                        return [l[0] for l in result]
394: (0)            @_api.define_aliases({"facecolor": ["fc"]})
395: (0)            class _AxesBase(martist.Artist):
396: (4)                name = "rectilinear"
397: (4)                _axis_names = ("x", "y")
398: (4)                _shared_axes = {name: cbook.Grouper() for name in _axis_names}
399: (4)                _twinned_axes = cbook.Grouper()
400: (4)                _subclass_uses_cla = False
401: (4)                @property
402: (4)                def _axis_map(self):
403: (8)                    """A mapping of axis names, e.g. 'x', to `Axis` instances."""
404: (8)                    return {name: getattr(self, f"{name}axis")
405: (16)                           for name in self._axis_names}
406: (4)                def __str__(self):
407: (8)                    return "{0}({1[0]:g},{1[1]:g};{1[2]:g}x{1[3]:g})".format(
408: (12)                       type(self).__name__, self._position.bounds)
409: (4)                def __init__(self, fig,
410: (17)                            *args,
411: (17)                            facecolor=None,  # defaults to rc axes.facecolor
412: (17)                            frameon=True,
413: (17)                            sharex=None,  # use Axes instance's xaxis info
414: (17)                            sharey=None,  # use Axes instance's yaxis info
415: (17)                            label='',
416: (17)                            xscale=None,
```

```
417: (17)                                yscale=None,
418: (17)                                box_aspect=None,
419: (17)                                forward_navigation_events="auto",
420: (17)                                **kwargs
421: (17)                                ):
422: (8)                        """
423: (8)                        Build an Axes in a figure.
424: (8)                        Parameters
425: (8)                        ----------
426: (8)                        fig : `~matplotlib.figure.Figure`
427: (12)                           The Axes is built in the `.Figure` *fig*.
428: (8)                        *args
429: (12)                           ``*args`` can be a single ``(left, bottom, width, height)``
430: (12)                           rectangle or a single `.Bbox`.  This specifies the rectangle (in
431: (12)                           figure coordinates) where the Axes is positioned.
432: (12)                           ``*args`` can also consist of three numbers or a single three-
digit
433: (12)                           number; in the latter case, the digits are considered as
434: (12)                           independent numbers.  The numbers are interpreted as ``(nrows,
435: (12)                           ncols, index)``: ``(nrows, ncols)`` specifies the size of an array
436: (12)                           of subplots, and ``index`` is the 1-based index of the subplot
437: (12)                           being created.  Finally, ``*args`` can also directly be a
438: (12)                           `.SubplotSpec` instance.
439: (8)                        sharex, sharey : `~matplotlib.axes.Axes`, optional
440: (12)                           The x- or y-`~.matplotlib.axis` is shared with the x- or y-axis in
441: (12)                           the input `~.axes.Axes`.  Note that it is not possible to unshare
442: (12)                           axes.
443: (8)                        frameon : bool, default: True
444: (12)                           Whether the Axes frame is visible.
445: (8)                        box_aspect : float, optional
446: (12)                           Set a fixed aspect for the Axes box, i.e. the ratio of height to
447: (12)                           width. See `~.axes.Axes.set_box_aspect` for details.
448: (8)                        forward_navigation_events : bool or "auto", default: "auto"
449: (12)                           Control whether pan/zoom events are passed through to Axes below
450: (12)                           this one. "auto" is *True* for axes with an invisible patch and
451: (12)                           *False* otherwise.
452: (8)                        **kwargs
453: (12)                           Other optional keyword arguments:
454: (12)                           %(Axes:kwdoc)s
455: (8)                        Returns
456: (8)                        -------
457: (8)                        `~.axes.Axes`
458: (12)                           The new `~.axes.Axes` object.
459: (8)                        """
460: (8)                        super().__init__()
461: (8)                        if "rect" in kwargs:
462: (12)                           if args:
463: (16)                               raise TypeError(
464: (20)                                   "'rect' cannot be used together with positional
arguments")
465: (12)                           rect = kwargs.pop("rect")
466: (12)                           _api.check_isinstance((mtransforms.Bbox, Iterable), rect=rect)
467: (12)                           args = (rect,)
468: (8)                        subplotspec = None
469: (8)                        if len(args) == 1 and isinstance(args[0], mtransforms.Bbox):
470: (12)                           self._position = args[0]
471: (8)                        elif len(args) == 1 and np.iterable(args[0]):
472: (12)                           self._position = mtransforms.Bbox.from_bounds(*args[0])
473: (8)                        else:
474: (12)                           self._position = self._originalPosition = mtransforms.Bbox.unit()
475: (12)                           subplotspec = SubplotSpec._from_subplot_args(fig, args)
476: (8)                        if self._position.width < 0 or self._position.height < 0:
477: (12)                           raise ValueError('Width and height specified must be non-
negative')
478: (8)                        self._originalPosition = self._position.frozen()
479: (8)                        self.axes = self
480: (8)                        self._aspect = 'auto'
481: (8)                        self._adjustable = 'box'
482: (8)                        self._anchor = 'C'
```

```
483: (8)                    self._stale_viewlims = {name: False for name in self._axis_names}
484: (8)                    self._forward_navigation_events = forward_navigation_events
485: (8)                    self._sharex = sharex
486: (8)                    self._sharey = sharey
487: (8)                    self.set_label(label)
488: (8)                    self.set_figure(fig)
489: (8)                    if subplotspec:
490: (12)                       self.set_subplotspec(subplotspec)
491: (8)                    else:
492: (12)                       self._subplotspec = None
493: (8)                    self.set_box_aspect(box_aspect)
494: (8)                    self._axes_locator = None  # Optionally set via update(kwargs).
495: (8)                    self._children = []
496: (8)                    self._colorbars = []
497: (8)                    self.spines = mspines.Spines.from_dict(self._gen_axes_spines())
498: (8)                    self._init_axis()
499: (8)                    if facecolor is None:
500: (12)                       facecolor = mpl.rcParams['axes.facecolor']
501: (8)                    self._facecolor = facecolor
502: (8)                    self._frameon = frameon
503: (8)                    self.set_axisbelow(mpl.rcParams['axes.axisbelow'])
504: (8)                    self._rasterization_zorder = None
505: (8)                    self.clear()
506: (8)                    self.fmt_xdata = None
507: (8)                    self.fmt_ydata = None
508: (8)                    self.set_navigate(True)
509: (8)                    self.set_navigate_mode(None)
510: (8)                    if xscale:
511: (12)                       self.set_xscale(xscale)
512: (8)                    if yscale:
513: (12)                       self.set_yscale(yscale)
514: (8)                    self._internal_update(kwargs)
515: (8)                    for name, axis in self._axis_map.items():
516: (12)                       axis.callbacks._connect_picklable(
517: (16)                           'units', self._unit_change_handler(name))
518: (8)                    rcParams = mpl.rcParams
519: (8)                    self.tick_params(
520: (12)                       top=rcParams['xtick.top'] and rcParams['xtick.minor.top'],
521: (12)                       bottom=rcParams['xtick.bottom'] and
rcParams['xtick.minor.bottom'],
522: (12)                       labeltop=(rcParams['xtick.labeltop'] and
523: (22)                                 rcParams['xtick.minor.top']),
524: (12)                       labelbottom=(rcParams['xtick.labelbottom'] and
525: (25)                                    rcParams['xtick.minor.bottom']),
526: (12)                       left=rcParams['ytick.left'] and rcParams['ytick.minor.left'],
527: (12)                       right=rcParams['ytick.right'] and rcParams['ytick.minor.right'],
528: (12)                       labelleft=(rcParams['ytick.labelleft'] and
529: (23)                                  rcParams['ytick.minor.left']),
530: (12)                       labelright=(rcParams['ytick.labelright'] and
531: (24)                                   rcParams['ytick.minor.right']),
532: (12)                       which='minor')
533: (8)                    self.tick_params(
534: (12)                       top=rcParams['xtick.top'] and rcParams['xtick.major.top'],
535: (12)                       bottom=rcParams['xtick.bottom'] and
rcParams['xtick.major.bottom'],
536: (12)                       labeltop=(rcParams['xtick.labeltop'] and
537: (22)                                 rcParams['xtick.major.top']),
538: (12)                       labelbottom=(rcParams['xtick.labelbottom'] and
539: (25)                                    rcParams['xtick.major.bottom']),
540: (12)                       left=rcParams['ytick.left'] and rcParams['ytick.major.left'],
541: (12)                       right=rcParams['ytick.right'] and rcParams['ytick.major.right'],
542: (12)                       labelleft=(rcParams['ytick.labelleft'] and
543: (23)                                  rcParams['ytick.major.left']),
544: (12)                       labelright=(rcParams['ytick.labelright'] and
545: (24)                                   rcParams['ytick.major.right']),
546: (12)                       which='major')
547: (4)                def __init_subclass__(cls, **kwargs):
548: (8)                    parent_uses_cla = super(cls, cls)._subclass_uses_cla
549: (8)                    if 'cla' in cls.__dict__:
```

```
(12)                        _api.warn_deprecated(
(16)                            '3.6',
(16)                            pending=True,
(16)                            message=f'Overriding `Axes.cla` in {cls.__qualname__} is '
(16)                            'pending deprecation in %(since)s and will be fully '
(16)                            'deprecated in favor of `Axes.clear` in the future. '
(16)                            'Please report '
(16)                            f'this to the {cls.__module__!r} author.')
(8)                     cls._subclass_uses_cla = 'cla' in cls.__dict__ or parent_uses_cla
(8)                     super().__init_subclass__(**kwargs)
(4)                 def __getstate__(self):
(8)                     state = super().__getstate__()
(8)                     state["_shared_axes"] = {
(12)                        name: self._shared_axes[name].get_siblings(self)
(12)                        for name in self._axis_names if self in self._shared_axes[name]}
(8)                     state["_twinned_axes"] = (self._twinned_axes.get_siblings(self)
(34)                                              if self in self._twinned_axes else None)
(8)                     return state
(4)                 def __setstate__(self, state):
(8)                     shared_axes = state.pop("_shared_axes")
(8)                     for name, shared_siblings in shared_axes.items():
(12)                        self._shared_axes[name].join(*shared_siblings)
(8)                     twinned_siblings = state.pop("_twinned_axes")
(8)                     if twinned_siblings:
(12)                        self._twinned_axes.join(*twinned_siblings)
(8)                     self.__dict__ = state
(8)                     self._stale = True
(4)                 def __repr__(self):
(8)                     fields = []
(8)                     if self.get_label():
(12)                        fields += [f"label={self.get_label()!r}"]
(8)                     if hasattr(self, "get_title"):
(12)                        titles = {}
(12)                        for k in ["left", "center", "right"]:
(16)                            title = self.get_title(loc=k)
(16)                            if title:
(20)                                titles[k] = title
(12)                        if titles:
(16)                            fields += [f"title={titles}"]
(8)                     for name, axis in self._axis_map.items():
(12)                        if axis.get_label() and axis.get_label().get_text():
(16)                            fields += [f"{name}label={axis.get_label().get_text()!r}"]
(8)                     return f"<{self.__class__.__name__}: " + ", ".join(fields) + ">"
(4)                 def get_subplotspec(self):
(8)                     """Return the `.SubplotSpec` associated with the subplot, or None."""
(8)                     return self._subplotspec
(4)                 def set_subplotspec(self, subplotspec):
(8)                     """Set the `.SubplotSpec`. associated with the subplot."""
(8)                     self._subplotspec = subplotspec
(8)                     self._set_position(subplotspec.get_position(self.figure))
(4)                 def get_gridspec(self):
(8)                     """Return the `.GridSpec` associated with the subplot, or None."""
(8)                     return self._subplotspec.get_gridspec() if self._subplotspec else None
(4)                 def get_window_extent(self, renderer=None):
(8)                     """
(8)                     Return the Axes bounding box in display space.
(8)                     This bounding box does not include the spines, ticks, ticklabels,
(8)                     or other labels.  For a bounding box including these elements use
(8)                     `~matplotlib.axes.Axes.get_tightbbox`.
(8)                     See Also
(8)                     --------
(8)                     matplotlib.axes.Axes.get_tightbbox
(8)                     matplotlib.axis.Axis.get_tightbbox
(8)                     matplotlib.spines.Spine.get_window_extent
(8)                     """
(8)                     return self.bbox
(4)                 def _init_axis(self):
(8)                     self.xaxis = maxis.XAxis(self, clear=False)
(8)                     self.spines.bottom.register_axis(self.xaxis)
```

```
619: (8)                    self.spines.top.register_axis(self.xaxis)
620: (8)                    self.yaxis = maxis.YAxis(self, clear=False)
621: (8)                    self.spines.left.register_axis(self.yaxis)
622: (8)                    self.spines.right.register_axis(self.yaxis)
623: (4)                def set_figure(self, fig):
624: (8)                    super().set_figure(fig)
625: (8)                    self.bbox = mtransforms.TransformedBbox(self._position,
626: (48)                                                            fig.transSubfigure)
627: (8)                    self.dataLim = mtransforms.Bbox.null()
628: (8)                    self._viewLim = mtransforms.Bbox.unit()
629: (8)                    self.transScale = mtransforms.TransformWrapper(
630: (12)                       mtransforms.IdentityTransform())
631: (8)                    self._set_lim_and_transforms()
632: (4)                def _unstale_viewLim(self):
633: (8)                    need_scale = {
634: (12)                       name: any(ax._stale_viewlims[name]
635: (22)                                 for ax in self._shared_axes[name].get_siblings(self))
636: (12)                       for name in self._axis_names}
637: (8)                    if any(need_scale.values()):
638: (12)                       for name in need_scale:
639: (16)                           for ax in self._shared_axes[name].get_siblings(self):
640: (20)                               ax._stale_viewlims[name] = False
641: (12)                       self.autoscale_view(**{f"scale{name}": scale
642: (35)                                              for name, scale in need_scale.items()})
643: (4)                @property
644: (4)                def viewLim(self):
645: (8)                    self._unstale_viewLim()
646: (8)                    return self._viewLim
647: (4)                def _request_autoscale_view(self, axis="all", tight=None):
648: (8)                    """
649: (8)                    Mark a single axis, or all of them, as stale wrt. autoscaling.
650: (8)                    No computation is performed until the next autoscaling; thus, separate
651: (8)                    calls to control individual axises incur negligible performance cost.
652: (8)                    Parameters
653: (8)                    ----------
654: (8)                    axis : str, default: "all"
655: (12)                       Either an element of ``self._axis_names``, or "all".
656: (8)                    tight : bool or None, default: None
657: (8)                    """
658: (8)                    axis_names = _api.check_getitem(
659: (12)                       {**{k: [k] for k in self._axis_names}, "all": self._axis_names},
660: (12)                       axis=axis)
661: (8)                    for name in axis_names:
662: (12)                       self._stale_viewlims[name] = True
663: (8)                    if tight is not None:
664: (12)                       self._tight = tight
665: (4)                def _set_lim_and_transforms(self):
666: (8)                    """
667: (8)                    Set the *_xaxis_transform*, *_yaxis_transform*, *transScale*,
668: (8)                    *transData*, *transLimits* and *transAxes* transformations.
669: (8)                    .. note::
670: (12)                       This method is primarily used by rectilinear projections of the
671: (12)                       `~matplotlib.axes.Axes` class, and is meant to be overridden by
672: (12)                       new kinds of projection Axes that need different transformations
673: (12)                       and limits. (See `~matplotlib.projections.polar.PolarAxes` for an
674: (12)                       example.)
675: (8)                    """
676: (8)                    self.transAxes = mtransforms.BboxTransformTo(self.bbox)
677: (8)                    self.transScale = mtransforms.TransformWrapper(
678: (12)                       mtransforms.IdentityTransform())
679: (8)                    self.transLimits = mtransforms.BboxTransformFrom(
680: (12)                       mtransforms.TransformedBbox(self._viewLim, self.transScale))
681: (8)                    self.transData = self.transScale + (self.transLimits + self.transAxes)
682: (8)                    self._xaxis_transform = mtransforms.blended_transform_factory(
683: (12)                       self.transData, self.transAxes)
684: (8)                    self._yaxis_transform = mtransforms.blended_transform_factory(
685: (12)                       self.transAxes, self.transData)
686: (4)                def get_xaxis_transform(self, which='grid'):
687: (8)                    """
```

```
(8)                     Get the transformation used for drawing x-axis labels, ticks
(8)                     and gridlines.  The x-direction is in data coordinates and the
(8)                     y-direction is in axis coordinates.
(8)                     .. note::
(12)                        This transformation is primarily used by the
(12)                        `~matplotlib.axis.Axis` class, and is meant to be
(12)                        overridden by new kinds of projections that may need to
(12)                        place axis elements in different locations.
(8)                     Parameters
(8)                     ----------
(8)                     which : {'grid', 'tick1', 'tick2'}
(8)                     """
(8)                     if which == 'grid':
(12)                        return self._xaxis_transform
(8)                     elif which == 'tick1':
(12)                        return self.spines.bottom.get_spine_transform()
(8)                     elif which == 'tick2':
(12)                        return self.spines.top.get_spine_transform()
(8)                     else:
(12)                        raise ValueError(f'unknown value for which: {which!r}')
(4)                 def get_xaxis_text1_transform(self, pad_points):
(8)                     """
(8)                     Returns
(8)                     -------
(8)                     transform : Transform
(12)                        The transform used for drawing x-axis labels, which will add
(12)                        *pad_points* of padding (in points) between the axis and the
label.
(12)                        The x-direction is in data coordinates and the y-direction is in
(12)                        axis coordinates
(8)                     valign : {'center', 'top', 'bottom', 'baseline', 'center_baseline'}
(12)                        The text vertical alignment.
(8)                     halign : {'center', 'left', 'right'}
(12)                        The text horizontal alignment.
(8)                     Notes
(8)                     -----
(8)                     This transformation is primarily used by the `~matplotlib.axis.Axis`
(8)                     class, and is meant to be overridden by new kinds of projections that
(8)                     may need to place axis elements in different locations.
(8)                     """
(8)                     labels_align = mpl.rcParams["xtick.alignment"]
(8)                     return (self.get_xaxis_transform(which='tick1') +
(16)                            mtransforms.ScaledTranslation(0, -1 * pad_points / 72,
(46)                                                          self.figure.dpi_scale_trans),
(16)                            "top", labels_align)
(4)                 def get_xaxis_text2_transform(self, pad_points):
(8)                     """
(8)                     Returns
(8)                     -------
(8)                     transform : Transform
(12)                        The transform used for drawing secondary x-axis labels, which will
(12)                        add *pad_points* of padding (in points) between the axis and the
(12)                        label.  The x-direction is in data coordinates and the y-direction
(12)                        is in axis coordinates
(8)                     valign : {'center', 'top', 'bottom', 'baseline', 'center_baseline'}
(12)                        The text vertical alignment.
(8)                     halign : {'center', 'left', 'right'}
(12)                        The text horizontal alignment.
(8)                     Notes
(8)                     -----
(8)                     This transformation is primarily used by the `~matplotlib.axis.Axis`
(8)                     class, and is meant to be overridden by new kinds of projections that
(8)                     may need to place axis elements in different locations.
(8)                     """
(8)                     labels_align = mpl.rcParams["xtick.alignment"]
(8)                     return (self.get_xaxis_transform(which='tick2') +
(16)                            mtransforms.ScaledTranslation(0, pad_points / 72,
(46)                                                          self.figure.dpi_scale_trans),
(16)                            "bottom", labels_align)
```

```
    def get_yaxis_transform(self, which='grid'):
        """
        Get the transformation used for drawing y-axis labels, ticks
        and gridlines.  The x-direction is in axis coordinates and the
        y-direction is in data coordinates.
        .. note::
            This transformation is primarily used by the
            `~matplotlib.axis.Axis` class, and is meant to be
            overridden by new kinds of projections that may need to
            place axis elements in different locations.
        Parameters
        ----------
        which : {'grid', 'tick1', 'tick2'}
        """
        if which == 'grid':
            return self._yaxis_transform
        elif which == 'tick1':
            return self.spines.left.get_spine_transform()
        elif which == 'tick2':
            return self.spines.right.get_spine_transform()
        else:
            raise ValueError(f'unknown value for which: {which!r}')
    def get_yaxis_text1_transform(self, pad_points):
        """
        Returns
        -------
        transform : Transform
            The transform used for drawing y-axis labels, which will add
            *pad_points* of padding (in points) between the axis and the
label.

            The x-direction is in axis coordinates and the y-direction is in
            data coordinates
        valign : {'center', 'top', 'bottom', 'baseline', 'center_baseline'}
            The text vertical alignment.
        halign : {'center', 'left', 'right'}
            The text horizontal alignment.
        Notes
        -----
        This transformation is primarily used by the `~matplotlib.axis.Axis`
        class, and is meant to be overridden by new kinds of projections that
        may need to place axis elements in different locations.
        """
        labels_align = mpl.rcParams["ytick.alignment"]
        return (self.get_yaxis_transform(which='tick1') +
                mtransforms.ScaledTranslation(-1 * pad_points / 72, 0,
                                              self.figure.dpi_scale_trans),
                labels_align, "right")
    def get_yaxis_text2_transform(self, pad_points):
        """
        Returns
        -------
        transform : Transform
            The transform used for drawing secondart y-axis labels, which will
            add *pad_points* of padding (in points) between the axis and the
            label.  The x-direction is in axis coordinates and the y-direction
            is in data coordinates
        valign : {'center', 'top', 'bottom', 'baseline', 'center_baseline'}
            The text vertical alignment.
        halign : {'center', 'left', 'right'}
            The text horizontal alignment.
        Notes
        -----
        This transformation is primarily used by the `~matplotlib.axis.Axis`
        class, and is meant to be overridden by new kinds of projections that
        may need to place axis elements in different locations.
        """
        labels_align = mpl.rcParams["ytick.alignment"]
        return (self.get_yaxis_transform(which='tick2') +
                mtransforms.ScaledTranslation(pad_points / 72, 0,
```

```
(46)                                                self.figure.dpi_scale_trans),
(16)                            labels_align, "left")
(4)                 def _update_transScale(self):
(8)                     self.transScale.set(
(12)                        mtransforms.blended_transform_factory(
(16)                            self.xaxis.get_transform(), self.yaxis.get_transform()))
(4)                 def get_position(self, original=False):
(8)                     """
(8)                     Return the position of the Axes within the figure as a `.Bbox`.
(8)                     Parameters
(8)                     ----------
(8)                     original : bool
(12)                        If ``True``, return the original position. Otherwise, return the
(12)                        active position. For an explanation of the positions see
(12)                        `.set_position`.
(8)                     Returns
(8)                     -------
(8)                     `.Bbox`
(8)                     """
(8)                     if original:
(12)                        return self._originalPosition.frozen()
(8)                     else:
(12)                        locator = self.get_axes_locator()
(12)                        if not locator:
(16)                            self.apply_aspect()
(12)                        return self._position.frozen()
(4)                 def set_position(self, pos, which='both'):
(8)                     """
(8)                     Set the Axes position.
(8)                     Axes have two position attributes. The 'original' position is the
(8)                     position allocated for the Axes. The 'active' position is the
(8)                     position the Axes is actually drawn at. These positions are usually
(8)                     the same unless a fixed aspect is set to the Axes. See
(8)                     `.Axes.set_aspect` for details.
(8)                     Parameters
(8)                     ----------
(8)                     pos : [left, bottom, width, height] or `~matplotlib.transforms.Bbox`
(12)                        The new position of the Axes in `.Figure` coordinates.
(8)                     which : {'both', 'active', 'original'}, default: 'both'
(12)                        Determines which position variables to change.
(8)                     See Also
(8)                     --------
(8)                     matplotlib.transforms.Bbox.from_bounds
(8)                     matplotlib.transforms.Bbox.from_extents
(8)                     """
(8)                     self._set_position(pos, which=which)
(8)                     self.set_in_layout(False)
(4)                 def _set_position(self, pos, which='both'):
(8)                     """
(8)                     Private version of set_position.
(8)                     Call this internally to get the same functionality of `set_position`,
(8)                     but not to take the axis out of the constrained_layout hierarchy.
(8)                     """
(8)                     if not isinstance(pos, mtransforms.BboxBase):
(12)                        pos = mtransforms.Bbox.from_bounds(*pos)
(8)                     for ax in self._twinned_axes.get_siblings(self):
(12)                        if which in ('both', 'active'):
(16)                            ax._position.set(pos)
(12)                        if which in ('both', 'original'):
(16)                            ax._originalPosition.set(pos)
(8)                     self.stale = True
(4)                 def reset_position(self):
(8)                     """
(8)                     Reset the active position to the original position.
(8)                     This undoes changes to the active position (as defined in
(8)                     `.set_position`) which may have been performed to satisfy fixed-aspect
(8)                     constraints.
(8)                     """
(8)                     for ax in self._twinned_axes.get_siblings(self):
```

```
(12)                        pos = ax.get_position(original=True)
(12)                        ax.set_position(pos, which='active')
(4)                 def set_axes_locator(self, locator):
(8)                     """
(8)                     Set the Axes locator.
(8)                     Parameters
(8)                     ----------
(8)                     locator : Callable[[Axes, Renderer], Bbox]
(8)                     """
(8)                     self._axes_locator = locator
(8)                     self.stale = True
(4)                 def get_axes_locator(self):
(8)                     """
(8)                     Return the axes_locator.
(8)                     """
(8)                     return self._axes_locator
(4)                 def _set_artist_props(self, a):
(8)                     """Set the boilerplate props for artists added to Axes."""
(8)                     a.set_figure(self.figure)
(8)                     if not a.is_transform_set():
(12)                        a.set_transform(self.transData)
(8)                     a.axes = self
(8)                     if a.get_mouseover():
(12)                        self._mouseover_set.add(a)
(4)                 def _gen_axes_patch(self):
(8)                     """
(8)                     Returns
(8)                     -------
(8)                     Patch
(12)                        The patch used to draw the background of the Axes.  It is also
used
(12)                        as the clipping path for any data elements on the Axes.
(12)                        In the standard Axes, this is a rectangle, but in other
projections
(12)                        it may not be.
(8)                     Notes
(8)                     -----
(8)                     Intended to be overridden by new projection types.
(8)                     """
(8)                     return mpatches.Rectangle((0.0, 0.0), 1.0, 1.0)
(4)                 def _gen_axes_spines(self, locations=None, offset=0.0, units='inches'):
(8)                     """
(8)                     Returns
(8)                     -------
(8)                     dict
(12)                        Mapping of spine names to `.Line2D` or `.Patch` instances that are
(12)                        used to draw Axes spines.
(12)                        In the standard Axes, spines are single line segments, but in
other
(12)                        projections they may not be.
(8)                     Notes
(8)                     -----
(8)                     Intended to be overridden by new projection types.
(8)                     """
(8)                     return {side: mspines.Spine.linear_spine(self, side)
(16)                            for side in ['left', 'right', 'bottom', 'top']}
(4)                 def sharex(self, other):
(8)                     """
(8)                     Share the x-axis with *other*.
(8)                     This is equivalent to passing ``sharex=other`` when constructing the
(8)                     Axes, and cannot be used if the x-axis is already being shared with
(8)                     another Axes.  Note that it is not possible to unshare axes.
(8)                     """
(8)                     _api.check_isinstance(_AxesBase, other=other)
(8)                     if self._sharex is not None and other is not self._sharex:
(12)                        raise ValueError("x-axis is already shared")
(8)                     self._shared_axes["x"].join(self, other)
(8)                     self._sharex = other
(8)                     self.xaxis.major = other.xaxis.major  # Ticker instances holding
```

```
(8)                 self.xaxis.minor = other.xaxis.minor  # locator and formatter.
(8)                 x0, x1 = other.get_xlim()
(8)                 self.set_xlim(x0, x1, emit=False, auto=other.get_autoscalex_on())
(8)                 self.xaxis._scale = other.xaxis._scale
(4)             def sharey(self, other):
(8)                 """
(8)                 Share the y-axis with *other*.
(8)                 This is equivalent to passing ``sharey=other`` when constructing the
(8)                 Axes, and cannot be used if the y-axis is already being shared with
(8)                 another Axes.  Note that it is not possible to unshare axes.
(8)                 """
(8)                 _api.check_isinstance(_AxesBase, other=other)
(8)                 if self._sharey is not None and other is not self._sharey:
(12)                    raise ValueError("y-axis is already shared")
(8)                 self._shared_axes["y"].join(self, other)
(8)                 self._sharey = other
(8)                 self.yaxis.major = other.yaxis.major  # Ticker instances holding
(8)                 self.yaxis.minor = other.yaxis.minor  # locator and formatter.
(8)                 y0, y1 = other.get_ylim()
(8)                 self.set_ylim(y0, y1, emit=False, auto=other.get_autoscaley_on())
(8)                 self.yaxis._scale = other.yaxis._scale
(4)             def __clear(self):
(8)                 """Clear the Axes."""
(8)                 if hasattr(self, 'patch'):
(12)                    patch_visible = self.patch.get_visible()
(8)                 else:
(12)                    patch_visible = True
(8)                 xaxis_visible = self.xaxis.get_visible()
(8)                 yaxis_visible = self.yaxis.get_visible()
(8)                 for axis in self._axis_map.values():
(12)                    axis.clear()  # Also resets the scale to linear.
(8)                 for spine in self.spines.values():
(12)                    spine._clear()  # Use _clear to not clear Axis again
(8)                 self.ignore_existing_data_limits = True
(8)                 self.callbacks = cbook.CallbackRegistry(
(12)                    signals=["xlim_changed", "ylim_changed", "zlim_changed"])
(8)                 if mpl.rcParams['xtick.minor.visible']:
(12)                    self.xaxis.set_minor_locator(mticker.AutoMinorLocator())
(8)                 if mpl.rcParams['ytick.minor.visible']:
(12)                    self.yaxis.set_minor_locator(mticker.AutoMinorLocator())
(8)                 self._xmargin = mpl.rcParams['axes.xmargin']
(8)                 self._ymargin = mpl.rcParams['axes.ymargin']
(8)                 self._tight = None
(8)                 self._use_sticky_edges = True
(8)                 self._get_lines = _process_plot_var_args()
(8)                 self._get_patches_for_fill = _process_plot_var_args('fill')
(8)                 self._gridOn = mpl.rcParams['axes.grid']
(8)                 old_children, self._children = self._children, []
(8)                 for chld in old_children:
(12)                    chld.axes = chld.figure = None
(8)                 self._mouseover_set = _OrderedSet()
(8)                 self.child_axes = []
(8)                 self._current_image = None  # strictly for pyplot via _sci, _gci
(8)                 self._projection_init = None  # strictly for pyplot.subplot
(8)                 self.legend_ = None
(8)                 self.containers = []
(8)                 self.grid(False)  # Disable grid on init to use rcParameter
(8)                 self.grid(self._gridOn, which=mpl.rcParams['axes.grid.which'],
(18)                          axis=mpl.rcParams['axes.grid.axis'])
(8)                 props = font_manager.FontProperties(
(12)                    size=mpl.rcParams['axes.titlesize'],
(12)                    weight=mpl.rcParams['axes.titleweight'])
(8)                 y = mpl.rcParams['axes.titley']
(8)                 if y is None:
(12)                    y = 1.0
(12)                    self._autotitlepos = True
(8)                 else:
(12)                    self._autotitlepos = False
(8)                 self.title = mtext.Text(
```

```
(12)                        x=0.5, y=y, text='',
(12)                        fontproperties=props,
(12)                        verticalalignment='baseline',
(12)                        horizontalalignment='center',
(12)                        )
(8)                     self._left_title = mtext.Text(
(12)                        x=0.0, y=y, text='',
(12)                        fontproperties=props.copy(),
(12)                        verticalalignment='baseline',
(12)                        horizontalalignment='left', )
(8)                     self._right_title = mtext.Text(
(12)                        x=1.0, y=y, text='',
(12)                        fontproperties=props.copy(),
(12)                        verticalalignment='baseline',
(12)                        horizontalalignment='right',
(12)                        )
(8)                     title_offset_points = mpl.rcParams['axes.titlepad']
(8)                     self._set_title_offset_trans(title_offset_points)
(8)                     for _title in (self.title, self._left_title, self._right_title):
(12)                        self._set_artist_props(_title)
(8)                     self.patch = self._gen_axes_patch()
(8)                     self.patch.set_figure(self.figure)
(8)                     self.patch.set_facecolor(self._facecolor)
(8)                     self.patch.set_edgecolor('none')
(8)                     self.patch.set_linewidth(0)
(8)                     self.patch.set_transform(self.transAxes)
(8)                     self.set_axis_on()
(8)                     self.xaxis.set_clip_path(self.patch)
(8)                     self.yaxis.set_clip_path(self.patch)
(8)                     if self._sharex is not None:
(12)                        self.xaxis.set_visible(xaxis_visible)
(12)                        self.patch.set_visible(patch_visible)
(8)                     if self._sharey is not None:
(12)                        self.yaxis.set_visible(yaxis_visible)
(12)                        self.patch.set_visible(patch_visible)
(8)                     for name, axis in self._axis_map.items():
(12)                        share = getattr(self, f"_share{name}")
(12)                        if share is not None:
(16)                            getattr(self, f"share{name}")(share)
(12)                        else:
(16)                            if self.name == "polar":
(20)                                axis._set_scale("linear")
(16)                            axis._set_lim(0, 1, auto=True)
(8)                     self._update_transScale()
(8)                     self.stale = True
(4)                 def clear(self):
(8)                     """Clear the Axes."""
(8)                     if self._subclass_uses_cla:
(12)                        self.cla()
(8)                     else:
(12)                        self.__clear()
(4)                 def cla(self):
(8)                     """Clear the Axes."""
(8)                     if self._subclass_uses_cla:
(12)                        self.__clear()
(8)                     else:
(12)                        self.clear()
(4)                 class ArtistList(Sequence):
(8)                     """
(8)                     A sublist of Axes children based on their type.
(8)                     The type-specific children sublists were made immutable in Matplotlib
(8)                     3.7.  In the future these artist lists may be replaced by tuples. Use
(8)                     as if this is a tuple already.
(8)                     """
(8)                     def __init__(self, axes, prop_name,
(21)                                 valid_types=None, invalid_types=None):
(12)                        """
(12)                        Parameters
(12)                        ----------
```

```
(12)                        axes : `~matplotlib.axes.Axes`
(16)                            The Axes from which this sublist will pull the children
(16)                            Artists.
(12)                        prop_name : str
(16)                            The property name used to access this sublist from the Axes;
(16)                            used to generate deprecation warnings.
(12)                        valid_types : list of type, optional
(16)                            A list of types that determine which children will be returned
(16)                            by this sublist. If specified, then the Artists in the sublist
(16)                            must be instances of any of these types. If unspecified, then
(16)                            any type of Artist is valid (unless limited by
(16)                            *invalid_types*.)
(12)                        invalid_types : tuple, optional
(16)                            A list of types that determine which children will *not* be
(16)                            returned by this sublist. If specified, then Artists in the
(16)                            sublist will never be an instance of these types. Otherwise,
no
(16)                            types will be excluded.
(12)                        """
(12)                        self._axes = axes
(12)                        self._prop_name = prop_name
(12)                        self._type_check = lambda artist: (
(16)                            (not valid_types or isinstance(artist, valid_types)) and
(16)                            (not invalid_types or not isinstance(artist, invalid_types))
(12)                        )
(8)                     def __repr__(self):
(12)                        return f'<Axes.ArtistList of {len(self)} {self._prop_name}>'
(8)                     def __len__(self):
(12)                        return sum(self._type_check(artist)
(23)                                   for artist in self._axes._children)
(8)                     def __iter__(self):
(12)                        for artist in list(self._axes._children):
(16)                            if self._type_check(artist):
(20)                                yield artist
(8)                     def __getitem__(self, key):
(12)                        return [artist
(20)                                for artist in self._axes._children
(20)                                if self._type_check(artist)][key]
(8)                     def __add__(self, other):
(12)                        if isinstance(other, (list, _AxesBase.ArtistList)):
(16)                            return [*self, *other]
(12)                        if isinstance(other, (tuple, _AxesBase.ArtistList)):
(16)                            return (*self, *other)
(12)                        return NotImplemented
(8)                     def __radd__(self, other):
(12)                        if isinstance(other, list):
(16)                            return other + list(self)
(12)                        if isinstance(other, tuple):
(16)                            return other + tuple(self)
(12)                        return NotImplemented
(4)                 @property
(4)                 def artists(self):
(8)                     return self.ArtistList(self, 'artists', invalid_types=(
(12)                        mcoll.Collection, mimage.AxesImage, mlines.Line2D, mpatches.Patch,
(12)                        mtable.Table, mtext.Text))
(4)                 @property
(4)                 def collections(self):
(8)                     return self.ArtistList(self, 'collections',
(31)                                           valid_types=mcoll.Collection)
(4)                 @property
(4)                 def images(self):
(8)                     return self.ArtistList(self, 'images', valid_types=mimage.AxesImage)
(4)                 @property
(4)                 def lines(self):
(8)                     return self.ArtistList(self, 'lines', valid_types=mlines.Line2D)
(4)                 @property
(4)                 def patches(self):
(8)                     return self.ArtistList(self, 'patches', valid_types=mpatches.Patch)
(4)                 @property
```

```
    def tables(self):
        return self.ArtistList(self, 'tables', valid_types=mtable.Table)
    @property
    def texts(self):
        return self.ArtistList(self, 'texts', valid_types=mtext.Text)
    def get_facecolor(self):
        """Get the facecolor of the Axes."""
        return self.patch.get_facecolor()
    def set_facecolor(self, color):
        """
        Set the facecolor of the Axes.
        Parameters
        ----------
        color : :mpltype:`color`
        """
        self._facecolor = color
        self.stale = True
        return self.patch.set_facecolor(color)
    def _set_title_offset_trans(self, title_offset_points):
        """
        Set the offset for the title either from :rc:`axes.titlepad`
        or from set_title kwarg ``pad``.
        """
        self.titleOffsetTrans = mtransforms.ScaledTranslation(
                0.0, title_offset_points / 72,
                self.figure.dpi_scale_trans)
        for _title in (self.title, self._left_title, self._right_title):
            _title.set_transform(self.transAxes + self.titleOffsetTrans)
            _title.set_clip_box(None)
    def set_prop_cycle(self, *args, **kwargs):
        """
        Set the property cycle of the Axes.
        The property cycle controls the style properties such as color,
        marker and linestyle of future plot commands. The style properties
        of data already added to the Axes are not modified.
        Call signatures::
          set_prop_cycle(cycler)
          set_prop_cycle(label=values[, label2=values2[, ...]])
          set_prop_cycle(label, values)
        Form 1 sets given `~cycler.Cycler` object.
        Form 2 creates a `~cycler.Cycler` which cycles over one or more
        properties simultaneously and set it as the property cycle of the
        Axes. If multiple properties are given, their value lists must have
        the same length. This is just a shortcut for explicitly creating a
        cycler and passing it to the function, i.e. it's short for
        ``set_prop_cycle(cycler(label=values, label2=values2, ...))``.
        Form 3 creates a `~cycler.Cycler` for a single property and set it
        as the property cycle of the Axes. This form exists for compatibility
        with the original `cycler.cycler` interface. Its use is discouraged
        in favor of the kwarg form, i.e. ``set_prop_cycle(label=values)``.
        Parameters
        ----------
        cycler : `~cycler.Cycler`
            Set the given Cycler. *None* resets to the cycle defined by the
            current style.
            .. ACCEPTS: `~cycler.Cycler`
        label : str
            The property key. Must be a valid `.Artist` property.
            For example, 'color' or 'linestyle'. Aliases are allowed,
            such as 'c' for 'color' and 'lw' for 'linewidth'.
        values : iterable
            Finite-length iterable of the property values. These values
            are validated and will raise a ValueError if invalid.
        See Also
        --------
        matplotlib.rcsetup.cycler
            Convenience function for creating validated cyclers for
properties.
        cycler.cycler
```

```
(12)                        The original function for creating unvalidated cyclers.
(8)                     Examples
(8)                     --------
(8)                     Setting the property cycle for a single property:
(8)                     >>> ax.set_prop_cycle(color=['red', 'green', 'blue'])
(8)                     Setting the property cycle for simultaneously cycling over multiple
(8)                     properties (e.g. red circle, green plus, blue cross):
(8)                     >>> ax.set_prop_cycle(color=['red', 'green', 'blue'],
(8)                     ...                    marker=['o', '+', 'x'])
(8)                     """
(8)                     if args and kwargs:
(12)                        raise TypeError("Cannot supply both positional and keyword "
(28)                                        "arguments to this method.")
(8)                     if len(args) == 1 and args[0] is None:
(12)                        prop_cycle = None
(8)                     else:
(12)                        prop_cycle = cycler(*args, **kwargs)
(8)                     self._get_lines.set_prop_cycle(prop_cycle)
(8)                     self._get_patches_for_fill.set_prop_cycle(prop_cycle)
(4)                 def get_aspect(self):
(8)                     """
(8)                     Return the aspect ratio of the Axes scaling.
(8)                     This is either "auto" or a float giving the ratio of y/x-scale.
(8)                     """
(8)                     return self._aspect
(4)                 def set_aspect(self, aspect, adjustable=None, anchor=None, share=False):
(8)                     """
(8)                     Set the aspect ratio of the Axes scaling, i.e. y/x-scale.
(8)                     Parameters
(8)                     ----------
(8)                     aspect : {'auto', 'equal'} or float
(12)                        Possible values:
(12)                        - 'auto': fill the position rectangle with data.
(12)                        - 'equal': same as ``aspect=1``, i.e. same scaling for x and y.
(12)                        - *float*: The displayed size of 1 unit in y-data coordinates will
(14)                          be *aspect* times the displayed size of 1 unit in x-data
(14)                          coordinates; e.g. for ``aspect=2`` a square in data coordinates
(14)                          will be rendered with a height of twice its width.
(8)                     adjustable : None or {'box', 'datalim'}, optional
(12)                        If not ``None``, this defines which parameter will be adjusted to
(12)                        meet the required aspect. See `.set_adjustable` for further
(12)                        details.
(8)                     anchor : None or str or (float, float), optional
(12)                        If not ``None``, this defines where the Axes will be drawn if
there
(12)                        is extra space due to aspect constraints. The most common way
(12)                        to specify the anchor are abbreviations of cardinal directions:
(12)                        =====   =====================
(12)                        value   description
(12)                        =====   =====================
(12)                        'C'     centered
(12)                        'SW'    lower left corner
(12)                        'S'     middle of bottom edge
(12)                        'SE'    lower right corner
(12)                        etc.
(12)                        =====   =====================
(12)                        See `~.Axes.set_anchor` for further details.
(8)                     share : bool, default: False
(12)                        If ``True``, apply the settings to all shared Axes.
(8)                     See Also
(8)                     --------
(8)                     matplotlib.axes.Axes.set_adjustable
(12)                        Set how the Axes adjusts to achieve the required aspect ratio.
(8)                     matplotlib.axes.Axes.set_anchor
(12)                        Set the position in case of extra space.
(8)                     """
(8)                     if cbook._str_equal(aspect, 'equal'):
(12)                        aspect = 1
(8)                     if not cbook._str_equal(aspect, 'auto'):
```

```
(12)                        aspect = float(aspect)  # raise ValueError if necessary
(12)                        if aspect <= 0 or not np.isfinite(aspect):
(16)                            raise ValueError("aspect must be finite and positive ")
(8)                     if share:
(12)                        axes = {sibling for name in self._axis_names
(20)                                for sibling in self._shared_axes[name].get_siblings(self)}
(8)                     else:
(12)                        axes = [self]
(8)                     for ax in axes:
(12)                        ax._aspect = aspect
(8)                     if adjustable is None:
(12)                        adjustable = self._adjustable
(8)                     self.set_adjustable(adjustable, share=share)  # Handle sharing.
(8)                     if anchor is not None:
(12)                        self.set_anchor(anchor, share=share)
(8)                     self.stale = True
(4)                 def get_adjustable(self):
(8)                     """
(8)                     Return whether the Axes will adjust its physical dimension ('box') or
(8)                     its data limits ('datalim') to achieve the desired aspect ratio.
(8)                     See Also
(8)                     --------
(8)                     matplotlib.axes.Axes.set_adjustable
(12)                        Set how the Axes adjusts to achieve the required aspect ratio.
(8)                     matplotlib.axes.Axes.set_aspect
(12)                        For a description of aspect handling.
(8)                     """
(8)                     return self._adjustable
(4)                 def set_adjustable(self, adjustable, share=False):
(8)                     """
(8)                     Set how the Axes adjusts to achieve the required aspect ratio.
(8)                     Parameters
(8)                     ----------
(8)                     adjustable : {'box', 'datalim'}
(12)                        If 'box', change the physical dimensions of the Axes.
(12)                        If 'datalim', change the ``x`` or ``y`` data limits.
(8)                     share : bool, default: False
(12)                        If ``True``, apply the settings to all shared Axes.
(8)                     See Also
(8)                     --------
(8)                     matplotlib.axes.Axes.set_aspect
(12)                        For a description of aspect handling.
(8)                     Notes
(8)                     -----
(8)                     Shared Axes (of which twinned Axes are a special case)
(8)                     impose restrictions on how aspect ratios can be imposed.
(8)                     For twinned Axes, use 'datalim'.  For Axes that share both
(8)                     x and y, use 'box'.  Otherwise, either 'datalim' or 'box'
(8)                     may be used.  These limitations are partly a requirement
(8)                     to avoid over-specification, and partly a result of the
(8)                     particular implementation we are currently using, in
(8)                     which the adjustments for aspect ratios are done sequentially
(8)                     and independently on each Axes as it is drawn.
(8)                     """
(8)                     _api.check_in_list(["box", "datalim"], adjustable=adjustable)
(8)                     if share:
(12)                        axs = {sibling for name in self._axis_names
(19)                               for sibling in self._shared_axes[name].get_siblings(self)}
(8)                     else:
(12)                        axs = [self]
(8)                     if (adjustable == "datalim"
(16)                            and any(getattr(ax.get_data_ratio, "__func__", None)
(24)                                    != _AxesBase.get_data_ratio
(24)                                    for ax in axs)):
(12)                        raise ValueError("Cannot set Axes adjustable to 'datalim' for "
(29)                                         "Axes which override 'get_data_ratio'")
(8)                     for ax in axs:
(12)                        ax._adjustable = adjustable
(8)                     self.stale = True
```

```
    def get_box_aspect(self):
        """
        Return the Axes box aspect, i.e. the ratio of height to width.
        The box aspect is ``None`` (i.e. chosen depending on the available
        figure space) unless explicitly specified.
        See Also
        --------
        matplotlib.axes.Axes.set_box_aspect
            for a description of box aspect.
        matplotlib.axes.Axes.set_aspect
            for a description of aspect handling.
        """
        return self._box_aspect
    def set_box_aspect(self, aspect=None):
        """
        Set the Axes box aspect, i.e. the ratio of height to width.
        This defines the aspect of the Axes in figure space and is not to be
        confused with the data aspect (see `~.Axes.set_aspect`).
        Parameters
        ----------
        aspect : float or None
            Changes the physical dimensions of the Axes, such that the ratio
            of the Axes height to the Axes width in physical units is equal to
            *aspect*. Defining a box aspect will change the *adjustable*
            property to 'datalim' (see `~.Axes.set_adjustable`).
            *None* will disable a fixed box aspect so that height and width
            of the Axes are chosen independently.
        See Also
        --------
        matplotlib.axes.Axes.set_aspect
            for a description of aspect handling.
        """
        axs = {*self._twinned_axes.get_siblings(self),
               *self._twinned_axes.get_siblings(self)}
        if aspect is not None:
            aspect = float(aspect)
            for ax in axs:
                ax.set_adjustable("datalim")
        for ax in axs:
            ax._box_aspect = aspect
            ax.stale = True
    def get_anchor(self):
        """
        Get the anchor location.
        See Also
        --------
        matplotlib.axes.Axes.set_anchor
            for a description of the anchor.
        matplotlib.axes.Axes.set_aspect
            for a description of aspect handling.
        """
        return self._anchor
    def set_anchor(self, anchor, share=False):
        """
        Define the anchor location.
        The actual drawing area (active position) of the Axes may be smaller
        than the Bbox (original position) when a fixed aspect is required. The
        anchor defines where the drawing area will be located within the
        available space.
        Parameters
        ----------
        anchor : (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...}
            Either an (*x*, *y*) pair of relative coordinates (0 is left or
            bottom, 1 is right or top), 'C' (center), or a cardinal direction
            ('SW', southwest, is bottom left, etc.).  str inputs are
shorthands
            for (*x*, *y*) coordinates, as shown in the following diagram::
                ┌────────────────┬────────────────┬────────────────┐
                │ 'NW' (0.0, 1.0) │ 'N' (0.5, 1.0) │ 'NE' (1.0, 1.0) │
```

```
                        |                    |                    |
                        | 'W'  (0.0, 0.5)    | 'C' (0.5, 0.5)     | 'E'  (1.0, 0.5)    |
                        |                    |                    |
                        | 'SW' (0.0, 0.0)    | 'S' (0.5, 0.0)     | 'SE' (1.0, 0.0)    |
                        |                    |                    |

                share : bool, default: False
                    If ``True``, apply the settings to all shared Axes.
                See Also
                --------
                matplotlib.axes.Axes.set_aspect
                    for a description of aspect handling.
                """
                if not (anchor in mtransforms.Bbox.coefs or len(anchor) == 2):
                    raise ValueError('argument must be among %s' %
                                     ', '.join(mtransforms.Bbox.coefs))
                if share:
                    axes = {sibling for name in self._axis_names
                            for sibling in self._shared_axes[name].get_siblings(self)}
                else:
                    axes = [self]
                for ax in axes:
                    ax._anchor = anchor
                self.stale = True
            def get_data_ratio(self):
                """
                Return the aspect ratio of the scaled data.
                Notes
                -----
                This method is intended to be overridden by new projection types.
                """
                txmin, txmax = self.xaxis.get_transform().transform(self.get_xbound())
                tymin, tymax = self.yaxis.get_transform().transform(self.get_ybound())
                xsize = max(abs(txmax - txmin), 1e-30)
                ysize = max(abs(tymax - tymin), 1e-30)
                return ysize / xsize
            def apply_aspect(self, position=None):
                """
                Adjust the Axes for a specified data aspect ratio.
                Depending on `.get_adjustable` this will modify either the
                Axes box (position) or the view limits. In the former case,
                `~matplotlib.axes.Axes.get_anchor` will affect the position.
                Parameters
                ----------
                position : None or .Bbox
                    .. note::
                        This parameter exists for historic reasons and is considered
                        internal. End users should not use it.
                    If not ``None``, this defines the position of the
                    Axes within the figure as a Bbox. See `~.Axes.get_position`
                    for further details.
                Notes
                -----
                This is called automatically when each Axes is drawn.  You may need
                to call it yourself if you need to update the Axes position and/or
                view limits before the Figure is drawn.
                An alternative with a broader scope is `.Figure.draw_without_rendering`,
                which updates all stale components of a figure, not only the positioning /
                view limits of a single Axes.
                See Also
                --------
                matplotlib.axes.Axes.set_aspect
                    For a description of aspect ratio handling.
                matplotlib.axes.Axes.set_adjustable
                    Set how the Axes adjusts to achieve the required aspect ratio.
                matplotlib.axes.Axes.set_anchor
                    Set the position in case of extra space.
                matplotlib.figure.Figure.draw_without_rendering
```

```
(12)                    Update all stale components of a figure.
(8)                 Examples
(8)                 --------
(8)                 A typical usage example would be the following. `~.Axes.imshow` sets the
(8)                 aspect to 1, but adapting the Axes position and extent to reflect this is
(8)                 deferred until rendering for performance reasons. If you want to know the
(8)                 Axes size before, you need to call `.apply_aspect` to get the correct
(8)                 values.
(8)                 >>> fig, ax = plt.subplots()
(8)                 >>> ax.imshow(np.zeros((3, 3)))
(8)                 >>> ax.bbox.width, ax.bbox.height
(8)                 (496.0, 369.59999999999997)
(8)                 >>> ax.apply_aspect()
(8)                 >>> ax.bbox.width, ax.bbox.height
(8)                 (369.59999999999997, 369.59999999999997)
(8)                 """
(8)                 if position is None:
(12)                    position = self.get_position(original=True)
(8)                 aspect = self.get_aspect()
(8)                 if aspect == 'auto' and self._box_aspect is None:
(12)                    self._set_position(position, which='active')
(12)                    return
(8)                 trans = self.get_figure().transSubfigure
(8)                 bb = mtransforms.Bbox.unit().transformed(trans)
(8)                 fig_aspect = bb.height / bb.width
(8)                 if self._adjustable == 'box':
(12)                    if self in self._twinned_axes:
(16)                        raise RuntimeError("Adjustable 'box' is not allowed in a "
(35)                                           "twinned Axes; use 'datalim' instead")
(12)                    box_aspect = aspect * self.get_data_ratio()
(12)                    pb = position.frozen()
(12)                    pb1 = pb.shrunk_to_aspect(box_aspect, pb, fig_aspect)
(12)                    self._set_position(pb1.anchored(self.get_anchor(), pb), 'active')
(12)                    return
(8)                 if self._box_aspect is not None:
(12)                    pb = position.frozen()
(12)                    pb1 = pb.shrunk_to_aspect(self._box_aspect, pb, fig_aspect)
(12)                    self._set_position(pb1.anchored(self.get_anchor(), pb), 'active')
(12)                    if aspect == "auto":
(16)                        return
(8)                 if self._box_aspect is None:
(12)                    self._set_position(position, which='active')
(8)                 else:
(12)                    position = pb1.anchored(self.get_anchor(), pb)
(8)                 x_trf = self.xaxis.get_transform()
(8)                 y_trf = self.yaxis.get_transform()
(8)                 xmin, xmax = x_trf.transform(self.get_xbound())
(8)                 ymin, ymax = y_trf.transform(self.get_ybound())
(8)                 xsize = max(abs(xmax - xmin), 1e-30)
(8)                 ysize = max(abs(ymax - ymin), 1e-30)
(8)                 box_aspect = fig_aspect * (position.height / position.width)
(8)                 data_ratio = box_aspect / aspect
(8)                 y_expander = data_ratio * xsize / ysize - 1
(8)                 if abs(y_expander) < 0.005:
(12)                    return
(8)                 dL = self.dataLim
(8)                 x0, x1 = x_trf.transform(dL.intervalx)
(8)                 y0, y1 = y_trf.transform(dL.intervaly)
(8)                 xr = 1.05 * (x1 - x0)
(8)                 yr = 1.05 * (y1 - y0)
(8)                 xmarg = xsize - xr
(8)                 ymarg = ysize - yr
(8)                 Ysize = data_ratio * xsize
(8)                 Xsize = ysize / data_ratio
(8)                 Xmarg = Xsize - xr
(8)                 Ymarg = Ysize - yr
```

```
1571: (8)                   xm = 0
1572: (8)                   ym = 0
1573: (8)                   shared_x = self in self._shared_axes["x"]
1574: (8)                   shared_y = self in self._shared_axes["y"]
1575: (8)                   if shared_x and shared_y:
1576: (12)                      raise RuntimeError("set_aspect(..., adjustable='datalim') or "
1577: (31)                                         "axis('equal') are not allowed when both axes "
1578: (31)                                         "are shared.  Try set_aspect(..., "
1579: (31)                                         "adjustable='box').")
1580: (8)                   if shared_y:
1581: (12)                      adjust_y = False
1582: (8)                   else:
1583: (12)                      if xmarg > xm and ymarg > ym:
1584: (16)                          adjy = ((Ymarg > 0 and y_expander < 0) or
1585: (24)                                  (Xmarg < 0 and y_expander > 0))
1586: (12)                      else:
1587: (16)                          adjy = y_expander > 0
1588: (12)                      adjust_y = shared_x or adjy  # (Ymarg > xmarg)
1589: (8)                   if adjust_y:
1590: (12)                      yc = 0.5 * (ymin + ymax)
1591: (12)                      y0 = yc - Ysize / 2.0
1592: (12)                      y1 = yc + Ysize / 2.0
1593: (12)                      self.set_ybound(y_trf.inverted().transform([y0, y1]))
1594: (8)                   else:
1595: (12)                      xc = 0.5 * (xmin + xmax)
1596: (12)                      x0 = xc - Xsize / 2.0
1597: (12)                      x1 = xc + Xsize / 2.0
1598: (12)                      self.set_xbound(x_trf.inverted().transform([x0, x1]))
1599: (4)               def axis(self, arg=None, /, *, emit=True, **kwargs):
1600: (8)                   """
1601: (8)                   Convenience method to get or set some axis properties.
1602: (8)                   Call signatures::
1603: (10)                    xmin, xmax, ymin, ymax = axis()
1604: (10)                    xmin, xmax, ymin, ymax = axis([xmin, xmax, ymin, ymax])
1605: (10)                    xmin, xmax, ymin, ymax = axis(option)
1606: (10)                    xmin, xmax, ymin, ymax = axis(**kwargs)
1607: (8)                   Parameters
1608: (8)                   ----------
1609: (8)                   xmin, xmax, ymin, ymax : float, optional
1610: (12)                      The axis limits to be set.  This can also be achieved using ::
1611: (16)                          ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))
1612: (8)                   option : bool or str
1613: (12)                      If a bool, turns axis lines and labels on or off. If a string,
1614: (12)                      possible values are:
1615: (12)                      ================
============================================================
1616: (12)                      Value            Description
1617: (12)                      ================
============================================================
1618: (12)                      'off' or `False` Hide all axis decorations, i.e. axis labels,
spines,
1619: (29)                                       tick marks, tick labels, and grid lines.
1620: (29)                                       This is the same as `~.Axes.set_axis_off()`.
1621: (12)                      'on' or `True`   Do not hide all axis decorations, i.e. axis
labels, spines,
1622: (29)                                       tick marks, tick labels, and grid lines.
1623: (29)                                       This is the same as `~.Axes.set_axis_on()`.
1624: (12)                      'equal'          Set equal scaling (i.e., make circles circular)
by
1625: (29)                                       changing the axis limits. This is the same as
1626: (29)                                       ``ax.set_aspect('equal', adjustable='datalim')``.
1627: (29)                                       Explicit data limits may not be respected in this
case.
1628: (12)                      'scaled'         Set equal scaling (i.e., make circles circular)
by
1629: (29)                                       changing dimensions of the plot box. This is the
same as
1630: (29)                                       ``ax.set_aspect('equal', adjustable='box',
anchor='C')``.
```

```
(29)                                        Additionally, further autoscaling will be
disabled.
(12)                        'tight'         Set limits just large enough to show all data,
then
(29)                                        disable further autoscaling.
(12)                        'auto'          Automatic scaling (fill plot box with data).
(12)                        'image'         'scaled' with axis limits equal to data limits.
(12)                        'square'        Square plot; similar to 'scaled', but initially
forcing
(29)                                        ``xmax-xmin == ymax-ymin``.
(12)                        ================
==========================================================
(8)                     emit : bool, default: True
(12)                        Whether observers are notified of the axis limit change.
(12)                        This option is passed on to `~.Axes.set_xlim` and
(12)                        `~.Axes.set_ylim`.
(8)                     Returns
(8)                     -------
(8)                     xmin, xmax, ymin, ymax : float
(12)                        The axis limits.
(8)                     See Also
(8)                     --------
(8)                     matplotlib.axes.Axes.set_xlim
(8)                     matplotlib.axes.Axes.set_ylim
(8)                     Notes
(8)                     -----
(8)                     For 3D Axes, this method additionally takes *zmin*, *zmax* as
(8)                     parameters and likewise returns them.
(8)                     """
(8)                     if isinstance(arg, (str, bool)):
(12)                        if arg is True:
(16)                            arg = 'on'
(12)                        if arg is False:
(16)                            arg = 'off'
(12)                        arg = arg.lower()
(12)                        if arg == 'on':
(16)                            self.set_axis_on()
(12)                        elif arg == 'off':
(16)                            self.set_axis_off()
(12)                        elif arg in [
(20)                                'equal', 'tight', 'scaled', 'auto', 'image', 'square']:
(16)                            self.set_autoscale_on(True)
(16)                            self.set_aspect('auto')
(16)                            self.autoscale_view(tight=False)
(16)                            if arg == 'equal':
(20)                                self.set_aspect('equal', adjustable='datalim')
(16)                            elif arg == 'scaled':
(20)                                self.set_aspect('equal', adjustable='box', anchor='C')
(20)                                self.set_autoscale_on(False)  # Req. by Mark Bakker
(16)                            elif arg == 'tight':
(20)                                self.autoscale_view(tight=True)
(20)                                self.set_autoscale_on(False)
(16)                            elif arg == 'image':
(20)                                self.autoscale_view(tight=True)
(20)                                self.set_autoscale_on(False)
(20)                                self.set_aspect('equal', adjustable='box', anchor='C')
(16)                            elif arg == 'square':
(20)                                self.set_aspect('equal', adjustable='box', anchor='C')
(20)                                self.set_autoscale_on(False)
(20)                                xlim = self.get_xlim()
(20)                                ylim = self.get_ylim()
(20)                                edge_size = max(np.diff(xlim), np.diff(ylim))[0]
(20)                                self.set_xlim([xlim[0], xlim[0] + edge_size],
(34)                                              emit=emit, auto=False)
(20)                                self.set_ylim([ylim[0], ylim[0] + edge_size],
(34)                                              emit=emit, auto=False)
(12)                        else:
(16)                            raise ValueError(f"Unrecognized string {arg!r} to axis; "
(33)                                             "try 'on' or 'off'")
```

```
(8)                 else:
(12)                    if arg is not None:
(16)                        if len(arg) != 2*len(self._axis_names):
(20)                            raise TypeError(
(24)                                "The first argument to axis() must be an iterable of
the form "
(24)                                "[{}]".format(", ".join(
(28)                                    f"{name}min, {name}max" for name in
self._axis_names)))
(16)                        limits = {
(20)                            name: arg[2*i:2*(i+1)]
(20)                            for i, name in enumerate(self._axis_names)
(16)                        }
(12)                    else:
(16)                        limits = {}
(16)                        for name in self._axis_names:
(20)                            ax_min = kwargs.pop(f'{name}min', None)
(20)                            ax_max = kwargs.pop(f'{name}max', None)
(20)                            limits[name] = (ax_min, ax_max)
(12)                    for name, (ax_min, ax_max) in limits.items():
(16)                        ax_auto = (None  # Keep autoscale state as is.
(27)                                   if ax_min is None and ax_max is None
(27)                                   else False)  # Turn off autoscale.
(16)                        set_ax_lim = getattr(self, f'set_{name}lim')
(16)                        set_ax_lim(ax_min, ax_max, emit=emit, auto=ax_auto)
(8)                 if kwargs:
(12)                    raise _api.kwarg_error("axis", kwargs)
(8)                 lims = ()
(8)                 for name in self._axis_names:
(12)                    get_ax_lim = getattr(self, f'get_{name}lim')
(12)                    lims += get_ax_lim()
(8)                 return lims
(4)             def get_legend(self):
(8)                 """Return the `.Legend` instance, or None if no legend is defined."""
(8)                 return self.legend_
(4)             def get_images(self):
(8)                 r"""Return a list of `.AxesImage`\s contained by the Axes."""
(8)                 return cbook.silent_list('AxesImage', self.images)
(4)             def get_lines(self):
(8)                 """Return a list of lines contained by the Axes."""
(8)                 return cbook.silent_list('Line2D', self.lines)
(4)             def get_xaxis(self):
(8)                 """
(8)                 [*Discouraged*] Return the XAxis instance.
(8)                 .. admonition:: Discouraged
(12)                    The use of this function is discouraged. You should instead
(12)                    directly access the attribute ``ax.xaxis``.
(8)                 """
(8)                 return self.xaxis
(4)             def get_yaxis(self):
(8)                 """
(8)                 [*Discouraged*] Return the YAxis instance.
(8)                 .. admonition:: Discouraged
(12)                    The use of this function is discouraged. You should instead
(12)                    directly access the attribute ``ax.yaxis``.
(8)                 """
(8)                 return self.yaxis
(4)             get_xgridlines = _axis_method_wrapper("xaxis", "get_gridlines")
(4)             get_xticklines = _axis_method_wrapper("xaxis", "get_ticklines")
(4)             get_ygridlines = _axis_method_wrapper("yaxis", "get_gridlines")
(4)             get_yticklines = _axis_method_wrapper("yaxis", "get_ticklines")
(4)             def _sci(self, im):
(8)                 """
(8)                 Set the current image.
(8)                 This image will be the target of colormap functions like
(8)                 ``pyplot.viridis``, and other functions such as `~.pyplot.clim`.  The
(8)                 current image is an attribute of the current Axes.
(8)                 """
(8)                 _api.check_isinstance((mcoll.Collection, mimage.AxesImage), im=im)
```

```
(8)                 if im not in self._children:
(12)                    raise ValueError("Argument must be an image or collection in this
Axes")
(8)                 self._current_image = im
(4)             def _gci(self):
(8)                 """Helper for `~matplotlib.pyplot.gci`; do not use elsewhere."""
(8)                 return self._current_image
(4)             def has_data(self):
(8)                 """
(8)                 Return whether any artists have been added to the Axes.
(8)                 This should not be used to determine whether the *dataLim*
(8)                 need to be updated, and may not actually be useful for
(8)                 anything.
(8)                 """
(8)                 return any(isinstance(a, (mcoll.Collection, mimage.AxesImage,
(34)                                          mlines.Line2D, mpatches.Patch))
(19)                           for a in self._children)
(4)             def add_artist(self, a):
(8)                 """
(8)                 Add an `.Artist` to the Axes; return the artist.
(8)                 Use `add_artist` only for artists for which there is no dedicated
(8)                 "add" method; and if necessary, use a method such as `update_datalim`
(8)                 to manually update the dataLim if the artist is to be included in
(8)                 autoscaling.
(8)                 If no ``transform`` has been specified when creating the artist (e.g.
(8)                 ``artist.get_transform() == None``) then the transform is set to
(8)                 ``ax.transData``.
(8)                 """
(8)                 a.axes = self
(8)                 self._children.append(a)
(8)                 a._remove_method = self._children.remove
(8)                 self._set_artist_props(a)
(8)                 if a.get_clip_path() is None:
(12)                    a.set_clip_path(self.patch)
(8)                 self.stale = True
(8)                 return a
(4)             def add_child_axes(self, ax):
(8)                 """
(8)                 Add an `.AxesBase` to the Axes' children; return the child Axes.
(8)                 This is the lowlevel version.  See `.axes.Axes.inset_axes`.
(8)                 """
(8)                 ax._axes = self
(8)                 ax.stale_callback = martist._stale_axes_callback
(8)                 self.child_axes.append(ax)
(8)                 ax._remove_method = functools.partial(
(12)                    self.figure._remove_axes, owners=[self.child_axes])
(8)                 self.stale = True
(8)                 return ax
(4)             def add_collection(self, collection, autolim=True):
(8)                 """
(8)                 Add a `.Collection` to the Axes; return the collection.
(8)                 """
(8)                 _api.check_isinstance(mcoll.Collection, collection=collection)
(8)                 if not collection.get_label():
(12)                    collection.set_label(f'_child{len(self._children)}')
(8)                 self._children.append(collection)
(8)                 collection._remove_method = self._children.remove
(8)                 self._set_artist_props(collection)
(8)                 if collection.get_clip_path() is None:
(12)                    collection.set_clip_path(self.patch)
(8)                 if autolim:
(12)                    self._unstale_viewLim()
(12)                    datalim = collection.get_datalim(self.transData)
(12)                    points = datalim.get_points()
(12)                    if not np.isinf(datalim.minpos).all():
(16)                        points = np.concatenate([points, [datalim.minpos]])
(12)                    self.update_datalim(points)
(8)                 self.stale = True
(8)                 return collection
```

```
(4)     def add_image(self, image):
(8)         """
(8)         Add an `.AxesImage` to the Axes; return the image.
(8)         """
(8)         _api.check_isinstance(mimage.AxesImage, image=image)
(8)         self._set_artist_props(image)
(8)         if not image.get_label():
(12)            image.set_label(f'_child{len(self._children)}')
(8)         self._children.append(image)
(8)         image._remove_method = self._children.remove
(8)         self.stale = True
(8)         return image
(4)     def _update_image_limits(self, image):
(8)         xmin, xmax, ymin, ymax = image.get_extent()
(8)         self.axes.update_datalim(((xmin, ymin), (xmax, ymax)))
(4)     def add_line(self, line):
(8)         """
(8)         Add a `.Line2D` to the Axes; return the line.
(8)         """
(8)         _api.check_isinstance(mlines.Line2D, line=line)
(8)         self._set_artist_props(line)
(8)         if line.get_clip_path() is None:
(12)            line.set_clip_path(self.patch)
(8)         self._update_line_limits(line)
(8)         if not line.get_label():
(12)            line.set_label(f'_child{len(self._children)}')
(8)         self._children.append(line)
(8)         line._remove_method = self._children.remove
(8)         self.stale = True
(8)         return line
(4)     def _add_text(self, txt):
(8)         """
(8)         Add a `.Text` to the Axes; return the text.
(8)         """
(8)         _api.check_isinstance(mtext.Text, txt=txt)
(8)         self._set_artist_props(txt)
(8)         self._children.append(txt)
(8)         txt._remove_method = self._children.remove
(8)         self.stale = True
(8)         return txt
(4)     def _update_line_limits(self, line):
(8)         """
(8)         Figures out the data limit of the given line, updating self.dataLim.
(8)         """
(8)         path = line.get_path()
(8)         if path.vertices.size == 0:
(12)            return
(8)         line_trf = line.get_transform()
(8)         if line_trf == self.transData:
(12)            data_path = path
(8)         elif any(line_trf.contains_branch_seperately(self.transData)):
(12)            trf_to_data = line_trf - self.transData
(12)            if self.transData.is_affine:
(16)                line_trans_path = line._get_transformed_path()
(16)                na_path, _ = line_trans_path.get_transformed_path_and_affine()
(16)                data_path = trf_to_data.transform_path_affine(na_path)
(12)            else:
(16)                data_path = trf_to_data.transform_path(path)
(8)         else:
(12)            data_path = path
(8)         if not data_path.vertices.size:
(12)            return
(8)         updatex, updatey = line_trf.contains_branch_seperately(self.transData)
(8)         if self.name != "rectilinear":
(12)            if updatex and line_trf == self.get_yaxis_transform():
(16)                updatex = False
(12)            if updatey and line_trf == self.get_xaxis_transform():
(16)                updatey = False
(8)         self.dataLim.update_from_path(data_path,
```

```
(38)                                                self.ignore_existing_data_limits,
(38)                                                updatex=updatex, updatey=updatey)
(8)                     self.ignore_existing_data_limits = False
(4)                 def add_patch(self, p):
(8)                     """
(8)                     Add a `.Patch` to the Axes; return the patch.
(8)                     """
(8)                     _api.check_isinstance(mpatches.Patch, p=p)
(8)                     self._set_artist_props(p)
(8)                     if p.get_clip_path() is None:
(12)                        p.set_clip_path(self.patch)
(8)                     self._update_patch_limits(p)
(8)                     self._children.append(p)
(8)                     p._remove_method = self._children.remove
(8)                     return p
(4)                 def _update_patch_limits(self, patch):
(8)                     """Update the data limits for the given patch."""
(8)                     if (isinstance(patch, mpatches.Rectangle) and
(16)                            ((not patch.get_width()) and (not patch.get_height()))):
(12)                        return
(8)                     p = patch.get_path()
(8)                     vertices = []
(8)                     for curve, code in p.iter_bezier(simplify=False):
(12)                        _, dzeros = curve.axis_aligned_extrema()
(12)                        vertices.append(curve([0, *dzeros, 1]))
(8)                     if len(vertices):
(12)                        vertices = np.vstack(vertices)
(8)                     patch_trf = patch.get_transform()
(8)                     updatex, updatey =
patch_trf.contains_branch_seperately(self.transData)
(8)                     if not (updatex or updatey):
(12)                        return
(8)                     if self.name != "rectilinear":
(12)                        if updatex and patch_trf == self.get_yaxis_transform():
(16)                            updatex = False
(12)                        if updatey and patch_trf == self.get_xaxis_transform():
(16)                            updatey = False
(8)                     trf_to_data = patch_trf - self.transData
(8)                     xys = trf_to_data.transform(vertices)
(8)                     self.update_datalim(xys, updatex=updatex, updatey=updatey)
(4)                 def add_table(self, tab):
(8)                     """
(8)                     Add a `.Table` to the Axes; return the table.
(8)                     """
(8)                     _api.check_isinstance(mtable.Table, tab=tab)
(8)                     self._set_artist_props(tab)
(8)                     self._children.append(tab)
(8)                     if tab.get_clip_path() is None:
(12)                        tab.set_clip_path(self.patch)
(8)                     tab._remove_method = self._children.remove
(8)                     return tab
(4)                 def add_container(self, container):
(8)                     """
(8)                     Add a `.Container` to the Axes' containers; return the container.
(8)                     """
(8)                     label = container.get_label()
(8)                     if not label:
(12)                        container.set_label('_container%d' % len(self.containers))
(8)                     self.containers.append(container)
(8)                     container._remove_method = self.containers.remove
(8)                     return container
(4)                 def _unit_change_handler(self, axis_name, event=None):
(8)                     """
(8)                     Process axis units changes: requests updates to data and view limits.
(8)                     """
(8)                     if event is None:  # Allow connecting
`self._unit_change_handler(name)`
(12)                        return functools.partial(
(16)                            self._unit_change_handler, axis_name, event=object())
```

```
(8)                     _api.check_in_list(self._axis_map, axis_name=axis_name)
(8)                     for line in self.lines:
(12)                        line.recache_always()
(8)                     self.relim()
(8)                     self._request_autoscale_view(axis_name)
(4)                 def relim(self, visible_only=False):
(8)                     """
(8)                     Recompute the data limits based on current artists.
(8)                     At present, `.Collection` instances are not supported.
(8)                     Parameters
(8)                     ----------
(8)                     visible_only : bool, default: False
(12)                        Whether to exclude invisible artists.
(8)                     """
(8)                     self.dataLim.ignore(True)
(8)                     self.dataLim.set_points(mtransforms.Bbox.null().get_points())
(8)                     self.ignore_existing_data_limits = True
(8)                     for artist in self._children:
(12)                        if not visible_only or artist.get_visible():
(16)                            if isinstance(artist, mlines.Line2D):
(20)                                self._update_line_limits(artist)
(16)                            elif isinstance(artist, mpatches.Patch):
(20)                                self._update_patch_limits(artist)
(16)                            elif isinstance(artist, mimage.AxesImage):
(20)                                self._update_image_limits(artist)
(4)                 def update_datalim(self, xys, updatex=True, updatey=True):
(8)                     """
(8)                     Extend the `~.Axes.dataLim` Bbox to include the given points.
(8)                     If no data is set currently, the Bbox will ignore its limits and set
(8)                     the bound to be the bounds of the xydata (*xys*). Otherwise, it will
(8)                     compute the bounds of the union of its current data and the data in
(8)                     *xys*.
(8)                     Parameters
(8)                     ----------
(8)                     xys : 2D array-like
(12)                        The points to include in the data limits Bbox. This can be either
(12)                        a list of (x, y) tuples or a (N, 2) array.
(8)                     updatex, updatey : bool, default: True
(12)                        Whether to update the x/y limits.
(8)                     """
(8)                     xys = np.asarray(xys)
(8)                     if not np.any(np.isfinite(xys)):
(12)                        return
(8)                     self.dataLim.update_from_data_xy(xys,
self.ignore_existing_data_limits,
(41)                                                     updatex=updatex, updatey=updatey)
(8)                     self.ignore_existing_data_limits = False
(4)                 def _process_unit_info(self, datasets=None, kwargs=None, *, convert=True):
(8)                     """
(8)                     Set axis units based on *datasets* and *kwargs*, and optionally apply
(8)                     unit conversions to *datasets*.
(8)                     Parameters
(8)                     ----------
(8)                     datasets : list
(12)                        List of (axis_name, dataset) pairs (where the axis name is defined
(12)                        as in `._axis_map`).  Individual datasets can also be None
(12)                        (which gets passed through).
(8)                     kwargs : dict
(12)                        Other parameters from which unit info (i.e., the *xunits*,
(12)                        *yunits*, *zunits* (for 3D Axes), *runits* and *thetaunits* (for
(12)                        polar) entries) is popped, if present.  Note that this dict is
(12)                        mutated in-place!
(8)                     convert : bool, default: True
(12)                        Whether to return the original datasets or the converted ones.
(8)                     Returns
(8)                     -------
(8)                     list
(12)                        Either the original datasets if *convert* is False, or the
(12)                        converted ones if *convert* is True (the default).
```

```
2035: (8)                    """
2036: (8)                    datasets = datasets or []
2037: (8)                    kwargs = kwargs or {}
2038: (8)                    axis_map = self._axis_map
2039: (8)                    for axis_name, data in datasets:
2040: (12)                       try:
2041: (16)                           axis = axis_map[axis_name]
2042: (12)                       except KeyError:
2043: (16)                           raise ValueError(f"Invalid axis name: {axis_name!r}") from
None
2044: (12)                       if axis is not None and data is not None and not
axis.have_units():
2045: (16)                           axis.update_units(data)
2046: (8)                    for axis_name, axis in axis_map.items():
2047: (12)                       if axis is None:
2048: (16)                           continue
2049: (12)                       units = kwargs.pop(f"{axis_name}units", axis.units)
2050: (12)                       if self.name == "polar":
2051: (16)                           polar_units = {"x": "thetaunits", "y": "runits"}
2052: (16)                           units = kwargs.pop(polar_units[axis_name], units)
2053: (12)                       if units != axis.units and units is not None:
2054: (16)                           axis.set_units(units)
2055: (16)                           for dataset_axis_name, data in datasets:
2056: (20)                               if dataset_axis_name == axis_name and data is not None:
2057: (24)                                   axis.update_units(data)
2058: (8)                    return [axis_map[axis_name].convert_units(data)
2059: (16)                            if convert and data is not None else data
2060: (16)                            for axis_name, data in datasets]
2061: (4)                def in_axes(self, mouseevent):
2062: (8)                    """
2063: (8)                    Return whether the given event (in display coords) is in the Axes.
2064: (8)                    """
2065: (8)                    return self.patch.contains(mouseevent)[0]
2066: (4)                get_autoscalex_on = _axis_method_wrapper("xaxis", "_get_autoscale_on")
2067: (4)                get_autoscaley_on = _axis_method_wrapper("yaxis", "_get_autoscale_on")
2068: (4)                set_autoscalex_on = _axis_method_wrapper("xaxis", "_set_autoscale_on")
2069: (4)                set_autoscaley_on = _axis_method_wrapper("yaxis", "_set_autoscale_on")
2070: (4)                def get_autoscale_on(self):
2071: (8)                    """Return True if each axis is autoscaled, False otherwise."""
2072: (8)                    return all(axis._get_autoscale_on()
2073: (19)                              for axis in self._axis_map.values())
2074: (4)                def set_autoscale_on(self, b):
2075: (8)                    """
2076: (8)                    Set whether autoscaling is applied to each axis on the next draw or
2077: (8)                    call to `.Axes.autoscale_view`.
2078: (8)                    Parameters
2079: (8)                    ----------
2080: (8)                    b : bool
2081: (8)                    """
2082: (8)                    for axis in self._axis_map.values():
2083: (12)                       axis._set_autoscale_on(b)
2084: (4)                @property
2085: (4)                def use_sticky_edges(self):
2086: (8)                    """
2087: (8)                    When autoscaling, whether to obey all `Artist.sticky_edges`.
2088: (8)                    Default is ``True``.
2089: (8)                    Setting this to ``False`` ensures that the specified margins
2090: (8)                    will be applied, even if the plot includes an image, for
2091: (8)                    example, which would otherwise force a view limit to coincide
2092: (8)                    with its data limit.
2093: (8)                    The changing this property does not change the plot until
2094: (8)                    `autoscale` or `autoscale_view` is called.
2095: (8)                    """
2096: (8)                    return self._use_sticky_edges
2097: (4)                @use_sticky_edges.setter
2098: (4)                def use_sticky_edges(self, b):
2099: (8)                    self._use_sticky_edges = bool(b)
2100: (4)                def get_xmargin(self):
2101: (8)                    """
```

```
(8)                 Retrieve autoscaling margin of the x-axis.
(8)                 .. versionadded:: 3.9
(8)                 Returns
(8)                 -------
(8)                 xmargin : float
(8)                 See Also
(8)                 --------
(8)                 matplotlib.axes.Axes.set_xmargin
(8)                 """
(8)                 return self._xmargin
(4)             def get_ymargin(self):
(8)                 """
(8)                 Retrieve autoscaling margin of the y-axis.
(8)                 .. versionadded:: 3.9
(8)                 Returns
(8)                 -------
(8)                 ymargin : float
(8)                 See Also
(8)                 --------
(8)                 matplotlib.axes.Axes.set_ymargin
(8)                 """
(8)                 return self._ymargin
(4)             def set_xmargin(self, m):
(8)                 """
(8)                 Set padding of X data limits prior to autoscaling.
(8)                 *m* times the data interval will be added to each end of that interval
(8)                 before it is used in autoscaling.  If *m* is negative, this will clip
(8)                 the data range instead of expanding it.
(8)                 For example, if your data is in the range [0, 2], a margin of 0.1 will
(8)                 result in a range [-0.2, 2.2]; a margin of -0.1 will result in a range
(8)                 of [0.2, 1.8].
(8)                 Parameters
(8)                 ----------
(8)                 m : float greater than -0.5
(8)                 """
(8)                 if m <= -0.5:
(12)                    raise ValueError("margin must be greater than -0.5")
(8)                 self._xmargin = m
(8)                 self._request_autoscale_view("x")
(8)                 self.stale = True
(4)             def set_ymargin(self, m):
(8)                 """
(8)                 Set padding of Y data limits prior to autoscaling.
(8)                 *m* times the data interval will be added to each end of that interval
(8)                 before it is used in autoscaling.  If *m* is negative, this will clip
(8)                 the data range instead of expanding it.
(8)                 For example, if your data is in the range [0, 2], a margin of 0.1 will
(8)                 result in a range [-0.2, 2.2]; a margin of -0.1 will result in a range
(8)                 of [0.2, 1.8].
(8)                 Parameters
(8)                 ----------
(8)                 m : float greater than -0.5
(8)                 """
(8)                 if m <= -0.5:
(12)                    raise ValueError("margin must be greater than -0.5")
(8)                 self._ymargin = m
(8)                 self._request_autoscale_view("y")
(8)                 self.stale = True
(4)             def margins(self, *margins, x=None, y=None, tight=True):
(8)                 """
(8)                 Set or retrieve autoscaling margins.
(8)                 The padding added to each limit of the Axes is the *margin*
(8)                 times the data interval. All input parameters must be floats
(8)                 greater than -0.5. Passing both positional and keyword
(8)                 arguments is invalid and will raise a TypeError. If no
(8)                 arguments (positional or otherwise) are provided, the current
(8)                 margins will remain unchanged and simply be returned.
(8)                 Specifying any margin changes only the autoscaling; for example,
(8)                 if *xmargin* is not None, then *xmargin* times the X data
```

```
        interval will be added to each end of that interval before
        it is used in autoscaling.
        Parameters
        ----------
        *margins : float, optional
            If a single positional argument is provided, it specifies
            both margins of the x-axis and y-axis limits. If two
            positional arguments are provided, they will be interpreted
            as *xmargin*, *ymargin*. If setting the margin on a single
            axis is desired, use the keyword arguments described below.
        x, y : float, optional
            Specific margin values for the x-axis and y-axis,
            respectively. These cannot be used with positional
            arguments, but can be used individually to alter on e.g.,
            only the y-axis.
        tight : bool or None, default: True
            The *tight* parameter is passed to `~.axes.Axes.autoscale_view`,
            which is executed after a margin is changed; the default
            here is *True*, on the assumption that when margins are
            specified, no additional padding to match tick marks is
            usually desired.  Setting *tight* to *None* preserves
            the previous setting.
        Returns
        -------
        xmargin, ymargin : float
        Notes
        -----
        If a previously used Axes method such as :meth:`pcolor` has set
        :attr:`use_sticky_edges` to `True`, only the limits not set by
        the "sticky artists" will be modified. To force all of the
        margins to be set, set :attr:`use_sticky_edges` to `False`
        before calling :meth:`margins`.
        """
        if margins and (x is not None or y is not None):
            raise TypeError('Cannot pass both positional and keyword '
                            'arguments for x and/or y.')
        elif len(margins) == 1:
            x = y = margins[0]
        elif len(margins) == 2:
            x, y = margins
        elif margins:
            raise TypeError('Must pass a single positional argument for all '
                            'margins, or one for each margin (x, y).')
        if x is None and y is None:
            if tight is not True:
                _api.warn_external(f'ignoring tight={tight!r} in get mode')
            return self._xmargin, self._ymargin
        if tight is not None:
            self._tight = tight
        if x is not None:
            self.set_xmargin(x)
        if y is not None:
            self.set_ymargin(y)
    def set_rasterization_zorder(self, z):
        """
        Set the zorder threshold for rasterization for vector graphics output.
        All artists with a zorder below the given value will be rasterized if
        they support rasterization.
        This setting is ignored for pixel-based output.
        See also :doc:`/gallery/misc/rasterization_demo`.
        Parameters
        ----------
        z : float or None
            The zorder below which artists are rasterized.
            If ``None`` rasterization based on zorder is deactivated.
        """
        self._rasterization_zorder = z
        self.stale = True
    def get_rasterization_zorder(self):
```

```
(8)                 """Return the zorder value below which artists will be rasterized."""
(8)                 return self._rasterization_zorder
(4)             def autoscale(self, enable=True, axis='both', tight=None):
(8)                 """
(8)                 Autoscale the axis view to the data (toggle).
(8)                 Convenience method for simple axis view autoscaling.
(8)                 It turns autoscaling on or off, and then,
(8)                 if autoscaling for either axis is on, it performs
(8)                 the autoscaling on the specified axis or Axes.
(8)                 Parameters
(8)                 ----------
(8)                 enable : bool or None, default: True
(12)                    True turns autoscaling on, False turns it off.
(12)                    None leaves the autoscaling state unchanged.
(8)                 axis : {'both', 'x', 'y'}, default: 'both'
(12)                    The axis on which to operate.  (For 3D Axes, *axis* can also be
set
(12)                    to 'z', and 'both' refers to all three Axes.)
(8)                 tight : bool or None, default: None
(12)                    If True, first set the margins to zero.  Then, this argument is
(12)                    forwarded to `~.axes.Axes.autoscale_view` (regardless of
(12)                    its value); see the description of its behavior there.
(8)                 """
(8)                 if enable is None:
(12)                    scalex = True
(12)                    scaley = True
(8)                 else:
(12)                    if axis in ['x', 'both']:
(16)                        self.set_autoscalex_on(bool(enable))
(16)                        scalex = self.get_autoscalex_on()
(12)                    else:
(16)                        scalex = False
(12)                    if axis in ['y', 'both']:
(16)                        self.set_autoscaley_on(bool(enable))
(16)                        scaley = self.get_autoscaley_on()
(12)                    else:
(16)                        scaley = False
(8)                 if tight and scalex:
(12)                    self._xmargin = 0
(8)                 if tight and scaley:
(12)                    self._ymargin = 0
(8)                 if scalex:
(12)                    self._request_autoscale_view("x", tight=tight)
(8)                 if scaley:
(12)                    self._request_autoscale_view("y", tight=tight)
(4)             def autoscale_view(self, tight=None, scalex=True, scaley=True):
(8)                 """
(8)                 Autoscale the view limits using the data limits.
(8)                 Parameters
(8)                 ----------
(8)                 tight : bool or None
(12)                    If *True*, only expand the axis limits using the margins.  Note
(12)                    that unlike for `autoscale`, ``tight=True`` does *not* set the
(12)                    margins to zero.
(12)                    If *False* and :rc:`axes.autolimit_mode` is 'round_numbers', then
(12)                    after expansion by the margins, further expand the axis limits
(12)                    using the axis major locator.
(12)                    If None (the default), reuse the value set in the previous call to
(12)                    `autoscale_view` (the initial value is False, but the default
style
(12)                    sets :rc:`axes.autolimit_mode` to 'data', in which case this
(12)                    behaves like True).
(8)                 scalex : bool, default: True
(12)                    Whether to autoscale the x-axis.
(8)                 scaley : bool, default: True
(12)                    Whether to autoscale the y-axis.
(8)                 Notes
(8)                 -----
(8)                 The autoscaling preserves any preexisting axis direction reversal.
```

```
(8)                     The data limits are not updated automatically when artist data are
(8)                     changed after the artist has been added to an Axes instance.  In that
(8)                     case, use :meth:`matplotlib.axes.Axes.relim` prior to calling
(8)                     autoscale_view.
(8)                     If the views of the Axes are fixed, e.g. via `set_xlim`, they will
(8)                     not be changed by autoscale_view().
(8)                     See :meth:`matplotlib.axes.Axes.autoscale` for an alternative.
(8)                     """
(8)                     if tight is not None:
(12)                        self._tight = bool(tight)
(8)                     x_stickies = y_stickies = np.array([])
(8)                     if self.use_sticky_edges:
(12)                        if self._xmargin and scalex and self.get_autoscalex_on():
(16)                            x_stickies = np.sort(np.concatenate([
(20)                                artist.sticky_edges.x
(20)                                for ax in self._shared_axes["x"].get_siblings(self)
(20)                                for artist in ax.get_children()]))
(12)                        if self._ymargin and scaley and self.get_autoscaley_on():
(16)                            y_stickies = np.sort(np.concatenate([
(20)                                artist.sticky_edges.y
(20)                                for ax in self._shared_axes["y"].get_siblings(self)
(20)                                for artist in ax.get_children()]))
(8)                     if self.get_xscale() == 'log':
(12)                        x_stickies = x_stickies[x_stickies > 0]
(8)                     if self.get_yscale() == 'log':
(12)                        y_stickies = y_stickies[y_stickies > 0]
(8)                     def handle_single_axis(
(16)                            scale, shared_axes, name, axis, margin, stickies, set_bound):
(12)                        if not (scale and axis._get_autoscale_on()):
(16)                            return  # nothing to do...
(12)                        shared = shared_axes.get_siblings(self)
(12)                        values = [val for ax in shared
(22)                                  for val in getattr(ax.dataLim, f"interval{name}")
(22)                                  if np.isfinite(val)]
(12)                        if values:
(16)                            x0, x1 = (min(values), max(values))
(12)                        elif getattr(self._viewLim, f"mutated{name}")():
(16)                            return
(12)                        else:
(16)                            x0, x1 = (-np.inf, np.inf)
(12)                        locator = axis.get_major_locator()
(12)                        x0, x1 = locator.nonsingular(x0, x1)
(12)                        minimum_minpos = min(
(16)                            getattr(ax.dataLim, f"minpos{name}") for ax in shared)
(12)                        tol = 1e-5 * abs(x1 - x0)
(12)                        i0 = stickies.searchsorted(x0 + tol) - 1
(12)                        x0bound = stickies[i0] if i0 != -1 else None
(12)                        i1 = stickies.searchsorted(x1 - tol)
(12)                        x1bound = stickies[i1] if i1 != len(stickies) else None
(12)                        transform = axis.get_transform()
(12)                        inverse_trans = transform.inverted()
(12)                        x0, x1 = axis._scale.limit_range_for_scale(x0, x1, minimum_minpos)
(12)                        x0t, x1t = transform.transform([x0, x1])
(12)                        delta = (x1t - x0t) * margin
(12)                        if not np.isfinite(delta):
(16)                            delta = 0  # If a bound isn't finite, set margin to zero.
(12)                        x0, x1 = inverse_trans.transform([x0t - delta, x1t + delta])
(12)                        if x0bound is not None:
(16)                            x0 = max(x0, x0bound)
(12)                        if x1bound is not None:
(16)                            x1 = min(x1, x1bound)
(12)                        if not self._tight:
(16)                            x0, x1 = locator.view_limits(x0, x1)
(12)                        set_bound(x0, x1)
(8)                     handle_single_axis(
(12)                        scalex, self._shared_axes["x"], 'x', self.xaxis, self._xmargin,
(12)                        x_stickies, self.set_xbound)
(8)                     handle_single_axis(
(12)                        scaley, self._shared_axes["y"], 'y', self.yaxis, self._ymargin,
```

```
(12)                        y_stickies, self.set_ybound)
(4)                 def _update_title_position(self, renderer):
(8)                     """
(8)                     Update the title position based on the bounding box enclosing
(8)                     all the ticklabels and x-axis spine and xlabel...
(8)                     """
(8)                     if self._autotitlepos is not None and not self._autotitlepos:
(12)                        _log.debug('title position was updated manually, not adjusting')
(12)                        return
(8)                     titles = (self.title, self._left_title, self._right_title)
(8)                     axs = set()
(8)                     axs.update(self.child_axes)
(8)                     axs.update(self._twinned_axes.get_siblings(self))
(8)
axs.update(self.figure._align_label_groups['title'].get_siblings(self))
(8)                     for ax in self.child_axes:  # Child positions must be updated first.
(12)                        locator = ax.get_axes_locator()
(12)                        ax.apply_aspect(locator(self, renderer) if locator else None)
(8)                     for title in titles:
(12)                        x, _ = title.get_position()
(12)                        title.set_position((x, 1.0))
(12)                        top = -np.inf
(12)                        for ax in axs:
(16)                            bb = None
(16)                            if (ax.xaxis.get_ticks_position() in ['top', 'unknown']
(24)                                    or ax.xaxis.get_label_position() == 'top'):
(20)                                bb = ax.xaxis.get_tightbbox(renderer)
(16)                            if bb is None:
(20)                                if 'outline' in ax.spines:
(24)                                    bb = ax.spines['outline'].get_window_extent()
(20)                                else:
(24)                                    bb = ax.get_window_extent(renderer)
(16)                            top = max(top, bb.ymax)
(16)                            if title.get_text():
(20)                                ax.yaxis.get_tightbbox(renderer)  # update offsetText
(20)                                if ax.yaxis.offsetText.get_text():
(24)                                    bb = ax.yaxis.offsetText.get_tightbbox(renderer)
(24)                                    if bb.intersection(title.get_tightbbox(renderer), bb):
(28)                                        top = bb.ymax
(12)                        if top < 0:
(16)                            _log.debug('top of Axes not in the figure, so title not
moved')
(16)                            return
(12)                        if title.get_window_extent(renderer).ymin < top:
(16)                            _, y = self.transAxes.inverted().transform((0, top))
(16)                            title.set_position((x, y))
(16)                            if title.get_window_extent(renderer).ymin < top:
(20)                                _, y = self.transAxes.inverted().transform(
(24)                                    (0., 2 * top -
title.get_window_extent(renderer).ymin))
(20)                                title.set_position((x, y))
(8)                     ymax = max(title.get_position()[1] for title in titles)
(8)                     for title in titles:
(12)                        x, _ = title.get_position()
(12)                        title.set_position((x, ymax))
(4)                 @martist.allow_rasterization
(4)                 def draw(self, renderer):
(8)                     if renderer is None:
(12)                        raise RuntimeError('No renderer defined')
(8)                     if not self.get_visible():
(12)                        return
(8)                     self._unstale_viewLim()
(8)                     renderer.open_group('axes', gid=self.get_gid())
(8)                     self._stale = True
(8)                     locator = self.get_axes_locator()
(8)                     self.apply_aspect(locator(self, renderer) if locator else None)
(8)                     artists = self.get_children()
(8)                     artists.remove(self.patch)
(8)                     if not (self.axison and self._frameon):
```

```
(12)                        for spine in self.spines.values():
(16)                            artists.remove(spine)
(8)                     self._update_title_position(renderer)
(8)                     if not self.axison:
(12)                        for _axis in self._axis_map.values():
(16)                            artists.remove(_axis)
(8)                     if not self.figure.canvas.is_saving():
(12)                        artists = [
(16)                            a for a in artists
(16)                            if not a.get_animated() or isinstance(a, mimage.AxesImage)]
(8)                     artists = sorted(artists, key=attrgetter('zorder'))
(8)                     rasterization_zorder = self._rasterization_zorder
(8)                     if (rasterization_zorder is not None and
(16)                            artists and artists[0].zorder < rasterization_zorder):
(12)                        split_index = np.searchsorted(
(16)                            [art.zorder for art in artists],
(16)                            rasterization_zorder, side='right'
(12)                        )
(12)                        artists_rasterized = artists[:split_index]
(12)                        artists = artists[split_index:]
(8)                     else:
(12)                        artists_rasterized = []
(8)                     if self.axison and self._frameon:
(12)                        if artists_rasterized:
(16)                            artists_rasterized = [self.patch] + artists_rasterized
(12)                        else:
(16)                            artists = [self.patch] + artists
(8)                     if artists_rasterized:
(12)                        _draw_rasterized(self.figure, artists_rasterized, renderer)
(8)                     mimage._draw_list_compositing_images(
(12)                        renderer, self, artists, self.figure.suppressComposite)
(8)                     renderer.close_group('axes')
(8)                     self.stale = False
(4)                 def draw_artist(self, a):
(8)                     """
(8)                     Efficiently redraw a single artist.
(8)                     """
(8)                     a.draw(self.figure.canvas.get_renderer())
(4)                 def redraw_in_frame(self):
(8)                     """
(8)                     Efficiently redraw Axes data, but not axis ticks, labels, etc.
(8)                     """
(8)                     with ExitStack() as stack:
(12)                        for artist in [*self._axis_map.values(),
(27)                                       self.title, self._left_title, self._right_title]:
(16)                            stack.enter_context(artist._cm_set(visible=False))
(12)                        self.draw(self.figure.canvas.get_renderer())
(4)                 def get_frame_on(self):
(8)                     """Get whether the Axes rectangle patch is drawn."""
(8)                     return self._frameon
(4)                 def set_frame_on(self, b):
(8)                     """
(8)                     Set whether the Axes rectangle patch is drawn.
(8)                     Parameters
(8)                     ----------
(8)                     b : bool
(8)                     """
(8)                     self._frameon = b
(8)                     self.stale = True
(4)                 def get_axisbelow(self):
(8)                     """
(8)                     Get whether axis ticks and gridlines are above or below most artists.
(8)                     Returns
(8)                     -------
(8)                     bool or 'line'
(8)                     See Also
(8)                     --------
(8)                     set_axisbelow
(8)                     """
```

```
(8)                 return self._axisbelow
(4)             def set_axisbelow(self, b):
(8)                 """
(8)                 Set whether axis ticks and gridlines are above or below most artists.
(8)                 This controls the zorder of the ticks and gridlines. For more
(8)                 information on the zorder see :doc:`/gallery/misc/zorder_demo`.
(8)                 Parameters
(8)                 ----------
(8)                 b : bool or 'line'
(12)                    Possible values:
(12)                    - *True* (zorder = 0.5): Ticks and gridlines are below patches and
(14)                      lines, though still above images.
(12)                    - 'line' (zorder = 1.5): Ticks and gridlines are above patches
(14)                      (e.g. rectangles, with default zorder = 1) but still below lines
(14)                      and markers (with their default zorder = 2).
(12)                    - *False* (zorder = 2.5): Ticks and gridlines are above patches
(14)                      and lines / markers.
(8)                 Notes
(8)                 -----
(8)                 For more control, call the `~.Artist.set_zorder` method of each axis.
(8)                 See Also
(8)                 --------
(8)                 get_axisbelow
(8)                 """
(8)                 self._axisbelow = axisbelow = validate_axisbelow(b)
(8)                 zorder = {
(12)                    True: 0.5,
(12)                    'line': 1.5,
(12)                    False: 2.5,
(8)                 }[axisbelow]
(8)                 for axis in self._axis_map.values():
(12)                    axis.set_zorder(zorder)
(8)                 self.stale = True
(4)             @_docstring.dedent_interpd
(4)             def grid(self, visible=None, which='major', axis='both', **kwargs):
(8)                 """
(8)                 Configure the grid lines.
(8)                 Parameters
(8)                 ----------
(8)                 visible : bool or None, optional
(12)                    Whether to show the grid lines.  If any *kwargs* are supplied, it
(12)                    is assumed you want the grid on and *visible* will be set to True.
(12)                    If *visible* is *None* and there are no *kwargs*, this toggles the
(12)                    visibility of the lines.
(8)                 which : {'major', 'minor', 'both'}, optional
(12)                    The grid lines to apply the changes on.
(8)                 axis : {'both', 'x', 'y'}, optional
(12)                    The axis to apply the changes on.
(8)                 **kwargs : `~matplotlib.lines.Line2D` properties
(12)                    Define the line properties of the grid, e.g.::
(16)                        grid(color='r', linestyle='-', linewidth=2)
(12)                    Valid keyword arguments are:
(12)                    %(Line2D:kwdoc)s
(8)                 Notes
(8)                 -----
(8)                 The axis is drawn as a unit, so the effective zorder for drawing the
(8)                 grid is determined by the zorder of each axis, not by the zorder of
the
(8)                 `.Line2D` objects comprising the grid.  Therefore, to set grid zorder,
(8)                 use `.set_axisbelow` or, for more control, call the
(8)                 `~.Artist.set_zorder` method of each axis.
(8)                 """
(8)                 _api.check_in_list(['x', 'y', 'both'], axis=axis)
(8)                 if axis in ['x', 'both']:
(12)                    self.xaxis.grid(visible, which=which, **kwargs)
(8)                 if axis in ['y', 'both']:
(12)                    self.yaxis.grid(visible, which=which, **kwargs)
(4)             def ticklabel_format(self, *, axis='both', style=None, scilimits=None,
(25)                                 useOffset=None, useLocale=None, useMathText=None):
```

```
(8)                 r"""
(8)                 Configure the `.ScalarFormatter` used by default for linear Axes.
(8)                 If a parameter is not set, the corresponding property of the formatter
(8)                 is left unchanged.
(8)                 Parameters
(8)                 ----------
(8)                 axis : {'x', 'y', 'both'}, default: 'both'
(12)                    The axis to configure.  Only major ticks are affected.
(8)                 style : {'sci', 'scientific', 'plain'}
(12)                    Whether to use scientific notation.
(12)                    The formatter default is to use scientific notation.
(12)                    'sci' is equivalent to 'scientific'.
(8)                 scilimits : pair of ints (m, n)
(12)                    Scientific notation is used only for numbers outside the range
(12)                    10\ :sup:`m` to 10\ :sup:`n` (and only if the formatter is
(12)                    configured to use scientific notation at all).  Use (0, 0) to
(12)                    include all numbers.  Use (m, m) where m != 0 to fix the order of
(12)                    magnitude to 10\ :sup:`m`.
(12)                    The formatter default is :rc:`axes.formatter.limits`.
(8)                 useOffset : bool or float
(12)                    If True, the offset is calculated as needed.
(12)                    If False, no offset is used.
(12)                    If a numeric value, it sets the offset.
(12)                    The formatter default is :rc:`axes.formatter.useoffset`.
(8)                 useLocale : bool
(12)                    Whether to format the number using the current locale or using the
(12)                    C (English) locale.  This affects e.g. the decimal separator.  The
(12)                    formatter default is :rc:`axes.formatter.use_locale`.
(8)                 useMathText : bool
(12)                    Render the offset and scientific notation in mathtext.
(12)                    The formatter default is :rc:`axes.formatter.use_mathtext`.
(8)                 Raises
(8)                 ------
(8)                 AttributeError
(12)                    If the current formatter is not a `.ScalarFormatter`.
(8)                 """
(8)                 if isinstance(style, str):
(12)                    style = style.lower()
(8)                 axis = axis.lower()
(8)                 if scilimits is not None:
(12)                    try:
(16)                        m, n = scilimits
(16)                        m + n + 1  # check that both are numbers
(12)                    except (ValueError, TypeError) as err:
(16)                        raise ValueError("scilimits must be a sequence of 2 integers"
(33)                                         ) from err
(8)                 STYLES = {'sci': True, 'scientific': True, 'plain': False, '': None,
None: None}
(8)                 is_sci_style = _api.check_getitem(STYLES, style=style)
(8)                 axis_map = {**{k: [v] for k, v in self._axis_map.items()},
(20)                            'both': list(self._axis_map.values())}
(8)                 axises = _api.check_getitem(axis_map, axis=axis)
(8)                 try:
(12)                    for axis in axises:
(16)                        if is_sci_style is not None:
(20)                            axis.major.formatter.set_scientific(is_sci_style)
(16)                        if scilimits is not None:
(20)                            axis.major.formatter.set_powerlimits(scilimits)
(16)                        if useOffset is not None:
(20)                            axis.major.formatter.set_useOffset(useOffset)
(16)                        if useLocale is not None:
(20)                            axis.major.formatter.set_useLocale(useLocale)
(16)                        if useMathText is not None:
(20)                            axis.major.formatter.set_useMathText(useMathText)
(8)                 except AttributeError as err:
(12)                    raise AttributeError(
(16)                        "This method only works with the ScalarFormatter") from err
(4)             def locator_params(self, axis='both', tight=None, **kwargs):
(8)                 """
```

```
(8)                     Control behavior of major tick locators.
(8)                     Because the locator is involved in autoscaling,
`~.Axes.autoscale_view`
(8)                     is called automatically after the parameters are changed.
(8)                     Parameters
(8)                     ----------
(8)                     axis : {'both', 'x', 'y'}, default: 'both'
(12)                        The axis on which to operate.  (For 3D Axes, *axis* can also be
(12)                        set to 'z', and 'both' refers to all three axes.)
(8)                     tight : bool or None, optional
(12)                        Parameter passed to `~.Axes.autoscale_view`.
(12)                        Default is None, for no change.
(8)                     Other Parameters
(8)                     ----------------
(8)                     **kwargs
(12)                        Remaining keyword arguments are passed to directly to the
(12)                        ``set_params()`` method of the locator. Supported keywords depend
(12)                        on the type of the locator. See for example
(12)                        `~.ticker.MaxNLocator.set_params` for the `.ticker.MaxNLocator`
(12)                        used by default for linear.
(8)                     Examples
(8)                     --------
(8)                     When plotting small subplots, one might want to reduce the maximum
(8)                     number of ticks and use tight bounds, for example::
(12)                        ax.locator_params(tight=True, nbins=4)
(8)                     """
(8)                     _api.check_in_list([*self._axis_names, "both"], axis=axis)
(8)                     for name in self._axis_names:
(12)                        if axis in [name, "both"]:
(16)                            loc = self._axis_map[name].get_major_locator()
(16)                            loc.set_params(**kwargs)
(16)                            self._request_autoscale_view(name, tight=tight)
(8)                     self.stale = True
(4)                 def tick_params(self, axis='both', **kwargs):
(8)                     """
(8)                     Change the appearance of ticks, tick labels, and gridlines.
(8)                     Tick properties that are not explicitly set using the keyword
(8)                     arguments remain unchanged unless *reset* is True. For the current
(8)                     style settings, see `.Axis.get_tick_params`.
(8)                     Parameters
(8)                     ----------
(8)                     axis : {'x', 'y', 'both'}, default: 'both'
(12)                        The axis to which the parameters are applied.
(8)                     which : {'major', 'minor', 'both'}, default: 'major'
(12)                        The group of ticks to which the parameters are applied.
(8)                     reset : bool, default: False
(12)                        Whether to reset the ticks to defaults before updating them.
(8)                     Other Parameters
(8)                     ----------------
(8)                     direction : {'in', 'out', 'inout'}
(12)                        Puts ticks inside the Axes, outside the Axes, or both.
(8)                     length : float
(12)                        Tick length in points.
(8)                     width : float
(12)                        Tick width in points.
(8)                     color : :mpltype:`color`
(12)                        Tick color.
(8)                     pad : float
(12)                        Distance in points between tick and label.
(8)                     labelsize : float or str
(12)                        Tick label font size in points or as a string (e.g., 'large').
(8)                     labelcolor : :mpltype:`color`
(12)                        Tick label color.
(8)                     labelfontfamily : str
(12)                        Tick label font.
(8)                     colors : :mpltype:`color`
(12)                        Tick color and label color.
(8)                     zorder : float
(12)                        Tick and label zorder.
```

```
2715: (8)                   bottom, top, left, right : bool
2716: (12)                      Whether to draw the respective ticks.
2717: (8)                   labelbottom, labeltop, labelleft, labelright : bool
2718: (12)                      Whether to draw the respective tick labels.
2719: (8)                   labelrotation : float
2720: (12)                      Tick label rotation
2721: (8)                   grid_color : :mpltype:`color`
2722: (12)                      Gridline color.
2723: (8)                   grid_alpha : float
2724: (12)                      Transparency of gridlines: 0 (transparent) to 1 (opaque).
2725: (8)                   grid_linewidth : float
2726: (12)                      Width of gridlines in points.
2727: (8)                   grid_linestyle : str
2728: (12)                      Any valid `.Line2D` line style spec.
2729: (8)                   Examples
2730: (8)                   --------
2731: (8)                   ::
2732: (12)                      ax.tick_params(direction='out', length=6, width=2, colors='r',
2733: (27)                                     grid_color='r', grid_alpha=0.5)
2734: (8)                   This will make all major ticks be red, pointing out of the box,
2735: (8)                   and with dimensions 6 points by 2 points.  Tick labels will
2736: (8)                   also be red.  Gridlines will be red and translucent.
2737: (8)                   """
2738: (8)                   _api.check_in_list(['x', 'y', 'both'], axis=axis)
2739: (8)                   if axis in ['x', 'both']:
2740: (12)                      xkw = dict(kwargs)
2741: (12)                      xkw.pop('left', None)
2742: (12)                      xkw.pop('right', None)
2743: (12)                      xkw.pop('labelleft', None)
2744: (12)                      xkw.pop('labelright', None)
2745: (12)                      self.xaxis.set_tick_params(**xkw)
2746: (8)                   if axis in ['y', 'both']:
2747: (12)                      ykw = dict(kwargs)
2748: (12)                      ykw.pop('top', None)
2749: (12)                      ykw.pop('bottom', None)
2750: (12)                      ykw.pop('labeltop', None)
2751: (12)                      ykw.pop('labelbottom', None)
2752: (12)                      self.yaxis.set_tick_params(**ykw)
2753: (4)               def set_axis_off(self):
2754: (8)                   """
2755: (8)                   Hide all visual components of the x- and y-axis.
2756: (8)                   This sets a flag to suppress drawing of all axis decorations, i.e.
2757: (8)                   axis labels, axis spines, and the axis tick component (tick markers,
2758: (8)                   tick labels, and grid lines). Individual visibility settings of these
2759: (8)                   components are ignored as long as `set_axis_off()` is in effect.
2760: (8)                   """
2761: (8)                   self.axison = False
2762: (8)                   self.stale = True
2763: (4)               def set_axis_on(self):
2764: (8)                   """
2765: (8)                   Do not hide all visual components of the x- and y-axis.
2766: (8)                   This reverts the effect of a prior `.set_axis_off()` call. Whether the
2767: (8)                   individual axis decorations are drawn is controlled by their
respective
2768: (8)                   visibility settings.
2769: (8)                   This is on by default.
2770: (8)                   """
2771: (8)                   self.axison = True
2772: (8)                   self.stale = True
2773: (4)               def get_xlabel(self):
2774: (8)                   """
2775: (8)                   Get the xlabel text string.
2776: (8)                   """
2777: (8)                   label = self.xaxis.get_label()
2778: (8)                   return label.get_text()
2779: (4)               def set_xlabel(self, xlabel, fontdict=None, labelpad=None, *,
2780: (19)                             loc=None, **kwargs):
2781: (8)                   """
2782: (8)                   Set the label for the x-axis.
```

```
(8)                 Parameters
(8)                 ----------
(8)                 xlabel : str
(12)                    The label text.
(8)                 labelpad : float, default: :rc:`axes.labelpad`
(12)                    Spacing in points from the Axes bounding box including ticks
(12)                    and tick labels.  If None, the previous value is left as is.
(8)                 loc : {'left', 'center', 'right'}, default: :rc:`xaxis.labellocation`
(12)                    The label position. This is a high-level alternative for passing
(12)                    parameters *x* and *horizontalalignment*.
(8)                 Other Parameters
(8)                 ----------------
(8)                 **kwargs : `~matplotlib.text.Text` properties
(12)                    `.Text` properties control the appearance of the label.
(8)                 See Also
(8)                 --------
(8)                 text : Documents the properties supported by `.Text`.
(8)                 """
(8)                 if labelpad is not None:
(12)                    self.xaxis.labelpad = labelpad
(8)                 protected_kw = ['x', 'horizontalalignment', 'ha']
(8)                 if {*kwargs} & {*protected_kw}:
(12)                    if loc is not None:
(16)                        raise TypeError(f"Specifying 'loc' is disallowed when any of "
(32)                                        f"its corresponding low level keyword "
(32)                                        f"arguments ({protected_kw}) are also "
(32)                                        f"supplied")
(8)                 else:
(12)                    loc = (loc if loc is not None
(19)                           else mpl.rcParams['xaxis.labellocation'])
(12)                    _api.check_in_list(('left', 'center', 'right'), loc=loc)
(12)                    x = {
(16)                        'left': 0,
(16)                        'center': 0.5,
(16)                        'right': 1,
(12)                    }[loc]
(12)                    kwargs.update(x=x, horizontalalignment=loc)
(8)                 return self.xaxis.set_label_text(xlabel, fontdict, **kwargs)
(4)             def invert_xaxis(self):
(8)                 """
(8)                 Invert the x-axis.
(8)                 See Also
(8)                 --------
(8)                 xaxis_inverted
(8)                 get_xlim, set_xlim
(8)                 get_xbound, set_xbound
(8)                 """
(8)                 self.xaxis.set_inverted(not self.xaxis.get_inverted())
(4)             xaxis_inverted = _axis_method_wrapper("xaxis", "get_inverted")
(4)             def get_xbound(self):
(8)                 """
(8)                 Return the lower and upper x-axis bounds, in increasing order.
(8)                 See Also
(8)                 --------
(8)                 set_xbound
(8)                 get_xlim, set_xlim
(8)                 invert_xaxis, xaxis_inverted
(8)                 """
(8)                 left, right = self.get_xlim()
(8)                 if left < right:
(12)                    return left, right
(8)                 else:
(12)                    return right, left
(4)             def set_xbound(self, lower=None, upper=None):
(8)                 """
(8)                 Set the lower and upper numerical bounds of the x-axis.
(8)                 This method will honor axis inversion regardless of parameter order.
(8)                 It will not change the autoscaling setting (`.get_autoscalex_on()`).
(8)                 Parameters
```

```
(8)                 ----------
(8)                 lower, upper : float or None
(12)                    The lower and upper bounds. If *None*, the respective axis bound
(12)                    is not modified.
(12)                    .. ACCEPTS: (lower: float, upper: float)
(8)                 See Also
(8)                 --------
(8)                 get_xbound
(8)                 get_xlim, set_xlim
(8)                 invert_xaxis, xaxis_inverted
(8)                 """
(8)                 if upper is None and np.iterable(lower):
(12)                    lower, upper = lower
(8)                 old_lower, old_upper = self.get_xbound()
(8)                 if lower is None:
(12)                    lower = old_lower
(8)                 if upper is None:
(12)                    upper = old_upper
(8)                 self.set_xlim(sorted((lower, upper),
(29)                                  reverse=bool(self.xaxis_inverted())),
(22)                              auto=None)
(4)             def get_xlim(self):
(8)                 """
(8)                 Return the x-axis view limits.
(8)                 Returns
(8)                 -------
(8)                 left, right : (float, float)
(12)                    The current x-axis limits in data coordinates.
(8)                 See Also
(8)                 --------
(8)                 .Axes.set_xlim
(8)                 .Axes.set_xbound, .Axes.get_xbound
(8)                 .Axes.invert_xaxis, .Axes.xaxis_inverted
(8)                 Notes
(8)                 -----
(8)                 The x-axis may be inverted, in which case the *left* value will
(8)                 be greater than the *right* value.
(8)                 """
(8)                 return tuple(self.viewLim.intervalx)
(4)             def _validate_converted_limits(self, limit, convert):
(8)                 """
(8)                 Raise ValueError if converted limits are non-finite.
(8)                 Note that this function also accepts None as a limit argument.
(8)                 Returns
(8)                 -------
(8)                 The limit value after call to convert(), or None if limit is None.
(8)                 """
(8)                 if limit is not None:
(12)                    converted_limit = convert(limit)
(12)                    if isinstance(converted_limit, np.ndarray):
(16)                        converted_limit = converted_limit.squeeze()
(12)                    if (isinstance(converted_limit, Real)
(20)                            and not np.isfinite(converted_limit)):
(16)                        raise ValueError("Axis limits cannot be NaN or Inf")
(12)                    return converted_limit
(4)             def set_xlim(self, left=None, right=None, *, emit=True, auto=False,
(17)                         xmin=None, xmax=None):
(8)                 """
(8)                 Set the x-axis view limits.
(8)                 Parameters
(8)                 ----------
(8)                 left : float, optional
(12)                    The left xlim in data coordinates. Passing *None* leaves the
(12)                    limit unchanged.
(12)                    The left and right xlims may also be passed as the tuple
(12)                    (*left*, *right*) as the first positional argument (or as
(12)                    the *left* keyword argument).
(12)                    .. ACCEPTS: (left: float, right: float)
(8)                 right : float, optional
```

```
(12)                        The right xlim in data coordinates. Passing *None* leaves the
(12)                        limit unchanged.
(8)                     emit : bool, default: True
(12)                        Whether to notify observers of limit change.
(8)                     auto : bool or None, default: False
(12)                        Whether to turn on autoscaling of the x-axis. True turns on,
(12)                        False turns off, None leaves unchanged.
(8)                     xmin, xmax : float, optional
(12)                        They are equivalent to left and right respectively, and it is an
(12)                        error to pass both *xmin* and *left* or *xmax* and *right*.
(8)                     Returns
(8)                     -------
(8)                     left, right : (float, float)
(12)                        The new x-axis limits in data coordinates.
(8)                     See Also
(8)                     --------
(8)                     get_xlim
(8)                     set_xbound, get_xbound
(8)                     invert_xaxis, xaxis_inverted
(8)                     Notes
(8)                     -----
(8)                     The *left* value may be greater than the *right* value, in which
(8)                     case the x-axis values will decrease from left to right.
(8)                     Examples
(8)                     --------
(8)                     >>> set_xlim(left, right)
(8)                     >>> set_xlim((left, right))
(8)                     >>> left, right = set_xlim(left, right)
(8)                     One limit may be left unchanged.
(8)                     >>> set_xlim(right=right_lim)
(8)                     Limits may be passed in reverse order to flip the direction of
(8)                     the x-axis. For example, suppose *x* represents the number of
(8)                     years before present. The x-axis limits might be set like the
(8)                     following so 5000 years ago is on the left of the plot and the
(8)                     present is on the right.
(8)                     >>> set_xlim(5000, 0)
(8)                     """
(8)                     if right is None and np.iterable(left):
(12)                        left, right = left
(8)                     if xmin is not None:
(12)                        if left is not None:
(16)                            raise TypeError("Cannot pass both 'left' and 'xmin'")
(12)                        left = xmin
(8)                     if xmax is not None:
(12)                        if right is not None:
(16)                            raise TypeError("Cannot pass both 'right' and 'xmax'")
(12)                        right = xmax
(8)                     return self.xaxis._set_lim(left, right, emit=emit, auto=auto)
(4)                 get_xscale = _axis_method_wrapper("xaxis", "get_scale")
(4)                 set_xscale = _axis_method_wrapper("xaxis", "_set_axes_scale")
(4)                 get_xticks = _axis_method_wrapper("xaxis", "get_ticklocs")
(4)                 set_xticks = _axis_method_wrapper("xaxis", "set_ticks",
(38)                                                  doc_sub={'set_ticks': 'set_xticks'})
(4)                 get_xmajorticklabels = _axis_method_wrapper("xaxis",
"get_majorticklabels")
(4)                 get_xminorticklabels = _axis_method_wrapper("xaxis",
"get_minorticklabels")
(4)                 get_xticklabels = _axis_method_wrapper("xaxis", "get_ticklabels")
(4)                 set_xticklabels = _axis_method_wrapper(
(8)                     "xaxis", "set_ticklabels",
(8)                     doc_sub={"Axis.set_ticks": "Axes.set_xticks"})
(4)                 def get_ylabel(self):
(8)                     """
(8)                     Get the ylabel text string.
(8)                     """
(8)                     label = self.yaxis.get_label()
(8)                     return label.get_text()
(4)                 def set_ylabel(self, ylabel, fontdict=None, labelpad=None, *,
(19)                                   loc=None, **kwargs):
```

```
(8)                 """
(8)                 Set the label for the y-axis.
(8)                 Parameters
(8)                 ----------
(8)                 ylabel : str
(12)                    The label text.
(8)                 labelpad : float, default: :rc:`axes.labelpad`
(12)                    Spacing in points from the Axes bounding box including ticks
(12)                    and tick labels.  If None, the previous value is left as is.
(8)                 loc : {'bottom', 'center', 'top'}, default: :rc:`yaxis.labellocation`
(12)                    The label position. This is a high-level alternative for passing
(12)                    parameters *y* and *horizontalalignment*.
(8)                 Other Parameters
(8)                 ----------------
(8)                 **kwargs : `~matplotlib.text.Text` properties
(12)                    `.Text` properties control the appearance of the label.
(8)                 See Also
(8)                 --------
(8)                 text : Documents the properties supported by `.Text`.
(8)                 """
(8)                 if labelpad is not None:
(12)                    self.yaxis.labelpad = labelpad
(8)                 protected_kw = ['y', 'horizontalalignment', 'ha']
(8)                 if {*kwargs} & {*protected_kw}:
(12)                    if loc is not None:
(16)                        raise TypeError(f"Specifying 'loc' is disallowed when any of "
(32)                                        f"its corresponding low level keyword "
(32)                                        f"arguments ({protected_kw}) are also "
(32)                                        f"supplied")
(8)                 else:
(12)                    loc = (loc if loc is not None
(19)                           else mpl.rcParams['yaxis.labellocation'])
(12)                    _api.check_in_list(('bottom', 'center', 'top'), loc=loc)
(12)                    y, ha = {
(16)                        'bottom': (0, 'left'),
(16)                        'center': (0.5, 'center'),
(16)                        'top': (1, 'right')
(12)                    }[loc]
(12)                    kwargs.update(y=y, horizontalalignment=ha)
(8)                 return self.yaxis.set_label_text(ylabel, fontdict, **kwargs)
(4)             def invert_yaxis(self):
(8)                 """
(8)                 Invert the y-axis.
(8)                 See Also
(8)                 --------
(8)                 yaxis_inverted
(8)                 get_ylim, set_ylim
(8)                 get_ybound, set_ybound
(8)                 """
(8)                 self.yaxis.set_inverted(not self.yaxis.get_inverted())
(4)             yaxis_inverted = _axis_method_wrapper("yaxis", "get_inverted")
(4)             def get_ybound(self):
(8)                 """
(8)                 Return the lower and upper y-axis bounds, in increasing order.
(8)                 See Also
(8)                 --------
(8)                 set_ybound
(8)                 get_ylim, set_ylim
(8)                 invert_yaxis, yaxis_inverted
(8)                 """
(8)                 bottom, top = self.get_ylim()
(8)                 if bottom < top:
(12)                    return bottom, top
(8)                 else:
(12)                    return top, bottom
(4)             def set_ybound(self, lower=None, upper=None):
(8)                 """
(8)                 Set the lower and upper numerical bounds of the y-axis.
(8)                 This method will honor axis inversion regardless of parameter order.
```

```
(8)                 It will not change the autoscaling setting (`.get_autoscaley_on()`).
(8)                 Parameters
(8)                 ----------
(8)                 lower, upper : float or None
(12)                    The lower and upper bounds. If *None*, the respective axis bound
(12)                    is not modified.
(9)                  .. ACCEPTS: (lower: float, upper: float)
(8)                 See Also
(8)                 --------
(8)                 get_ybound
(8)                 get_ylim, set_ylim
(8)                 invert_yaxis, yaxis_inverted
(8)                 """
(8)                 if upper is None and np.iterable(lower):
(12)                    lower, upper = lower
(8)                 old_lower, old_upper = self.get_ybound()
(8)                 if lower is None:
(12)                    lower = old_lower
(8)                 if upper is None:
(12)                    upper = old_upper
(8)                 self.set_ylim(sorted((lower, upper),
(29)                                  reverse=bool(self.yaxis_inverted())),
(22)                              auto=None)
(4)             def get_ylim(self):
(8)                 """
(8)                 Return the y-axis view limits.
(8)                 Returns
(8)                 -------
(8)                 bottom, top : (float, float)
(12)                    The current y-axis limits in data coordinates.
(8)                 See Also
(8)                 --------
(8)                 .Axes.set_ylim
(8)                 .Axes.set_ybound, .Axes.get_ybound
(8)                 .Axes.invert_yaxis, .Axes.yaxis_inverted
(8)                 Notes
(8)                 -----
(8)                 The y-axis may be inverted, in which case the *bottom* value
(8)                 will be greater than the *top* value.
(8)                 """
(8)                 return tuple(self.viewLim.intervaly)
(4)             def set_ylim(self, bottom=None, top=None, *, emit=True, auto=False,
(17)                         ymin=None, ymax=None):
(8)                 """
(8)                 Set the y-axis view limits.
(8)                 Parameters
(8)                 ----------
(8)                 bottom : float, optional
(12)                    The bottom ylim in data coordinates. Passing *None* leaves the
(12)                    limit unchanged.
(12)                    The bottom and top ylims may also be passed as the tuple
(12)                    (*bottom*, *top*) as the first positional argument (or as
(12)                    the *bottom* keyword argument).
(12)                    .. ACCEPTS: (bottom: float, top: float)
(8)                 top : float, optional
(12)                    The top ylim in data coordinates. Passing *None* leaves the
(12)                    limit unchanged.
(8)                 emit : bool, default: True
(12)                    Whether to notify observers of limit change.
(8)                 auto : bool or None, default: False
(12)                    Whether to turn on autoscaling of the y-axis. *True* turns on,
(12)                    *False* turns off, *None* leaves unchanged.
(8)                 ymin, ymax : float, optional
(12)                    They are equivalent to bottom and top respectively, and it is an
(12)                    error to pass both *ymin* and *bottom* or *ymax* and *top*.
(8)                 Returns
(8)                 -------
(8)                 bottom, top : (float, float)
(12)                    The new y-axis limits in data coordinates.
```

```
3126: (8)                    See Also
3127: (8)                    --------
3128: (8)                    get_ylim
3129: (8)                    set_ybound, get_ybound
3130: (8)                    invert_yaxis, yaxis_inverted
3131: (8)                    Notes
3132: (8)                    -----
3133: (8)                    The *bottom* value may be greater than the *top* value, in which
3134: (8)                    case the y-axis values will decrease from *bottom* to *top*.
3135: (8)                    Examples
3136: (8)                    --------
3137: (8)                    >>> set_ylim(bottom, top)
3138: (8)                    >>> set_ylim((bottom, top))
3139: (8)                    >>> bottom, top = set_ylim(bottom, top)
3140: (8)                    One limit may be left unchanged.
3141: (8)                    >>> set_ylim(top=top_lim)
3142: (8)                    Limits may be passed in reverse order to flip the direction of
3143: (8)                    the y-axis. For example, suppose ``y`` represents depth of the
3144: (8)                    ocean in m. The y-axis limits might be set like the following
3145: (8)                    so 5000 m depth is at the bottom of the plot and the surface,
3146: (8)                    0 m, is at the top.
3147: (8)                    >>> set_ylim(5000, 0)
3148: (8)                    """
3149: (8)                    if top is None and np.iterable(bottom):
3150: (12)                       bottom, top = bottom
3151: (8)                    if ymin is not None:
3152: (12)                       if bottom is not None:
3153: (16)                           raise TypeError("Cannot pass both 'bottom' and 'ymin'")
3154: (12)                       bottom = ymin
3155: (8)                    if ymax is not None:
3156: (12)                       if top is not None:
3157: (16)                           raise TypeError("Cannot pass both 'top' and 'ymax'")
3158: (12)                       top = ymax
3159: (8)                    return self.yaxis._set_lim(bottom, top, emit=emit, auto=auto)
3160: (4)                get_yscale = _axis_method_wrapper("yaxis", "get_scale")
3161: (4)                set_yscale = _axis_method_wrapper("yaxis", "_set_axes_scale")
3162: (4)                get_yticks = _axis_method_wrapper("yaxis", "get_ticklocs")
3163: (4)                set_yticks = _axis_method_wrapper("yaxis", "set_ticks",
3164: (38)                                                 doc_sub={'set_ticks': 'set_yticks'})
3165: (4)                get_ymajorticklabels = _axis_method_wrapper("yaxis",
"get_majorticklabels")
3166: (4)                get_yminorticklabels = _axis_method_wrapper("yaxis",
"get_minorticklabels")
3167: (4)                get_yticklabels = _axis_method_wrapper("yaxis", "get_ticklabels")
3168: (4)                set_yticklabels = _axis_method_wrapper(
3169: (8)                    "yaxis", "set_ticklabels",
3170: (8)                    doc_sub={"Axis.set_ticks": "Axes.set_yticks"})
3171: (4)                xaxis_date = _axis_method_wrapper("xaxis", "axis_date")
3172: (4)                yaxis_date = _axis_method_wrapper("yaxis", "axis_date")
3173: (4)                def format_xdata(self, x):
3174: (8)                    """
3175: (8)                    Return *x* formatted as an x-value.
3176: (8)                    This function will use the `.fmt_xdata` attribute if it is not None,
3177: (8)                    else will fall back on the xaxis major formatter.
3178: (8)                    """
3179: (8)                    return (self.fmt_xdata if self.fmt_xdata is not None
3180: (16)                           else self.xaxis.get_major_formatter().format_data_short)(x)
3181: (4)                def format_ydata(self, y):
3182: (8)                    """
3183: (8)                    Return *y* formatted as a y-value.
3184: (8)                    This function will use the `.fmt_ydata` attribute if it is not None,
3185: (8)                    else will fall back on the yaxis major formatter.
3186: (8)                    """
3187: (8)                    return (self.fmt_ydata if self.fmt_ydata is not None
3188: (16)                           else self.yaxis.get_major_formatter().format_data_short)(y)
3189: (4)                def format_coord(self, x, y):
3190: (8)                    """Return a format string formatting the *x*, *y* coordinates."""
3191: (8)                    twins = self._twinned_axes.get_siblings(self)
3192: (8)                    if len(twins) == 1:
```

```
(12)                        return "(x, y) = ({}, {})".format(
(16)                            "???" if x is None else self.format_xdata(x),
(16)                            "???" if y is None else self.format_ydata(y))
(8)                     screen_xy = self.transData.transform((x, y))
(8)                     xy_strs = []
(8)                     for ax in sorted(twins, key=attrgetter("zorder")):
(12)                        data_x, data_y = ax.transData.inverted().transform(screen_xy)
(12)                        xy_strs.append(
(16)                            "({}, {})".format(ax.format_xdata(data_x),
ax.format_ydata(data_y)))
(8)                     return "(x, y) = {}".format(" | ".join(xy_strs))
(4)                 def minorticks_on(self):
(8)                     """
(8)                     Display minor ticks on the Axes.
(8)                     Displaying minor ticks may reduce performance; you may turn them off
(8)                     using `minorticks_off()` if drawing speed is a problem.
(8)                     """
(8)                     self.xaxis.minorticks_on()
(8)                     self.yaxis.minorticks_on()
(4)                 def minorticks_off(self):
(8)                     """Remove minor ticks from the Axes."""
(8)                     self.xaxis.minorticks_off()
(8)                     self.yaxis.minorticks_off()
(4)                 def can_zoom(self):
(8)                     """
(8)                     Return whether this Axes supports the zoom box button functionality.
(8)                     """
(8)                     return True
(4)                 def can_pan(self):
(8)                     """
(8)                     Return whether this Axes supports any pan/zoom button functionality.
(8)                     """
(8)                     return True
(4)                 def get_navigate(self):
(8)                     """
(8)                     Get whether the Axes responds to navigation commands.
(8)                     """
(8)                     return self._navigate
(4)                 def set_navigate(self, b):
(8)                     """
(8)                     Set whether the Axes responds to navigation toolbar commands.
(8)                     Parameters
(8)                     ----------
(8)                     b : bool
(8)                     See Also
(8)                     --------
(8)                     matplotlib.axes.Axes.set_forward_navigation_events
(8)                     """
(8)                     self._navigate = b
(4)                 def get_navigate_mode(self):
(8)                     """
(8)                     Get the navigation toolbar button status: 'PAN', 'ZOOM', or None.
(8)                     """
(8)                     return self._navigate_mode
(4)                 def set_navigate_mode(self, b):
(8)                     """
(8)                     Set the navigation toolbar button status.
(8)                     .. warning::
(12)                        This is not a user-API function.
(8)                     """
(8)                     self._navigate_mode = b
(4)                 def _get_view(self):
(8)                     """
(8)                     Save information required to reproduce the current view.
(8)                     This method is called before a view is changed, such as during a pan
or zoom
(8)                     initiated by the user.  It returns an opaque object that describes the
current
(8)                     view, in a format compatible with :meth:`_set_view`.
```

```
3259: (8)                    The default implementation saves the view limits and autoscaling
state.
3260: (8)                    Subclasses may override this as needed, as long as :meth:`_set_view`
is also
3261: (8)                    adjusted accordingly.
3262: (8)                    """
3263: (8)                    return {
3264: (12)                       "xlim": self.get_xlim(), "autoscalex_on":
self.get_autoscalex_on(),
3265: (12)                       "ylim": self.get_ylim(), "autoscaley_on":
self.get_autoscaley_on(),
3266: (8)                    }
3267: (4)                def _set_view(self, view):
3268: (8)                    """
3269: (8)                    Apply a previously saved view.
3270: (8)                    This method is called when restoring a view (with the return value of
3271: (8)                    :meth:`_get_view` as argument), such as with the navigation buttons.
3272: (8)                    Subclasses that override :meth:`_get_view` also need to override this
method
3273: (8)                    accordingly.
3274: (8)                    """
3275: (8)                    self.set(**view)
3276: (4)                def _prepare_view_from_bbox(self, bbox, direction='in',
3277: (32)                                           mode=None, twinx=False, twiny=False):
3278: (8)                    """
3279: (8)                    Helper function to prepare the new bounds from a bbox.
3280: (8)                    This helper function returns the new x and y bounds from the zoom
3281: (8)                    bbox. This a convenience method to abstract the bbox logic
3282: (8)                    out of the base setter.
3283: (8)                    """
3284: (8)                    if len(bbox) == 3:
3285: (12)                       xp, yp, scl = bbox  # Zooming code
3286: (12)                       if scl == 0:  # Should not happen
3287: (16)                           scl = 1.
3288: (12)                       if scl > 1:
3289: (16)                           direction = 'in'
3290: (12)                       else:
3291: (16)                           direction = 'out'
3292: (16)                           scl = 1/scl
3293: (12)                       (xmin, ymin), (xmax, ymax) = self.transData.transform(
3294: (16)                           np.transpose([self.get_xlim(), self.get_ylim()]))
3295: (12)                       xwidth = xmax - xmin
3296: (12)                       ywidth = ymax - ymin
3297: (12)                       xcen = (xmax + xmin)*.5
3298: (12)                       ycen = (ymax + ymin)*.5
3299: (12)                       xzc = (xp*(scl - 1) + xcen)/scl
3300: (12)                       yzc = (yp*(scl - 1) + ycen)/scl
3301: (12)                       bbox = [xzc - xwidth/2./scl, yzc - ywidth/2./scl,
3302: (20)                               xzc + xwidth/2./scl, yzc + ywidth/2./scl]
3303: (8)                    elif len(bbox) != 4:
3304: (12)                       _api.warn_external(
3305: (16)                           "Warning in _set_view_from_bbox: bounding box is not a tuple "
3306: (16)                           "of length 3 or 4. Ignoring the view change.")
3307: (12)                       return
3308: (8)                    xmin0, xmax0 = self.get_xbound()
3309: (8)                    ymin0, ymax0 = self.get_ybound()
3310: (8)                    startx, starty, stopx, stopy = bbox
3311: (8)                    (startx, starty), (stopx, stopy) =
self.transData.inverted().transform(
3312: (12)                       [(startx, starty), (stopx, stopy)])
3313: (8)                    xmin, xmax = np.clip(sorted([startx, stopx]), xmin0, xmax0)
3314: (8)                    ymin, ymax = np.clip(sorted([starty, stopy]), ymin0, ymax0)
3315: (8)                    if twinx or mode == "y":
3316: (12)                       xmin, xmax = xmin0, xmax0
3317: (8)                    if twiny or mode == "x":
3318: (12)                       ymin, ymax = ymin0, ymax0
3319: (8)                    if direction == "in":
3320: (12)                       new_xbound = xmin, xmax
3321: (12)                       new_ybound = ymin, ymax
```

```
(8)                 elif direction == "out":
(12)                    x_trf = self.xaxis.get_transform()
(12)                    sxmin0, sxmax0, sxmin, sxmax = x_trf.transform(
(16)                        [xmin0, xmax0, xmin, xmax])  # To screen space.
(12)                    factor = (sxmax0 - sxmin0) / (sxmax - sxmin)  # Unzoom factor.
(12)                    sxmin1 = sxmin0 - factor * (sxmin - sxmin0)
(12)                    sxmax1 = sxmax0 + factor * (sxmax0 - sxmax)
(12)                    new_xbound = x_trf.inverted().transform([sxmin1, sxmax1])
(12)                    y_trf = self.yaxis.get_transform()
(12)                    symin0, symax0, symin, symax = y_trf.transform(
(16)                        [ymin0, ymax0, ymin, ymax])
(12)                    factor = (symax0 - symin0) / (symax - symin)
(12)                    symin1 = symin0 - factor * (symin - symin0)
(12)                    symax1 = symax0 + factor * (symax0 - symax)
(12)                    new_ybound = y_trf.inverted().transform([symin1, symax1])
(8)                 return new_xbound, new_ybound
(4)             def _set_view_from_bbox(self, bbox, direction='in',
(28)                                    mode=None, twinx=False, twiny=False):
(8)                 """
(8)                 Update view from a selection bbox.
(8)                 .. note::
(12)                    Intended to be overridden by new projection types, but if not, the
(12)                    default implementation sets the view limits to the bbox directly.
(8)                 Parameters
(8)                 ----------
(8)                 bbox : 4-tuple or 3 tuple
(12)                    * If bbox is a 4 tuple, it is the selected bounding box limits,
(14)                      in *display* coordinates.
(12)                    * If bbox is a 3 tuple, it is an (xp, yp, scl) triple, where
(14)                      (xp, yp) is the center of zooming and scl the scale factor to
(14)                      zoom by.
(8)                 direction : str
(12)                    The direction to apply the bounding box.
(16)                        * `'in'` - The bounding box describes the view directly, i.e.,
(27)                                   it zooms in.
(16)                        * `'out'` - The bounding box describes the size to make the
(28)                                    existing view, i.e., it zooms out.
(8)                 mode : str or None
(12)                    The selection mode, whether to apply the bounding box in only the
(12)                    `'x'` direction, `'y'` direction or both (`None`).
(8)                 twinx : bool
(12)                    Whether this axis is twinned in the *x*-direction.
(8)                 twiny : bool
(12)                    Whether this axis is twinned in the *y*-direction.
(8)                 """
(8)                 new_xbound, new_ybound = self._prepare_view_from_bbox(
(12)                    bbox, direction=direction, mode=mode, twinx=twinx, twiny=twiny)
(8)                 if not twinx and mode != "y":
(12)                    self.set_xbound(new_xbound)
(12)                    self.set_autoscalex_on(False)
(8)                 if not twiny and mode != "x":
(12)                    self.set_ybound(new_ybound)
(12)                    self.set_autoscaley_on(False)
(4)             def start_pan(self, x, y, button):
(8)                 """
(8)                 Called when a pan operation has started.
(8)                 Parameters
(8)                 ----------
(8)                 x, y : float
(12)                    The mouse coordinates in display coords.
(8)                 button : `.MouseButton`
(12)                    The pressed mouse button.
(8)                 Notes
(8)                 -----
(8)                 This is intended to be overridden by new projection types.
(8)                 """
(8)                 self._pan_start = types.SimpleNamespace(
(12)                    lim=self.viewLim.frozen(),
(12)                    trans=self.transData.frozen(),
```

```
(12)                trans_inverse=self.transData.inverted().frozen(),
(12)                bbox=self.bbox.frozen(),
(12)                x=x,
(12)                y=y)
(4)         def end_pan(self):
(8)             """
(8)             Called when a pan operation completes (when the mouse button is up.)
(8)             Notes
(8)             -----
(8)             This is intended to be overridden by new projection types.
(8)             """
(8)             del self._pan_start
(4)         def _get_pan_points(self, button, key, x, y):
(8)             """
(8)             Helper function to return the new points after a pan.
(8)             This helper function returns the points on the axis after a pan has
(8)             occurred. This is a convenience method to abstract the pan logic
(8)             out of the base setter.
(8)             """
(8)             def format_deltas(key, dx, dy):
(12)                if key == 'control':
(16)                    if abs(dx) > abs(dy):
(20)                        dy = dx
(16)                    else:
(20)                        dx = dy
(12)                elif key == 'x':
(16)                    dy = 0
(12)                elif key == 'y':
(16)                    dx = 0
(12)                elif key == 'shift':
(16)                    if 2 * abs(dx) < abs(dy):
(20)                        dx = 0
(16)                    elif 2 * abs(dy) < abs(dx):
(20)                        dy = 0
(16)                    elif abs(dx) > abs(dy):
(20)                        dy = dy / abs(dy) * abs(dx)
(16)                    else:
(20)                        dx = dx / abs(dx) * abs(dy)
(12)                return dx, dy
(8)             p = self._pan_start
(8)             dx = x - p.x
(8)             dy = y - p.y
(8)             if dx == dy == 0:
(12)                return
(8)             if button == 1:
(12)                dx, dy = format_deltas(key, dx, dy)
(12)                result = p.bbox.translated(-dx, -dy).transformed(p.trans_inverse)
(8)             elif button == 3:
(12)                try:
(16)                    dx = -dx / self.bbox.width
(16)                    dy = -dy / self.bbox.height
(16)                    dx, dy = format_deltas(key, dx, dy)
(16)                    if self.get_aspect() != 'auto':
(20)                        dx = dy = 0.5 * (dx + dy)
(16)                    alpha = np.power(10.0, (dx, dy))
(16)                    start = np.array([p.x, p.y])
(16)                    oldpoints = p.lim.transformed(p.trans)
(16)                    newpoints = start + alpha * (oldpoints - start)
(16)                    result = (mtransforms.Bbox(newpoints)
(26)                              .transformed(p.trans_inverse))
(12)                except OverflowError:
(16)                    _api.warn_external('Overflow while panning')
(16)                    return
(8)             else:
(12)                return
(8)             valid = np.isfinite(result.transformed(p.trans))
(8)             points = result.get_points().astype(object)
(8)             points[~valid] = None
(8)             return points
```

```
(4)     def drag_pan(self, button, key, x, y):
(8)         """
(8)         Called when the mouse moves during a pan operation.
(8)         Parameters
(8)         ----------
(8)         button : `.MouseButton`
(12)            The pressed mouse button.
(8)         key : str or None
(12)            The pressed key, if any.
(8)         x, y : float
(12)            The mouse coordinates in display coords.
(8)         Notes
(8)         -----
(8)         This is intended to be overridden by new projection types.
(8)         """
(8)         points = self._get_pan_points(button, key, x, y)
(8)         if points is not None:
(12)            self.set_xlim(points[:, 0])
(12)            self.set_ylim(points[:, 1])
(4)     def get_children(self):
(8)         return [
(12)            *self._children,
(12)            *self.spines.values(),
(12)            *self._axis_map.values(),
(12)            self.title, self._left_title, self._right_title,
(12)            *self.child_axes,
(12)            *([self.legend_] if self.legend_ is not None else []),
(12)            self.patch,
(8)         ]
(4)     def contains(self, mouseevent):
(8)         return self.patch.contains(mouseevent)
(4)     def contains_point(self, point):
(8)         """
(8)         Return whether *point* (pair of pixel coordinates) is inside the Axes
(8)         patch.
(8)         """
(8)         return self.patch.contains_point(point, radius=1.0)
(4)     def get_default_bbox_extra_artists(self):
(8)         """
(8)         Return a default list of artists that are used for the bounding box
(8)         calculation.
(8)         Artists are excluded either by not being visible or
(8)         ``artist.set_in_layout(False)``.
(8)         """
(8)         artists = self.get_children()
(8)         for axis in self._axis_map.values():
(12)            artists.remove(axis)
(8)         if not (self.axison and self._frameon):
(12)            for spine in self.spines.values():
(16)                artists.remove(spine)
(8)         artists.remove(self.title)
(8)         artists.remove(self._left_title)
(8)         artists.remove(self._right_title)
(8)         noclip = (_AxesBase, maxis.Axis,
(18)                  offsetbox.AnnotationBbox, offsetbox.OffsetBox)
(8)         return [a for a in artists if a.get_visible() and a.get_in_layout()
(16)                and (isinstance(a, noclip) or not a._fully_clipped_to_axes())]
(4)     @_api.make_keyword_only("3.8", "call_axes_locator")
(4)     def get_tightbbox(self, renderer=None, call_axes_locator=True,
(22)                      bbox_extra_artists=None, *, for_layout_only=False):
(8)         """
(8)         Return the tight bounding box of the Axes, including axis and their
(8)         decorators (xlabel, title, etc).
(8)         Artists that have ``artist.set_in_layout(False)`` are not included
(8)         in the bbox.
(8)         Parameters
(8)         ----------
(8)         renderer : `.RendererBase` subclass
(12)            renderer that will be used to draw the figures (i.e.
```

```
(12)                    ``fig.canvas.get_renderer()``)
(8)                 bbox_extra_artists : list of `.Artist` or ``None``
(12)                    List of artists to include in the tight bounding box.  If
(12)                    ``None`` (default), then all artist children of the Axes are
(12)                    included in the tight bounding box.
(8)                 call_axes_locator : bool, default: True
(12)                    If *call_axes_locator* is ``False``, it does not call the
(12)                    ``_axes_locator`` attribute, which is necessary to get the correct
(12)                    bounding box. ``call_axes_locator=False`` can be used if the
(12)                    caller is only interested in the relative size of the tightbbox
(12)                    compared to the Axes bbox.
(8)                 for_layout_only : default: False
(12)                    The bounding box will *not* include the x-extent of the title and
(12)                    the xlabel, or the y-extent of the ylabel.
(8)                 Returns
(8)                 -------
(8)                 `.BboxBase`
(12)                    Bounding box in figure pixel coordinates.
(8)                 See Also
(8)                 --------
(8)                 matplotlib.axes.Axes.get_window_extent
(8)                 matplotlib.axis.Axis.get_tightbbox
(8)                 matplotlib.spines.Spine.get_window_extent
(8)                 """
(8)                 bb = []
(8)                 if renderer is None:
(12)                    renderer = self.figure._get_renderer()
(8)                 if not self.get_visible():
(12)                    return None
(8)                 locator = self.get_axes_locator()
(8)                 self.apply_aspect(
(12)                    locator(self, renderer) if locator and call_axes_locator else
None)
(8)                 for axis in self._axis_map.values():
(12)                    if self.axison and axis.get_visible():
(16)                        ba = martist._get_tightbbox_for_layout_only(axis, renderer)
(16)                        if ba:
(20)                            bb.append(ba)
(8)                 self._update_title_position(renderer)
(8)                 axbbox = self.get_window_extent(renderer)
(8)                 bb.append(axbbox)
(8)                 for title in [self.title, self._left_title, self._right_title]:
(12)                    if title.get_visible():
(16)                        bt = title.get_window_extent(renderer)
(16)                        if for_layout_only and bt.width > 0:
(20)                            bt.x0 = (bt.x0 + bt.x1) / 2 - 0.5
(20)                            bt.x1 = bt.x0 + 1.0
(16)                        bb.append(bt)
(8)                 bbox_artists = bbox_extra_artists
(8)                 if bbox_artists is None:
(12)                    bbox_artists = self.get_default_bbox_extra_artists()
(8)                 for a in bbox_artists:
(12)                    bbox = a.get_tightbbox(renderer)
(12)                    if (bbox is not None
(20)                            and 0 < bbox.width < np.inf
(20)                            and 0 < bbox.height < np.inf):
(16)                        bb.append(bbox)
(8)                 return mtransforms.Bbox.union(
(12)                    [b for b in bb if b.width != 0 or b.height != 0])
(4)             def _make_twin_axes(self, *args, **kwargs):
(8)                 """Make a twinx Axes of self. This is used for twinx and twiny."""
(8)                 if 'sharex' in kwargs and 'sharey' in kwargs:
(12)                    if kwargs["sharex"] is not self and kwargs["sharey"] is not self:
(16)                        raise ValueError("Twinned Axes may share only one axis")
(8)                 ss = self.get_subplotspec()
(8)                 if ss:
(12)                    twin = self.figure.add_subplot(ss, *args, **kwargs)
(8)                 else:
(12)                    twin = self.figure.add_axes(
```

```
(16)                            self.get_position(True), *args, **kwargs,
(16)                            axes_locator=_TransformedBoundsLocator(
(20)                                [0, 0, 1, 1], self.transAxes))
(8)                     self.set_adjustable('datalim')
(8)                     twin.set_adjustable('datalim')
(8)                     twin.set_zorder(self.zorder)
(8)                     self._twinned_axes.join(self, twin)
(8)                     return twin
(4)                 def twinx(self):
(8)                     """
(8)                     Create a twin Axes sharing the xaxis.
(8)                     Create a new Axes with an invisible x-axis and an independent
(8)                     y-axis positioned opposite to the original one (i.e. at right). The
(8)                     x-axis autoscale setting will be inherited from the original
(8)                     Axes.  To ensure that the tick marks of both y-axes align, see
(8)                     `~matplotlib.ticker.LinearLocator`.
(8)                     Returns
(8)                     -------
(8)                     Axes
(12)                        The newly created Axes instance
(8)                     Notes
(8)                     -----
(8)                     For those who are 'picking' artists while using twinx, pick
(8)                     events are only called for the artists in the top-most Axes.
(8)                     """
(8)                     ax2 = self._make_twin_axes(sharex=self)
(8)                     ax2.yaxis.tick_right()
(8)                     ax2.yaxis.set_label_position('right')
(8)                     ax2.yaxis.set_offset_position('right')
(8)                     ax2.set_autoscalex_on(self.get_autoscalex_on())
(8)                     self.yaxis.tick_left()
(8)                     ax2.xaxis.set_visible(False)
(8)                     ax2.patch.set_visible(False)
(8)                     ax2.xaxis.units = self.xaxis.units
(8)                     return ax2
(4)                 def twiny(self):
(8)                     """
(8)                     Create a twin Axes sharing the yaxis.
(8)                     Create a new Axes with an invisible y-axis and an independent
(8)                     x-axis positioned opposite to the original one (i.e. at top). The
(8)                     y-axis autoscale setting will be inherited from the original Axes.
(8)                     To ensure that the tick marks of both x-axes align, see
(8)                     `~matplotlib.ticker.LinearLocator`.
(8)                     Returns
(8)                     -------
(8)                     Axes
(12)                        The newly created Axes instance
(8)                     Notes
(8)                     -----
(8)                     For those who are 'picking' artists while using twiny, pick
(8)                     events are only called for the artists in the top-most Axes.
(8)                     """
(8)                     ax2 = self._make_twin_axes(sharey=self)
(8)                     ax2.xaxis.tick_top()
(8)                     ax2.xaxis.set_label_position('top')
(8)                     ax2.set_autoscaley_on(self.get_autoscaley_on())
(8)                     self.xaxis.tick_bottom()
(8)                     ax2.yaxis.set_visible(False)
(8)                     ax2.patch.set_visible(False)
(8)                     ax2.yaxis.units = self.yaxis.units
(8)                     return ax2
(4)                 def get_shared_x_axes(self):
(8)                     """Return an immutable view on the shared x-axes Grouper."""
(8)                     return cbook.GrouperView(self._shared_axes["x"])
(4)                 def get_shared_y_axes(self):
(8)                     """Return an immutable view on the shared y-axes Grouper."""
(8)                     return cbook.GrouperView(self._shared_axes["y"])
(4)                 def label_outer(self, remove_inner_ticks=False):
(8)                     """
```

```
(8)                 Only show "outer" labels and tick labels.
(8)                 x-labels are only kept for subplots on the last row (or first row, if
(8)                 labels are on the top side); y-labels only for subplots on the first
(8)                 column (or last column, if labels are on the right side).
(8)                 Parameters
(8)                 ----------
(8)                 remove_inner_ticks : bool, default: False
(12)                    If True, remove the inner ticks as well (not only tick labels).
(12)                    .. versionadded:: 3.8
(8)                 """
(8)                 self._label_outer_xaxis(skip_non_rectangular_axes=False,
(32)                                        remove_inner_ticks=remove_inner_ticks)
(8)                 self._label_outer_yaxis(skip_non_rectangular_axes=False,
(32)                                        remove_inner_ticks=remove_inner_ticks)
(4)             def _label_outer_xaxis(self, *, skip_non_rectangular_axes,
(27)                                   remove_inner_ticks=False):
(8)                 if skip_non_rectangular_axes and not isinstance(self.patch,
(56)
mpl.patches.Rectangle):
(12)                    return
(8)                 ss = self.get_subplotspec()
(8)                 if not ss:
(12)                    return
(8)                 label_position = self.xaxis.get_label_position()
(8)                 if not ss.is_first_row():  # Remove top label/ticklabels/offsettext.
(12)                    if label_position == "top":
(16)                        self.set_xlabel("")
(12)                    top_kw = {'top': False} if remove_inner_ticks else {}
(12)                    self.xaxis.set_tick_params(
(16)                        which="both", labeltop=False, **top_kw)
(12)                    if self.xaxis.offsetText.get_position()[1] == 1:
(16)                        self.xaxis.offsetText.set_visible(False)
(8)                 if not ss.is_last_row():  # Remove bottom label/ticklabels/offsettext.
(12)                    if label_position == "bottom":
(16)                        self.set_xlabel("")
(12)                    bottom_kw = {'bottom': False} if remove_inner_ticks else {}
(12)                    self.xaxis.set_tick_params(
(16)                        which="both", labelbottom=False, **bottom_kw)
(12)                    if self.xaxis.offsetText.get_position()[1] == 0:
(16)                        self.xaxis.offsetText.set_visible(False)
(4)             def _label_outer_yaxis(self, *, skip_non_rectangular_axes,
(27)                                   remove_inner_ticks=False):
(8)                 if skip_non_rectangular_axes and not isinstance(self.patch,
(56)
mpl.patches.Rectangle):
(12)                    return
(8)                 ss = self.get_subplotspec()
(8)                 if not ss:
(12)                    return
(8)                 label_position = self.yaxis.get_label_position()
(8)                 if not ss.is_first_col():  # Remove left label/ticklabels/offsettext.
(12)                    if label_position == "left":
(16)                        self.set_ylabel("")
(12)                    left_kw = {'left': False} if remove_inner_ticks else {}
(12)                    self.yaxis.set_tick_params(
(16)                        which="both", labelleft=False, **left_kw)
(12)                    if self.yaxis.offsetText.get_position()[0] == 0:
(16)                        self.yaxis.offsetText.set_visible(False)
(8)                 if not ss.is_last_col():  # Remove right label/ticklabels/offsettext.
(12)                    if label_position == "right":
(16)                        self.set_ylabel("")
(12)                    right_kw = {'right': False} if remove_inner_ticks else {}
(12)                    self.yaxis.set_tick_params(
(16)                        which="both", labelright=False, **right_kw)
(12)                    if self.yaxis.offsetText.get_position()[0] == 1:
(16)                        self.yaxis.offsetText.set_visible(False)
(4)             def set_forward_navigation_events(self, forward):
(8)                 """
(8)                 Set how pan/zoom events are forwarded to Axes below this one.
```

```
3733: (8)                   Parameters
3734: (8)                   ----------
3735: (8)                   forward : bool or "auto"
3736: (12)                      Possible values:
3737: (12)                      - True: Forward events to other axes with lower or equal zorder.
3738: (12)                      - False: Events are only executed on this axes.
3739: (12)                      - "auto": Default behaviour (*True* for axes with an invisible
3740: (14)                        patch and *False* otherwise)
3741: (8)                   See Also
3742: (8)                   --------
3743: (8)                   matplotlib.axes.Axes.set_navigate
3744: (8)                   """
3745: (8)                   self._forward_navigation_events = forward
3746: (4)               def get_forward_navigation_events(self):
3747: (8)                   """Get how pan/zoom events are forwarded to Axes below this one."""
3748: (8)                   return self._forward_navigation_events
3749: (0)           def _draw_rasterized(figure, artists, renderer):
3750: (4)               """
3751: (4)               A helper function for rasterizing the list of artists.
3752: (4)               The bookkeeping to track if we are or are not in rasterizing mode
3753: (4)               with the mixed-mode backends is relatively complicated and is now
3754: (4)               handled in the matplotlib.artist.allow_rasterization decorator.
3755: (4)               This helper defines the absolute minimum methods and attributes on a
3756: (4)               shim class to be compatible with that decorator and then uses it to
3757: (4)               rasterize the list of artists.
3758: (4)               This is maybe too-clever, but allows us to reuse the same code that is
3759: (4)               used on normal artists to participate in the "are we rasterizing"
3760: (4)               accounting.
3761: (4)               Please do not use this outside of the "rasterize below a given zorder"
3762: (4)               functionality of Axes.
3763: (4)               Parameters
3764: (4)               ----------
3765: (4)               figure : matplotlib.figure.Figure
3766: (8)                   The figure all of the artists belong to (not checked).  We need this
3767: (8)                   because we can at the figure level suppress composition and insert
each
3768: (8)                   rasterized artist as its own image.
3769: (4)               artists : List[matplotlib.artist.Artist]
3770: (8)                   The list of Artists to be rasterized.  These are assumed to all
3771: (8)                   be in the same Figure.
3772: (4)               renderer : matplotlib.backendbases.RendererBase
3773: (8)                   The currently active renderer
3774: (4)               Returns
3775: (4)               -------
3776: (4)               None
3777: (4)               """
3778: (4)               class _MinimalArtist:
3779: (8)                   def get_rasterized(self):
3780: (12)                      return True
3781: (8)                   def get_agg_filter(self):
3782: (12)                      return None
3783: (8)                   def __init__(self, figure, artists):
3784: (12)                      self.figure = figure
3785: (12)                      self.artists = artists
3786: (8)                   @martist.allow_rasterization
3787: (8)                   def draw(self, renderer):
3788: (12)                      for a in self.artists:
3789: (16)                          a.draw(renderer)
3790: (4)               return _MinimalArtist(figure, artists).draw(renderer)
```

File 77 - _secondary_axes.py:

```
1: (0)              import numbers
2: (0)              import numpy as np
3: (0)              from matplotlib import _api, _docstring, transforms
4: (0)              import matplotlib.ticker as mticker
5: (0)              from matplotlib.axes._base import _AxesBase, _TransformedBoundsLocator
```

```
6: (0)              from matplotlib.axis import Axis
7: (0)              from matplotlib.transforms import Transform
8: (0)              class SecondaryAxis(_AxesBase):
9: (4)                  """
10: (4)                 General class to hold a Secondary_X/Yaxis.
11: (4)                 """
12: (4)                 def __init__(self, parent, orientation, location, functions,
transform=None,
13: (17)                            **kwargs):
14: (8)                     """
15: (8)                     See `.secondary_xaxis` and `.secondary_yaxis` for the doc string.
16: (8)                     While there is no need for this to be private, it should really be
17: (8)                     called by those higher level functions.
18: (8)                     """
19: (8)                     _api.check_in_list(["x", "y"], orientation=orientation)
20: (8)                     self._functions = functions
21: (8)                     self._parent = parent
22: (8)                     self._orientation = orientation
23: (8)                     self._ticks_set = False
24: (8)                     if self._orientation == 'x':
25: (12)                        super().__init__(self._parent.figure, [0, 1., 1, 0.0001],
**kwargs)
26: (12)                        self._axis = self.xaxis
27: (12)                        self._locstrings = ['top', 'bottom']
28: (12)                        self._otherstrings = ['left', 'right']
29: (8)                     else:  # 'y'
30: (12)                        super().__init__(self._parent.figure, [0, 1., 0.0001, 1],
**kwargs)
31: (12)                        self._axis = self.yaxis
32: (12)                        self._locstrings = ['right', 'left']
33: (12)                        self._otherstrings = ['top', 'bottom']
34: (8)                     self._parentscale = None
35: (8)                     self.set_location(location, transform)
36: (8)                     self.set_functions(functions)
37: (8)                     otheraxis = self.yaxis if self._orientation == 'x' else self.xaxis
38: (8)                     otheraxis.set_major_locator(mticker.NullLocator())
39: (8)                     otheraxis.set_ticks_position('none')
40: (8)                     self.spines[self._otherstrings].set_visible(False)
41: (8)                     self.spines[self._locstrings].set_visible(True)
42: (8)                     if self._pos < 0.5:
43: (12)                        self._locstrings = self._locstrings[::-1]
44: (8)                     self.set_alignment(self._locstrings[0])
45: (4)                 def set_alignment(self, align):
46: (8)                     """
47: (8)                     Set if axes spine and labels are drawn at top or bottom (or
left/right)
48: (8)                     of the Axes.
49: (8)                     Parameters
50: (8)                     ----------
51: (8)                     align : {'top', 'bottom', 'left', 'right'}
52: (12)                        Either 'top' or 'bottom' for orientation='x' or
53: (12)                        'left' or 'right' for orientation='y' axis.
54: (8)                     """
55: (8)                     _api.check_in_list(self._locstrings, align=align)
56: (8)                     if align == self._locstrings[1]:  # Need to change the orientation.
57: (12)                        self._locstrings = self._locstrings[::-1]
58: (8)                     self.spines[self._locstrings[0]].set_visible(True)
59: (8)                     self.spines[self._locstrings[1]].set_visible(False)
60: (8)                     self._axis.set_ticks_position(align)
61: (8)                     self._axis.set_label_position(align)
62: (4)                 def set_location(self, location, transform=None):
63: (8)                     """
64: (8)                     Set the vertical or horizontal location of the axes in
65: (8)                     parent-normalized coordinates.
66: (8)                     Parameters
67: (8)                     ----------
68: (8)                     location : {'top', 'bottom', 'left', 'right'} or float
69: (12)                        The position to put the secondary axis.  Strings can be 'top' or
70: (12)                        'bottom' for orientation='x' and 'right' or 'left' for
```

```
 71: (12)                        orientation='y'. A float indicates the relative position on the
 72: (12)                        parent Axes to put the new Axes, 0.0 being the bottom (or left)
 73: (12)                        and 1.0 being the top (or right).
 74: (8)                     transform : `.Transform`, optional
 75: (12)                        Transform for the location to use. Defaults to
 76: (12)                        the parent's ``transAxes``, so locations are normally relative to
 77: (12)                        the parent axes.
 78: (12)                        .. versionadded:: 3.9
 79: (8)                     """
 80: (8)                     _api.check_isinstance((transforms.Transform, None),
transform=transform)
 81: (8)                     if isinstance(location, str):
 82: (12)                        _api.check_in_list(self._locstrings, location=location)
 83: (12)                        self._pos = 1. if location in ('top', 'right') else 0.
 84: (8)                     elif isinstance(location, numbers.Real):
 85: (12)                        self._pos = location
 86: (8)                     else:
 87: (12)                        raise ValueError(
 88: (16)                            f"location must be {self._locstrings[0]!r}, "
 89: (16)                            f"{self._locstrings[1]!r}, or a float, not {location!r}")
 90: (8)                     self._loc = location
 91: (8)                     if self._orientation == 'x':
 92: (12)                        bounds = [0, self._pos, 1., 1e-10]
 93: (12)                        if transform is not None:
 94: (16)                            transform = transforms.blended_transform_factory(
 95: (20)                                self._parent.transAxes, transform)
 96: (8)                     else:  # 'y'
 97: (12)                        bounds = [self._pos, 0, 1e-10, 1]
 98: (12)                        if transform is not None:
 99: (16)                            transform = transforms.blended_transform_factory(
100: (20)                                transform, self._parent.transAxes)  # Use provided x axis
101: (8)                     if transform is None:
102: (12)                        transform = self._parent.transAxes
103: (8)                     self.set_axes_locator(_TransformedBoundsLocator(bounds, transform))
104: (4)                 def apply_aspect(self, position=None):
105: (8)                     self._set_lims()
106: (8)                     super().apply_aspect(position)
107: (4)                 @_docstring.copy(Axis.set_ticks)
108: (4)                 def set_ticks(self, ticks, labels=None, *, minor=False, **kwargs):
109: (8)                     ret = self._axis.set_ticks(ticks, labels, minor=minor, **kwargs)
110: (8)                     self.stale = True
111: (8)                     self._ticks_set = True
112: (8)                     return ret
113: (4)                 def set_functions(self, functions):
114: (8)                     """
115: (8)                     Set how the secondary axis converts limits from the parent Axes.
116: (8)                     Parameters
117: (8)                     ----------
118: (8)                     functions : 2-tuple of func, or `Transform` with an inverse.
119: (12)                        Transform between the parent axis values and the secondary axis
120: (12)                        values.
121: (12)                        If supplied as a 2-tuple of functions, the first function is
122: (12)                        the forward transform function and the second is the inverse
123: (12)                        transform.
124: (12)                        If a transform is supplied, then the transform must have an
125: (12)                        inverse.
126: (8)                     """
127: (8)                     if (isinstance(functions, tuple) and len(functions) == 2 and
128: (16)                            callable(functions[0]) and callable(functions[1])):
129: (12)                        self._functions = functions
130: (8)                     elif isinstance(functions, Transform):
131: (12)                        self._functions = (
132: (17)                             functions.transform,
133: (17)                             lambda x: functions.inverted().transform(x)
134: (12)                        )
135: (8)                     elif functions is None:
136: (12)                        self._functions = (lambda x: x, lambda x: x)
137: (8)                     else:
138: (12)                        raise ValueError('functions argument of secondary Axes '
```

```
(29)                                        'must be a two-tuple of callable functions '
(29)                                        'with the first function being the transform '
(29)                                        'and the second being the inverse')
(8)                     self._set_scale()
(4)                 def draw(self, renderer):
(8)                     """
(8)                     Draw the secondary Axes.
(8)                     Consults the parent Axes for its limits and converts them
(8)                     using the converter specified by
(8)                     `~.axes._secondary_axes.set_functions` (or *functions*
(8)                     parameter when Axes initialized.)
(8)                     """
(8)                     self._set_lims()
(8)                     self._set_scale()
(8)                     super().draw(renderer)
(4)                 def _set_scale(self):
(8)                     """
(8)                     Check if parent has set its scale
(8)                     """
(8)                     if self._orientation == 'x':
(12)                        pscale = self._parent.xaxis.get_scale()
(12)                        set_scale = self.set_xscale
(8)                     else:  # 'y'
(12)                        pscale = self._parent.yaxis.get_scale()
(12)                        set_scale = self.set_yscale
(8)                     if pscale == self._parentscale:
(12)                        return
(8)                     if self._ticks_set:
(12)                        ticks = self._axis.get_ticklocs()
(8)                     set_scale('functionlog' if pscale == 'log' else 'function',
(18)                              functions=self._functions[::-1])
(8)                     if self._ticks_set:
(12)                        self._axis.set_major_locator(mticker.FixedLocator(ticks))
(8)                     self._parentscale = pscale
(4)                 def _set_lims(self):
(8)                     """
(8)                     Set the limits based on parent limits and the convert method
(8)                     between the parent and this secondary Axes.
(8)                     """
(8)                     if self._orientation == 'x':
(12)                        lims = self._parent.get_xlim()
(12)                        set_lim = self.set_xlim
(8)                     else:  # 'y'
(12)                        lims = self._parent.get_ylim()
(12)                        set_lim = self.set_ylim
(8)                     order = lims[0] < lims[1]
(8)                     lims = self._functions[0](np.array(lims))
(8)                     neworder = lims[0] < lims[1]
(8)                     if neworder != order:
(12)                        lims = lims[::-1]
(8)                     set_lim(lims)
(4)                 def set_aspect(self, *args, **kwargs):
(8)                     """
(8)                     Secondary Axes cannot set the aspect ratio, so calling this just
(8)                     sets a warning.
(8)                     """
(8)                     _api.warn_external("Secondary Axes can't set the aspect ratio")
(4)                 def set_color(self, color):
(8)                     """
(8)                     Change the color of the secondary Axes and all decorators.
(8)                     Parameters
(8)                     ----------
(8)                     color : :mpltype:`color`
(8)                     """
(8)                     axis = self._axis_map[self._orientation]
(8)                     axis.set_tick_params(colors=color)
(8)                     for spine in self.spines.values():
(12)                        if spine.axis is axis:
(16)                            spine.set_color(color)
```

```
208: (8)                    axis.label.set_color(color)
209: (0)            _secax_docstring = '''
210: (0)            Warnings
211: (0)            --------
212: (0)            This method is experimental as of 3.1, and the API may change.
213: (0)            Parameters
214: (0)            ----------
215: (0)            location : {'top', 'bottom', 'left', 'right'} or float
216: (4)                The position to put the secondary axis.  Strings can be 'top' or
217: (4)                'bottom' for orientation='x' and 'right' or 'left' for
218: (4)                orientation='y'. A float indicates the relative position on the
219: (4)                parent Axes to put the new Axes, 0.0 being the bottom (or left)
220: (4)                and 1.0 being the top (or right).
221: (0)            functions : 2-tuple of func, or Transform with an inverse
222: (4)                If a 2-tuple of functions, the user specifies the transform
223: (4)                function and its inverse.  i.e.
224: (4)                ``functions=(lambda x: 2 / x, lambda x: 2 / x)`` would be an
225: (4)                reciprocal transform with a factor of 2. Both functions must accept
226: (4)                numpy arrays as input.
227: (4)                The user can also directly supply a subclass of
228: (4)                `.transforms.Transform` so long as it has an inverse.
229: (4)                See :doc:`/gallery/subplots_axes_and_figures/secondary_axis`
230: (4)                for examples of making these conversions.
231: (0)            transform : `.Transform`, optional
232: (4)                If specified, *location* will be
233: (4)                placed relative to this transform (in the direction of the axis)
234: (4)                rather than the parent's axis. i.e. a secondary x-axis will
235: (4)                use the provided y transform and the x transform of the parent.
236: (4)                .. versionadded:: 3.9
237: (0)            Returns
238: (0)            -------
239: (0)            ax : axes._secondary_axes.SecondaryAxis
240: (0)            Other Parameters
241: (0)            ----------------
242: (0)            **kwargs : `~matplotlib.axes.Axes` properties.
243: (4)                Other miscellaneous Axes parameters.
244: (0)            '''
245: (0)            _docstring.interpd.update(_secax_docstring=_secax_docstring)
```

File 78 - __init__.py:

```
1: (0)              from .registry import BackendFilter, backend_registry  # noqa: F401
2: (0)              _QT_FORCE_QT5_BINDING = False
```

File 79 - backend_agg.py:

```
1: (0)              """
2: (0)              An `Anti-Grain Geometry`_ (AGG) backend.
3: (0)              Features that are implemented:
4: (0)              * capstyles and join styles
5: (0)              * dashes
6: (0)              * linewidth
7: (0)              * lines, rectangles, ellipses
8: (0)              * clipping to a rectangle
9: (0)              * output to RGBA and Pillow-supported image formats
10: (0)             * alpha blending
11: (0)             * DPI scaling properly - everything scales properly (dashes, linewidths, etc)
12: (0)             * draw polygon
13: (0)             * freetype2 w/ ft2font
14: (0)             Still TODO:
15: (0)             * integrate screen dpi w/ ppi and text
16: (0)             .. _Anti-Grain Geometry: http://agg.sourceforge.net/antigrain.com
17: (0)             """
18: (0)             from contextlib import nullcontext
19: (0)             from math import radians, cos, sin
```

```
(0)  import numpy as np
(0)  import matplotlib as mpl
(0)  from matplotlib import _api, cbook
(0)  from matplotlib.backend_bases import (
(4)      _Backend, FigureCanvasBase, FigureManagerBase, RendererBase)
(0)  from matplotlib.font_manager import fontManager as _fontManager, get_font
(0)  from matplotlib.ft2font import (LOAD_FORCE_AUTOHINT, LOAD_NO_HINTING,
(32)                                 LOAD_DEFAULT, LOAD_NO_AUTOHINT)
(0)  from matplotlib.mathtext import MathTextParser
(0)  from matplotlib.path import Path
(0)  from matplotlib.transforms import Bbox, BboxBase
(0)  from matplotlib.backends._backend_agg import RendererAgg as _RendererAgg
(0)  def get_hinting_flag():
(4)      mapping = {
(8)          'default': LOAD_DEFAULT,
(8)          'no_autohint': LOAD_NO_AUTOHINT,
(8)          'force_autohint': LOAD_FORCE_AUTOHINT,
(8)          'no_hinting': LOAD_NO_HINTING,
(8)          True: LOAD_FORCE_AUTOHINT,
(8)          False: LOAD_NO_HINTING,
(8)          'either': LOAD_DEFAULT,
(8)          'native': LOAD_NO_AUTOHINT,
(8)          'auto': LOAD_FORCE_AUTOHINT,
(8)          'none': LOAD_NO_HINTING,
(4)      }
(4)      return mapping[mpl.rcParams['text.hinting']]
(0)  class RendererAgg(RendererBase):
(4)      """
(4)      The renderer handles all the drawing primitives using a graphics
(4)      context instance that controls the colors/styles
(4)      """
(4)      def __init__(self, width, height, dpi):
(8)          super().__init__()
(8)          self.dpi = dpi
(8)          self.width = width
(8)          self.height = height
(8)          self._renderer = _RendererAgg(int(width), int(height), dpi)
(8)          self._filter_renderers = []
(8)          self._update_methods()
(8)          self.mathtext_parser = MathTextParser('agg')
(8)          self.bbox = Bbox.from_bounds(0, 0, self.width, self.height)
(4)      def __getstate__(self):
(8)          return {'width': self.width, 'height': self.height, 'dpi': self.dpi}
(4)      def __setstate__(self, state):
(8)          self.__init__(state['width'], state['height'], state['dpi'])
(4)      def _update_methods(self):
(8)          self.draw_gouraud_triangles = self._renderer.draw_gouraud_triangles
(8)          self.draw_image = self._renderer.draw_image
(8)          self.draw_markers = self._renderer.draw_markers
(8)          self.draw_path_collection = self._renderer.draw_path_collection
(8)          self.draw_quad_mesh = self._renderer.draw_quad_mesh
(8)          self.copy_from_bbox = self._renderer.copy_from_bbox
(4)      def draw_path(self, gc, path, transform, rgbFace=None):
(8)          nmax = mpl.rcParams['agg.path.chunksize']  # here at least for testing
(8)          npts = path.vertices.shape[0]
(8)          if (npts > nmax > 100 and path.should_simplify and
(16)                 rgbFace is None and gc.get_hatch() is None):
(12)             nch = np.ceil(npts / nmax)
(12)             chsize = int(np.ceil(npts / nch))
(12)             i0 = np.arange(0, npts, chsize)
(12)             i1 = np.zeros_like(i0)
(12)             i1[:-1] = i0[1:] - 1
(12)             i1[-1] = npts
(12)             for ii0, ii1 in zip(i0, i1):
(16)                 v = path.vertices[ii0:ii1, :]
(16)                 c = path.codes
(16)                 if c is not None:
(20)                     c = c[ii0:ii1]
(20)                     c[0] = Path.MOVETO  # move to end of last chunk
```

```
 89: (16)                            p = Path(v, c)
 90: (16)                            p.simplify_threshold = path.simplify_threshold
 91: (16)                            try:
 92: (20)                                self._renderer.draw_path(gc, p, transform, rgbFace)
 93: (16)                            except OverflowError:
 94: (20)                                msg = (
 95: (24)                                    "Exceeded cell block limit in Agg.\n\n"
 96: (24)                                    "Please reduce the value of "
 97: (24)                                    f"rcParams['agg.path.chunksize'] (currently {nmax}) "
 98: (24)                                    "or increase the path simplification threshold"
 99: (24)                                    "(rcParams['path.simplify_threshold'] = "
100: (24)                                    f"{mpl.rcParams['path.simplify_threshold']:.2f} by "
101: (24)                                    "default and path.simplify_threshold = "
102: (24)                                    f"{path.simplify_threshold:.2f} on the input)."
103: (20)                                )
104: (20)                                raise OverflowError(msg) from None
105: (8)                    else:
106: (12)                       try:
107: (16)                           self._renderer.draw_path(gc, path, transform, rgbFace)
108: (12)                       except OverflowError:
109: (16)                           cant_chunk = ''
110: (16)                           if rgbFace is not None:
111: (20)                               cant_chunk += "- cannot split filled path\n"
112: (16)                           if gc.get_hatch() is not None:
113: (20)                               cant_chunk += "- cannot split hatched path\n"
114: (16)                           if not path.should_simplify:
115: (20)                               cant_chunk += "- path.should_simplify is False\n"
116: (16)                           if len(cant_chunk):
117: (20)                               msg = (
118: (24)                                   "Exceeded cell block limit in Agg, however for the "
119: (24)                                   "following reasons:\n\n"
120: (24)                                   f"{cant_chunk}\n"
121: (24)                                   "we cannot automatically split up this path to draw."
122: (24)                                   "\n\nPlease manually simplify your path."
123: (20)                               )
124: (16)                           else:
125: (20)                               inc_threshold = (
126: (24)                                   "or increase the path simplification threshold"
127: (24)                                   "(rcParams['path.simplify_threshold'] = "
128: (24)                                   f"{mpl.rcParams['path.simplify_threshold']} "
129: (24)                                   "by default and path.simplify_threshold "
130: (24)                                   f"= {path.simplify_threshold} "
131: (24)                                   "on the input)."
132: (24)                                   )
133: (20)                               if nmax > 100:
134: (24)                                   msg = (
135: (28)                                       "Exceeded cell block limit in Agg.  Please reduce
"
136: (28)                                       "the value of rcParams['agg.path.chunksize'] "
137: (28)                                       f"(currently {nmax}) {inc_threshold}"
138: (24)                                   )
139: (20)                               else:
140: (24)                                   msg = (
141: (28)                                       "Exceeded cell block limit in Agg.  Please set "
142: (28)                                       "the value of rcParams['agg.path.chunksize'], "
143: (28)                                       f"(currently {nmax}) to be greater than 100 "
144: (28)                                       + inc_threshold
145: (24)                                   )
146: (16)                           raise OverflowError(msg) from None
147: (4)                def draw_mathtext(self, gc, x, y, s, prop, angle):
148: (8)                    """Draw mathtext using :mod:`matplotlib.mathtext`."""
149: (8)                    ox, oy, width, height, descent, font_image = \
150: (12)                       self.mathtext_parser.parse(s, self.dpi, prop,
151: (39)                                                  antialiased=gc.get_antialiased())
152: (8)                    xd = descent * sin(radians(angle))
153: (8)                    yd = descent * cos(radians(angle))
154: (8)                    x = round(x + ox + xd)
155: (8)                    y = round(y - oy + yd)
156: (8)                    self._renderer.draw_text_image(font_image, x, y + 1, angle, gc)
```

```
(4)     def draw_text(self, gc, x, y, s, prop, angle, ismath=False, mtext=None):
(8)         if ismath:
(12)            return self.draw_mathtext(gc, x, y, s, prop, angle)
(8)         font = self._prepare_font(prop)
(8)         font.set_text(s, 0, flags=get_hinting_flag())
(8)         font.draw_glyphs_to_bitmap(
(12)            antialiased=gc.get_antialiased())
(8)         d = font.get_descent() / 64.0
(8)         xo, yo = font.get_bitmap_offset()
(8)         xo /= 64.0
(8)         yo /= 64.0
(8)         xd = d * sin(radians(angle))
(8)         yd = d * cos(radians(angle))
(8)         x = round(x + xo + xd)
(8)         y = round(y + yo + yd)
(8)         self._renderer.draw_text_image(font, x, y + 1, angle, gc)
(4)     def get_text_width_height_descent(self, s, prop, ismath):
(8)         _api.check_in_list(["TeX", True, False], ismath=ismath)
(8)         if ismath == "TeX":
(12)            return super().get_text_width_height_descent(s, prop, ismath)
(8)         if ismath:
(12)            ox, oy, width, height, descent, font_image = \
(16)                self.mathtext_parser.parse(s, self.dpi, prop)
(12)            return width, height, descent
(8)         font = self._prepare_font(prop)
(8)         font.set_text(s, 0.0, flags=get_hinting_flag())
(8)         w, h = font.get_width_height()  # width and height of unrotated string
(8)         d = font.get_descent()
(8)         w /= 64.0  # convert from subpixels
(8)         h /= 64.0
(8)         d /= 64.0
(8)         return w, h, d
(4)     def draw_tex(self, gc, x, y, s, prop, angle, *, mtext=None):
(8)         size = prop.get_size_in_points()
(8)         texmanager = self.get_texmanager()
(8)         Z = texmanager.get_grey(s, size, self.dpi)
(8)         Z = np.array(Z * 255.0, np.uint8)
(8)         w, h, d = self.get_text_width_height_descent(s, prop, ismath="TeX")
(8)         xd = d * sin(radians(angle))
(8)         yd = d * cos(radians(angle))
(8)         x = round(x + xd)
(8)         y = round(y + yd)
(8)         self._renderer.draw_text_image(Z, x, y, angle, gc)
(4)     def get_canvas_width_height(self):
(8)         return self.width, self.height
(4)     def _prepare_font(self, font_prop):
(8)         """
(8)         Get the `.FT2Font` for *font_prop*, clear its buffer, and set its
size.
(8)         """
(8)         font = get_font(_fontManager._find_fonts_by_props(font_prop))
(8)         font.clear()
(8)         size = font_prop.get_size_in_points()
(8)         font.set_size(size, self.dpi)
(8)         return font
(4)     def points_to_pixels(self, points):
(8)         return points * self.dpi / 72
(4)     def buffer_rgba(self):
(8)         return memoryview(self._renderer)
(4)     def tostring_argb(self):
(8)         return np.asarray(self._renderer).take([3, 0, 1, 2], axis=2).tobytes()
(4)     @_api.deprecated("3.8", alternative="buffer_rgba")
(4)     def tostring_rgb(self):
(8)         return np.asarray(self._renderer).take([0, 1, 2], axis=2).tobytes()
(4)     def clear(self):
(8)         self._renderer.clear()
(4)     def option_image_nocomposite(self):
(8)         return True
(4)     def option_scale_image(self):
```

```
225: (8)                    return False
226: (4)                def restore_region(self, region, bbox=None, xy=None):
227: (8)                    """
228: (8)                    Restore the saved region. If bbox (instance of BboxBase, or
229: (8)                    its extents) is given, only the region specified by the bbox
230: (8)                    will be restored. *xy* (a pair of floats) optionally
231: (8)                    specifies the new position (the LLC of the original region,
232: (8)                    not the LLC of the bbox) where the region will be restored.
233: (8)                    >>> region = renderer.copy_from_bbox()
234: (8)                    >>> x1, y1, x2, y2 = region.get_extents()
235: (8)                    >>> renderer.restore_region(region, bbox=(x1+dx, y1, x2, y2),
236: (8)                    ...                         xy=(x1-dx, y1))
237: (8)                    """
238: (8)                    if bbox is not None or xy is not None:
239: (12)                       if bbox is None:
240: (16)                           x1, y1, x2, y2 = region.get_extents()
241: (12)                       elif isinstance(bbox, BboxBase):
242: (16)                           x1, y1, x2, y2 = bbox.extents
243: (12)                       else:
244: (16)                           x1, y1, x2, y2 = bbox
245: (12)                       if xy is None:
246: (16)                           ox, oy = x1, y1
247: (12)                       else:
248: (16)                           ox, oy = xy
249: (12)                       self._renderer.restore_region(region, int(x1), int(y1),
250: (42)                                                     int(x2), int(y2), int(ox), int(oy))
251: (8)                    else:
252: (12)                       self._renderer.restore_region(region)
253: (4)                def start_filter(self):
254: (8)                    """
255: (8)                    Start filtering. It simply creates a new canvas (the old one is
saved).
256: (8)                    """
257: (8)                    self._filter_renderers.append(self._renderer)
258: (8)                    self._renderer = _RendererAgg(int(self.width), int(self.height),
259: (38)                                                 self.dpi)
260: (8)                    self._update_methods()
261: (4)                def stop_filter(self, post_processing):
262: (8)                    """
263: (8)                    Save the current canvas as an image and apply post processing.
264: (8)                    The *post_processing* function::
265: (11)                      def post_processing(image, dpi):
266: (13)                        ...
267: (13)                        return new_image, offset_x, offset_y
268: (8)                    The saved renderer is restored and the returned image from
269: (8)                    post_processing is plotted (using draw_image) on it.
270: (8)                    """
271: (8)                    orig_img = np.asarray(self.buffer_rgba())
272: (8)                    slice_y, slice_x = cbook._get_nonzero_slices(orig_img[..., 3])
273: (8)                    cropped_img = orig_img[slice_y, slice_x]
274: (8)                    self._renderer = self._filter_renderers.pop()
275: (8)                    self._update_methods()
276: (8)                    if cropped_img.size:
277: (12)                       img, ox, oy = post_processing(cropped_img / 255, self.dpi)
278: (12)                       gc = self.new_gc()
279: (12)                       if img.dtype.kind == 'f':
280: (16)                           img = np.asarray(img * 255., np.uint8)
281: (12)                       self._renderer.draw_image(
282: (16)                           gc, slice_x.start + ox, int(self.height) - slice_y.stop + oy,
283: (16)                           img[::-1])
284: (0)            class FigureCanvasAgg(FigureCanvasBase):
285: (4)                _lastKey = None  # Overwritten per-instance on the first draw.
286: (4)                def copy_from_bbox(self, bbox):
287: (8)                    renderer = self.get_renderer()
288: (8)                    return renderer.copy_from_bbox(bbox)
289: (4)                def restore_region(self, region, bbox=None, xy=None):
290: (8)                    renderer = self.get_renderer()
291: (8)                    return renderer.restore_region(region, bbox, xy)
292: (4)                def draw(self):
```

```
(8)                 self.renderer = self.get_renderer()
(8)                 self.renderer.clear()
(8)                 with (self.toolbar._wait_cursor_for_draw_cm() if self.toolbar
(14)                      else nullcontext()):
(12)                    self.figure.draw(self.renderer)
(12)                    super().draw()
(4)             def get_renderer(self):
(8)                 w, h = self.figure.bbox.size
(8)                 key = w, h, self.figure.dpi
(8)                 reuse_renderer = (self._lastKey == key)
(8)                 if not reuse_renderer:
(12)                    self.renderer = RendererAgg(w, h, self.figure.dpi)
(12)                    self._lastKey = key
(8)                 return self.renderer
(4)             @_api.deprecated("3.8", alternative="buffer_rgba")
(4)             def tostring_rgb(self):
(8)                 """
(8)                 Get the image as RGB `bytes`.
(8)                 `draw` must be called at least once before this function will work and
(8)                 to update the renderer for any subsequent changes to the Figure.
(8)                 """
(8)                 return self.renderer.tostring_rgb()
(4)             def tostring_argb(self):
(8)                 """
(8)                 Get the image as ARGB `bytes`.
(8)                 `draw` must be called at least once before this function will work and
(8)                 to update the renderer for any subsequent changes to the Figure.
(8)                 """
(8)                 return self.renderer.tostring_argb()
(4)             def buffer_rgba(self):
(8)                 """
(8)                 Get the image as a `memoryview` to the renderer's buffer.
(8)                 `draw` must be called at least once before this function will work and
(8)                 to update the renderer for any subsequent changes to the Figure.
(8)                 """
(8)                 return self.renderer.buffer_rgba()
(4)             def print_raw(self, filename_or_obj, *, metadata=None):
(8)                 if metadata is not None:
(12)                    raise ValueError("metadata not supported for raw/rgba")
(8)                 FigureCanvasAgg.draw(self)
(8)                 renderer = self.get_renderer()
(8)                 with cbook.open_file_cm(filename_or_obj, "wb") as fh:
(12)                    fh.write(renderer.buffer_rgba())
(4)             print_rgba = print_raw
(4)             def _print_pil(self, filename_or_obj, fmt, pil_kwargs, metadata=None):
(8)                 """
(8)                 Draw the canvas, then save it using `.image.imsave` (to which
(8)                 *pil_kwargs* and *metadata* are forwarded).
(8)                 """
(8)                 FigureCanvasAgg.draw(self)
(8)                 mpl.image.imsave(
(12)                    filename_or_obj, self.buffer_rgba(), format=fmt, origin="upper",
(12)                    dpi=self.figure.dpi, metadata=metadata, pil_kwargs=pil_kwargs)
(4)             def print_png(self, filename_or_obj, *, metadata=None, pil_kwargs=None):
(8)                 """
(8)                 Write the figure to a PNG file.
(8)                 Parameters
(8)                 ----------
(8)                 filename_or_obj : str or path-like or file-like
(12)                    The file to write to.
(8)                 metadata : dict, optional
(12)                    Metadata in the PNG file as key-value pairs of bytes or latin-1
(12)                    encodable strings.
(12)                    According to the PNG specification, keys must be shorter than 79
(12)                    chars.
(12)                    The `PNG specification`_ defines some common keywords that may be
(12)                    used as appropriate:
(12)                    - Title: Short (one line) title or caption for image.
(12)                    - Author: Name of image's creator.
```

```
362: (12)                        - Description: Description of image (possibly long).
363: (12)                        - Copyright: Copyright notice.
364: (12)                        - Creation Time: Time of original image creation
365: (14)                          (usually RFC 1123 format).
366: (12)                        - Software: Software used to create the image.
367: (12)                        - Disclaimer: Legal disclaimer.
368: (12)                        - Warning: Warning of nature of content.
369: (12)                        - Source: Device used to create the image.
370: (12)                        - Comment: Miscellaneous comment;
371: (14)                          conversion from other image format.
372: (12)                        Other keywords may be invented for other purposes.
373: (12)                        If 'Software' is not given, an autogenerated value for Matplotlib
374: (12)                        will be used.  This can be removed by setting it to *None*.
375: (12)                        For more details see the `PNG specification`_.
376: (12)                        .. _PNG specification: \
377: (16)                            https://www.w3.org/TR/2003/REC-PNG-20031110/#11keywords
378: (8)                    pil_kwargs : dict, optional
379: (12)                        Keyword arguments passed to `PIL.Image.Image.save`.
380: (12)                        If the 'pnginfo' key is present, it completely overrides
381: (12)                        *metadata*, including the default 'Software' key.
382: (8)                    """
383: (8)                    self._print_pil(filename_or_obj, "png", pil_kwargs, metadata)
384: (4)                def print_to_buffer(self):
385: (8)                    FigureCanvasAgg.draw(self)
386: (8)                    renderer = self.get_renderer()
387: (8)                    return (bytes(renderer.buffer_rgba()),
388: (16)                            (int(renderer.width), int(renderer.height)))
389: (4)                def print_jpg(self, filename_or_obj, *, metadata=None, pil_kwargs=None):
390: (8)                    with mpl.rc_context({"savefig.facecolor": "white"}):
391: (12)                        self._print_pil(filename_or_obj, "jpeg", pil_kwargs, metadata)
392: (4)                print_jpeg = print_jpg
393: (4)                def print_tif(self, filename_or_obj, *, metadata=None, pil_kwargs=None):
394: (8)                    self._print_pil(filename_or_obj, "tiff", pil_kwargs, metadata)
395: (4)                print_tiff = print_tif
396: (4)                def print_webp(self, filename_or_obj, *, metadata=None, pil_kwargs=None):
397: (8)                    self._print_pil(filename_or_obj, "webp", pil_kwargs, metadata)
398: (4)                print_jpg.__doc__, print_tif.__doc__, print_webp.__doc__ = map(
399: (8)                    """
400: (8)                    Write the figure to a {} file.
401: (8)                    Parameters
402: (8)                    ----------
403: (8)                    filename_or_obj : str or path-like or file-like
404: (12)                        The file to write to.
405: (8)                    pil_kwargs : dict, optional
406: (12)                        Additional keyword arguments that are passed to
407: (12)                        `PIL.Image.Image.save` when saving the figure.
408: (8)                    """.format, ["JPEG", "TIFF", "WebP"])
409: (0)            @_Backend.export
410: (0)            class _BackendAgg(_Backend):
411: (4)                backend_version = 'v2.2'
412: (4)                FigureCanvas = FigureCanvasAgg
413: (4)                FigureManager = FigureManagerBase
```

File 80 - backend_cairo.py:

```
1: (0)              """
2: (0)              A Cairo backend for Matplotlib
3: (0)              ==============================
4: (0)              :Author: Steve Chaplin and others
5: (0)              This backend depends on cairocffi or pycairo.
6: (0)              """
7: (0)              import functools
8: (0)              import gzip
9: (0)              import math
10: (0)             import numpy as np
11: (0)             try:
12: (4)                 import cairo
```

```
13: (4)                 if cairo.version_info < (1, 14, 0):  # Introduced set_device_scale.
14: (8)                     raise ImportError(f"Cairo backend requires cairo>=1.14.0, "
15: (26)                                      f"but only {cairo.version_info} is available")
16: (0)             except ImportError:
17: (4)                 try:
18: (8)                     import cairocffi as cairo
19: (4)                 except ImportError as err:
20: (8)                     raise ImportError(
21: (12)                        "cairo backend requires that pycairo>=1.14.0 or cairocffi "
22: (12)                        "is installed") from err
23: (0)             from .. import _api, cbook, font_manager
24: (0)             from matplotlib.backend_bases import (
25: (4)                 _Backend, FigureCanvasBase, FigureManagerBase, GraphicsContextBase,
26: (4)                 RendererBase)
27: (0)             from matplotlib.font_manager import ttfFontProperty
28: (0)             from matplotlib.path import Path
29: (0)             from matplotlib.transforms import Affine2D
30: (0)             def _set_rgba(ctx, color, alpha, forced_alpha):
31: (4)                 if len(color) == 3 or forced_alpha:
32: (8)                     ctx.set_source_rgba(*color[:3], alpha)
33: (4)                 else:
34: (8)                     ctx.set_source_rgba(*color)
35: (0)             def _append_path(ctx, path, transform, clip=None):
36: (4)                 for points, code in path.iter_segments(
37: (12)                        transform, remove_nans=True, clip=clip):
38: (8)                     if code == Path.MOVETO:
39: (12)                        ctx.move_to(*points)
40: (8)                     elif code == Path.CLOSEPOLY:
41: (12)                        ctx.close_path()
42: (8)                     elif code == Path.LINETO:
43: (12)                        ctx.line_to(*points)
44: (8)                     elif code == Path.CURVE3:
45: (12)                        cur = np.asarray(ctx.get_current_point())
46: (12)                        a = points[:2]
47: (12)                        b = points[-2:]
48: (12)                        ctx.curve_to(*(cur / 3 + a * 2 / 3), *(a * 2 / 3 + b / 3), *b)
49: (8)                     elif code == Path.CURVE4:
50: (12)                        ctx.curve_to(*points)
51: (0)             def _cairo_font_args_from_font_prop(prop):
52: (4)                 """
53: (4)                 Convert a `.FontProperties` or a `.FontEntry` to arguments that can be
54: (4)                 passed to `.Context.select_font_face`.
55: (4)                 """
56: (4)                 def attr(field):
57: (8)                     try:
58: (12)                        return getattr(prop, f"get_{field}")()
59: (8)                     except AttributeError:
60: (12)                        return getattr(prop, field)
61: (4)                 name = attr("name")
62: (4)                 slant = getattr(cairo, f"FONT_SLANT_{attr('style').upper()}")
63: (4)                 weight = attr("weight")
64: (4)                 weight = (cairo.FONT_WEIGHT_NORMAL
65: (14)                          if font_manager.weight_dict.get(weight, weight) < 550
66: (14)                          else cairo.FONT_WEIGHT_BOLD)
67: (4)                 return name, slant, weight
68: (0)             class RendererCairo(RendererBase):
69: (4)                 def __init__(self, dpi):
70: (8)                     self.dpi = dpi
71: (8)                     self.gc = GraphicsContextCairo(renderer=self)
72: (8)                     self.width = None
73: (8)                     self.height = None
74: (8)                     self.text_ctx = cairo.Context(
75: (11)                       cairo.ImageSurface(cairo.FORMAT_ARGB32, 1, 1))
76: (8)                     super().__init__()
77: (4)                 def set_context(self, ctx):
78: (8)                     surface = ctx.get_target()
79: (8)                     if hasattr(surface, "get_width") and hasattr(surface, "get_height"):
80: (12)                        size = surface.get_width(), surface.get_height()
81: (8)                     elif hasattr(surface, "get_extents"):  # GTK4 RecordingSurface.
```

```
 82: (12)                        ext = surface.get_extents()
 83: (12)                        size = ext.width, ext.height
 84: (8)                     else:  # vector surfaces.
 85: (12)                        ctx.save()
 86: (12)                        ctx.reset_clip()
 87: (12)                        rect, *rest = ctx.copy_clip_rectangle_list()
 88: (12)                        if rest:
 89: (16)                            raise TypeError("Cannot infer surface size")
 90: (12)                        _, _, *size = rect
 91: (12)                        ctx.restore()
 92: (8)                     self.gc.ctx = ctx
 93: (8)                     self.width, self.height = size
 94: (4)                 @staticmethod
 95: (4)                 def _fill_and_stroke(ctx, fill_c, alpha, alpha_overrides):
 96: (8)                     if fill_c is not None:
 97: (12)                        ctx.save()
 98: (12)                        _set_rgba(ctx, fill_c, alpha, alpha_overrides)
 99: (12)                        ctx.fill_preserve()
 100: (12)                       ctx.restore()
 101: (8)                    ctx.stroke()
 102: (4)                def draw_path(self, gc, path, transform, rgbFace=None):
 103: (8)                    ctx = gc.ctx
 104: (8)                    clip = (ctx.clip_extents()
 105: (16)                           if rgbFace is None and gc.get_hatch() is None
 106: (16)                           else None)
 107: (8)                    transform = (transform
 108: (21)                                + Affine2D().scale(1, -1).translate(0, self.height))
 109: (8)                    ctx.new_path()
 110: (8)                    _append_path(ctx, path, transform, clip)
 111: (8)                    if rgbFace is not None:
 112: (12)                       ctx.save()
 113: (12)                       _set_rgba(ctx, rgbFace, gc.get_alpha(), gc.get_forced_alpha())
 114: (12)                       ctx.fill_preserve()
 115: (12)                       ctx.restore()
 116: (8)                    hatch_path = gc.get_hatch_path()
 117: (8)                    if hatch_path:
 118: (12)                       dpi = int(self.dpi)
 119: (12)                       hatch_surface = ctx.get_target().create_similar(
 120: (16)                           cairo.Content.COLOR_ALPHA, dpi, dpi)
 121: (12)                       hatch_ctx = cairo.Context(hatch_surface)
 122: (12)                       _append_path(hatch_ctx, hatch_path,
 123: (25)                                    Affine2D().scale(dpi, -dpi).translate(0, dpi),
 124: (25)                                    None)
 125: (12)
hatch_ctx.set_line_width(self.points_to_pixels(gc.get_hatch_linewidth()))
 126: (12)                       hatch_ctx.set_source_rgba(*gc.get_hatch_color())
 127: (12)                       hatch_ctx.fill_preserve()
 128: (12)                       hatch_ctx.stroke()
 129: (12)                       hatch_pattern = cairo.SurfacePattern(hatch_surface)
 130: (12)                       hatch_pattern.set_extend(cairo.Extend.REPEAT)
 131: (12)                       ctx.save()
 132: (12)                       ctx.set_source(hatch_pattern)
 133: (12)                       ctx.fill_preserve()
 134: (12)                       ctx.restore()
 135: (8)                    ctx.stroke()
 136: (4)                def draw_markers(self, gc, marker_path, marker_trans, path, transform,
 137: (21)                                rgbFace=None):
 138: (8)                    ctx = gc.ctx
 139: (8)                    ctx.new_path()
 140: (8)                    _append_path(ctx, marker_path, marker_trans + Affine2D().scale(1, -1))
 141: (8)                    marker_path = ctx.copy_path_flat()
 142: (8)                    x1, y1, x2, y2 = ctx.fill_extents()
 143: (8)                    if x1 == 0 and y1 == 0 and x2 == 0 and y2 == 0:
 144: (12)                       filled = False
 145: (12)                       rgbFace = None
 146: (8)                    else:
 147: (12)                       filled = True
 148: (8)                    transform = (transform
 149: (21)                               + Affine2D().scale(1, -1).translate(0, self.height))
```

```
150: (8)                    ctx.new_path()
151: (8)                    for i, (vertices, codes) in enumerate(
152: (16)                           path.iter_segments(transform, simplify=False)):
153: (12)                       if len(vertices):
154: (16)                           x, y = vertices[-2:]
155: (16)                           ctx.save()
156: (16)                           ctx.translate(x, y)
157: (16)                           ctx.append_path(marker_path)
158: (16)                           ctx.restore()
159: (16)                           if filled or i % 1000 == 0:
160: (20)                               self._fill_and_stroke(
161: (24)                                   ctx, rgbFace, gc.get_alpha(), gc.get_forced_alpha())
162: (8)                    if not filled:
163: (12)                       self._fill_and_stroke(
164: (16)                           ctx, rgbFace, gc.get_alpha(), gc.get_forced_alpha())
165: (4)                def draw_image(self, gc, x, y, im):
166: (8)                    im = cbook._unmultiplied_rgba8888_to_premultiplied_argb32(im[::-1])
167: (8)                    surface = cairo.ImageSurface.create_for_data(
168: (12)                       im.ravel().data, cairo.FORMAT_ARGB32,
169: (12)                       im.shape[1], im.shape[0], im.shape[1] * 4)
170: (8)                    ctx = gc.ctx
171: (8)                    y = self.height - y - im.shape[0]
172: (8)                    ctx.save()
173: (8)                    ctx.set_source_surface(surface, float(x), float(y))
174: (8)                    ctx.paint()
175: (8)                    ctx.restore()
176: (4)                def draw_text(self, gc, x, y, s, prop, angle, ismath=False, mtext=None):
177: (8)                    if ismath:
178: (12)                       self._draw_mathtext(gc, x, y, s, prop, angle)
179: (8)                    else:
180: (12)                       ctx = gc.ctx
181: (12)                       ctx.new_path()
182: (12)                       ctx.move_to(x, y)
183: (12)                       ctx.save()
184: (12)                       ctx.select_font_face(*_cairo_font_args_from_font_prop(prop))
185: (12)
ctx.set_font_size(self.points_to_pixels(prop.get_size_in_points()))
186: (12)                       opts = cairo.FontOptions()
187: (12)                       opts.set_antialias(gc.get_antialiased())
188: (12)                       ctx.set_font_options(opts)
189: (12)                       if angle:
190: (16)                           ctx.rotate(np.deg2rad(-angle))
191: (12)                       ctx.show_text(s)
192: (12)                       ctx.restore()
193: (4)                def _draw_mathtext(self, gc, x, y, s, prop, angle):
194: (8)                    ctx = gc.ctx
195: (8)                    width, height, descent, glyphs, rects = \
196: (12)                       self._text2path.mathtext_parser.parse(s, self.dpi, prop)
197: (8)                    ctx.save()
198: (8)                    ctx.translate(x, y)
199: (8)                    if angle:
200: (12)                       ctx.rotate(np.deg2rad(-angle))
201: (8)                    for font, fontsize, idx, ox, oy in glyphs:
202: (12)                       ctx.new_path()
203: (12)                       ctx.move_to(ox, -oy)
204: (12)                       ctx.select_font_face(
205: (16)                           *_cairo_font_args_from_font_prop(ttfFontProperty(font)))
206: (12)                       ctx.set_font_size(self.points_to_pixels(fontsize))
207: (12)                       ctx.show_text(chr(idx))
208: (8)                    for ox, oy, w, h in rects:
209: (12)                       ctx.new_path()
210: (12)                       ctx.rectangle(ox, -oy, w, -h)
211: (12)                       ctx.set_source_rgb(0, 0, 0)
212: (12)                       ctx.fill_preserve()
213: (8)                    ctx.restore()
214: (4)                def get_canvas_width_height(self):
215: (8)                    return self.width, self.height
216: (4)                def get_text_width_height_descent(self, s, prop, ismath):
217: (8)                    if ismath == 'TeX':
```

```
218: (12)                        return super().get_text_width_height_descent(s, prop, ismath)
219: (8)                    if ismath:
220: (12)                        width, height, descent, *_ = \
221: (16)                            self._text2path.mathtext_parser.parse(s, self.dpi, prop)
222: (12)                        return width, height, descent
223: (8)                    ctx = self.text_ctx
224: (8)                    ctx.save()
225: (8)                    ctx.select_font_face(*_cairo_font_args_from_font_prop(prop))
226: (8)                    ctx.set_font_size(self.points_to_pixels(prop.get_size_in_points()))
227: (8)                    y_bearing, w, h = ctx.text_extents(s)[1:4]
228: (8)                    ctx.restore()
229: (8)                    return w, h, h + y_bearing
230: (4)                def new_gc(self):
231: (8)                    self.gc.ctx.save()
232: (8)                    self.gc._alpha = 1
233: (8)                    self.gc._forced_alpha = False  # if True, _alpha overrides A from RGBA
234: (8)                    self.gc._hatch = None
235: (8)                    return self.gc
236: (4)                def points_to_pixels(self, points):
237: (8)                    return points / 72 * self.dpi
238: (0)            class GraphicsContextCairo(GraphicsContextBase):
239: (4)                _joind = {
240: (8)                    'bevel':  cairo.LINE_JOIN_BEVEL,
241: (8)                    'miter':  cairo.LINE_JOIN_MITER,
242: (8)                    'round':  cairo.LINE_JOIN_ROUND,
243: (4)                }
244: (4)                _capd = {
245: (8)                    'butt':        cairo.LINE_CAP_BUTT,
246: (8)                    'projecting':  cairo.LINE_CAP_SQUARE,
247: (8)                    'round':       cairo.LINE_CAP_ROUND,
248: (4)                }
249: (4)                def __init__(self, renderer):
250: (8)                    super().__init__()
251: (8)                    self.renderer = renderer
252: (4)                def restore(self):
253: (8)                    self.ctx.restore()
254: (4)                def set_alpha(self, alpha):
255: (8)                    super().set_alpha(alpha)
256: (8)                    _set_rgba(
257: (12)                       self.ctx, self._rgb, self.get_alpha(), self.get_forced_alpha())
258: (4)                def set_antialiased(self, b):
259: (8)                    self.ctx.set_antialias(
260: (12)                       cairo.ANTIALIAS_DEFAULT if b else cairo.ANTIALIAS_NONE)
261: (4)                def get_antialiased(self):
262: (8)                    return self.ctx.get_antialias()
263: (4)                def set_capstyle(self, cs):
264: (8)                    self.ctx.set_line_cap(_api.check_getitem(self._capd, capstyle=cs))
265: (8)                    self._capstyle = cs
266: (4)                def set_clip_rectangle(self, rectangle):
267: (8)                    if not rectangle:
268: (12)                       return
269: (8)                    x, y, w, h = np.round(rectangle.bounds)
270: (8)                    ctx = self.ctx
271: (8)                    ctx.new_path()
272: (8)                    ctx.rectangle(x, self.renderer.height - h - y, w, h)
273: (8)                    ctx.clip()
274: (4)                def set_clip_path(self, path):
275: (8)                    if not path:
276: (12)                       return
277: (8)                    tpath, affine = path.get_transformed_path_and_affine()
278: (8)                    ctx = self.ctx
279: (8)                    ctx.new_path()
280: (8)                    affine = (affine
281: (18)                             + Affine2D().scale(1, -1).translate(0,
self.renderer.height))
282: (8)                    _append_path(ctx, tpath, affine)
283: (8)                    ctx.clip()
284: (4)                def set_dashes(self, offset, dashes):
285: (8)                    self._dashes = offset, dashes
```

```
(8)                     if dashes is None:
(12)                        self.ctx.set_dash([], 0)  # switch dashes off
(8)                     else:
(12)                        self.ctx.set_dash(
(16)                            list(self.renderer.points_to_pixels(np.asarray(dashes))),
(16)                            offset)
(4)                 def set_foreground(self, fg, isRGBA=None):
(8)                     super().set_foreground(fg, isRGBA)
(8)                     if len(self._rgb) == 3:
(12)                        self.ctx.set_source_rgb(*self._rgb)
(8)                     else:
(12)                        self.ctx.set_source_rgba(*self._rgb)
(4)                 def get_rgb(self):
(8)                     return self.ctx.get_source().get_rgba()[:3]
(4)                 def set_joinstyle(self, js):
(8)                     self.ctx.set_line_join(_api.check_getitem(self._joind, joinstyle=js))
(8)                     self._joinstyle = js
(4)                 def set_linewidth(self, w):
(8)                     self._linewidth = float(w)
(8)                     self.ctx.set_line_width(self.renderer.points_to_pixels(w))
(0)             class _CairoRegion:
(4)                 def __init__(self, slices, data):
(8)                     self._slices = slices
(8)                     self._data = data
(0)             class FigureCanvasCairo(FigureCanvasBase):
(4)                 @property
(4)                 def _renderer(self):
(8)                     if not hasattr(self, "_cached_renderer"):
(12)                        self._cached_renderer = RendererCairo(self.figure.dpi)
(8)                     return self._cached_renderer
(4)                 def get_renderer(self):
(8)                     return self._renderer
(4)                 def copy_from_bbox(self, bbox):
(8)                     surface = self._renderer.gc.ctx.get_target()
(8)                     if not isinstance(surface, cairo.ImageSurface):
(12)                        raise RuntimeError(
(16)                            "copy_from_bbox only works when rendering to an ImageSurface")
(8)                     sw = surface.get_width()
(8)                     sh = surface.get_height()
(8)                     x0 = math.ceil(bbox.x0)
(8)                     x1 = math.floor(bbox.x1)
(8)                     y0 = math.ceil(sh - bbox.y1)
(8)                     y1 = math.floor(sh - bbox.y0)
(8)                     if not (0 <= x0 and x1 <= sw and bbox.x0 <= bbox.x1
(16)                            and 0 <= y0 and y1 <= sh and bbox.y0 <= bbox.y1):
(12)                        raise ValueError("Invalid bbox")
(8)                     sls = slice(y0, y0 + max(y1 - y0, 0)), slice(x0, x0 + max(x1 - x0, 0))
(8)                     data = (np.frombuffer(surface.get_data(), np.uint32)
(16)                            .reshape((sh, sw))[sls].copy())
(8)                     return _CairoRegion(sls, data)
(4)                 def restore_region(self, region):
(8)                     surface = self._renderer.gc.ctx.get_target()
(8)                     if not isinstance(surface, cairo.ImageSurface):
(12)                        raise RuntimeError(
(16)                            "restore_region only works when rendering to an ImageSurface")
(8)                     surface.flush()
(8)                     sw = surface.get_width()
(8)                     sh = surface.get_height()
(8)                     sly, slx = region._slices
(8)                     (np.frombuffer(surface.get_data(), np.uint32)
(9)                      .reshape((sh, sw))[sly, slx]) = region._data
(8)                     surface.mark_dirty_rectangle(
(12)                        slx.start, sly.start, slx.stop - slx.start, sly.stop - sly.start)
(4)                 def print_png(self, fobj):
(8)                     self._get_printed_image_surface().write_to_png(fobj)
(4)                 def print_rgba(self, fobj):
(8)                     width, height = self.get_width_height()
(8)                     buf = self._get_printed_image_surface().get_data()
(8)                     fobj.write(cbook._premultiplied_argb32_to_unmultiplied_rgba8888(
```

```
355: (12)                        np.asarray(buf).reshape((width, height, 4))))
356: (4)                print_raw = print_rgba
357: (4)                def _get_printed_image_surface(self):
358: (8)                    self._renderer.dpi = self.figure.dpi
359: (8)                    width, height = self.get_width_height()
360: (8)                    surface = cairo.ImageSurface(cairo.FORMAT_ARGB32, width, height)
361: (8)                    self._renderer.set_context(cairo.Context(surface))
362: (8)                    self.figure.draw(self._renderer)
363: (8)                    return surface
364: (4)                def _save(self, fmt, fobj, *, orientation='portrait'):
365: (8)                    dpi = 72
366: (8)                    self.figure.dpi = dpi
367: (8)                    w_in, h_in = self.figure.get_size_inches()
368: (8)                    width_in_points, height_in_points = w_in * dpi, h_in * dpi
369: (8)                    if orientation == 'landscape':
370: (12)                       width_in_points, height_in_points = (
371: (16)                           height_in_points, width_in_points)
372: (8)                    if fmt == 'ps':
373: (12)                       if not hasattr(cairo, 'PSSurface'):
374: (16)                           raise RuntimeError('cairo has not been compiled with PS '
375: (35)                                              'support enabled')
376: (12)                       surface = cairo.PSSurface(fobj, width_in_points, height_in_points)
377: (8)                    elif fmt == 'pdf':
378: (12)                       if not hasattr(cairo, 'PDFSurface'):
379: (16)                           raise RuntimeError('cairo has not been compiled with PDF '
380: (35)                                              'support enabled')
381: (12)                       surface = cairo.PDFSurface(fobj, width_in_points,
height_in_points)
382: (8)                    elif fmt in ('svg', 'svgz'):
383: (12)                       if not hasattr(cairo, 'SVGSurface'):
384: (16)                           raise RuntimeError('cairo has not been compiled with SVG '
385: (35)                                              'support enabled')
386: (12)                       if fmt == 'svgz':
387: (16)                           if isinstance(fobj, str):
388: (20)                               fobj = gzip.GzipFile(fobj, 'wb')
389: (16)                           else:
390: (20)                               fobj = gzip.GzipFile(None, 'wb', fileobj=fobj)
391: (12)                       surface = cairo.SVGSurface(fobj, width_in_points,
height_in_points)
392: (8)                    else:
393: (12)                       raise ValueError(f"Unknown format: {fmt!r}")
394: (8)                    self._renderer.dpi = self.figure.dpi
395: (8)                    self._renderer.set_context(cairo.Context(surface))
396: (8)                    ctx = self._renderer.gc.ctx
397: (8)                    if orientation == 'landscape':
398: (12)                       ctx.rotate(np.pi / 2)
399: (12)                       ctx.translate(0, -height_in_points)
400: (8)                    self.figure.draw(self._renderer)
401: (8)                    ctx.show_page()
402: (8)                    surface.finish()
403: (8)                    if fmt == 'svgz':
404: (12)                       fobj.close()
405: (4)                print_pdf = functools.partialmethod(_save, "pdf")
406: (4)                print_ps = functools.partialmethod(_save, "ps")
407: (4)                print_svg = functools.partialmethod(_save, "svg")
408: (4)                print_svgz = functools.partialmethod(_save, "svgz")
409: (0)            @_Backend.export
410: (0)            class _BackendCairo(_Backend):
411: (4)                backend_version = cairo.version
412: (4)                FigureCanvas = FigureCanvasCairo
413: (4)                FigureManager = FigureManagerBase
```

File 81 - _backend_gtk.py:

```
1: (0)              """
2: (0)              Common code for GTK3 and GTK4 backends.
3: (0)              """
```

```
 4: (0)              import logging
 5: (0)              import sys
 6: (0)              import matplotlib as mpl
 7: (0)              from matplotlib import _api, backend_tools, cbook
 8: (0)              from matplotlib._pylab_helpers import Gcf
 9: (0)              from matplotlib.backend_bases import (
10: (4)                  _Backend, FigureCanvasBase, FigureManagerBase, NavigationToolbar2,
11: (4)                  TimerBase)
12: (0)              from matplotlib.backend_tools import Cursors
13: (0)              import gi
14: (0)              from gi.repository import Gdk, Gio, GLib, Gtk
15: (0)              try:
16: (4)                  gi.require_foreign("cairo")
17: (0)              except ImportError as e:
18: (4)                  raise ImportError("Gtk-based backends require cairo") from e
19: (0)              _log = logging.getLogger(__name__)
20: (0)              _application = None  # Placeholder
21: (0)              def _shutdown_application(app):
22: (4)                  for win in app.get_windows():
23: (8)                      win.close()
24: (4)                  app._created_by_matplotlib = True
25: (4)                  global _application
26: (4)                  _application = None
27: (0)              def _create_application():
28: (4)                  global _application
29: (4)                  if _application is None:
30: (8)                      app = Gio.Application.get_default()
31: (8)                      if app is None or getattr(app, '_created_by_matplotlib', False):
32: (12)                         if not mpl._c_internal_utils.display_is_valid():
33: (16)                             raise RuntimeError('Invalid DISPLAY variable')
34: (12)                         _application = Gtk.Application.new('org.matplotlib.Matplotlib3',
35: (47)
Gio.ApplicationFlags.NON_UNIQUE)
36: (12)                         _application.connect('activate', lambda *args, **kwargs: None)
37: (12)                         _application.connect('shutdown', _shutdown_application)
38: (12)                         _application.register()
39: (12)                         cbook._setup_new_guiapp()
40: (8)                      else:
41: (12)                         _application = app
42: (4)                  return _application
43: (0)              def mpl_to_gtk_cursor_name(mpl_cursor):
44: (4)                  return _api.check_getitem({
45: (8)                      Cursors.MOVE: "move",
46: (8)                      Cursors.HAND: "pointer",
47: (8)                      Cursors.POINTER: "default",
48: (8)                      Cursors.SELECT_REGION: "crosshair",
49: (8)                      Cursors.WAIT: "wait",
50: (8)                      Cursors.RESIZE_HORIZONTAL: "ew-resize",
51: (8)                      Cursors.RESIZE_VERTICAL: "ns-resize",
52: (4)                  }, cursor=mpl_cursor)
53: (0)              class TimerGTK(TimerBase):
54: (4)                  """Subclass of `.TimerBase` using GTK timer events."""
55: (4)                  def __init__(self, *args, **kwargs):
56: (8)                      self._timer = None
57: (8)                      super().__init__(*args, **kwargs)
58: (4)                  def _timer_start(self):
59: (8)                      self._timer_stop()
60: (8)                      self._timer = GLib.timeout_add(self._interval, self._on_timer)
61: (4)                  def _timer_stop(self):
62: (8)                      if self._timer is not None:
63: (12)                         GLib.source_remove(self._timer)
64: (12)                         self._timer = None
65: (4)                  def _timer_set_interval(self):
66: (8)                      if self._timer is not None:
67: (12)                         self._timer_stop()
68: (12)                         self._timer_start()
69: (4)                  def _on_timer(self):
70: (8)                      super()._on_timer()
71: (8)                      if self.callbacks and not self._single:
```

```
 72: (12)                        return True
 73: (8)                     else:
 74: (12)                        self._timer = None
 75: (12)                        return False
 76: (0)             class _FigureCanvasGTK(FigureCanvasBase):
 77: (4)                 _timer_cls = TimerGTK
 78: (0)             class _FigureManagerGTK(FigureManagerBase):
 79: (4)                 """
 80: (4)                 Attributes
 81: (4)                 ----------
 82: (4)                 canvas : `FigureCanvas`
 83: (8)                     The FigureCanvas instance
 84: (4)                 num : int or str
 85: (8)                     The Figure number
 86: (4)                 toolbar : Gtk.Toolbar or Gtk.Box
 87: (8)                     The toolbar
 88: (4)                 vbox : Gtk.VBox
 89: (8)                     The Gtk.VBox containing the canvas and toolbar
 90: (4)                 window : Gtk.Window
 91: (8)                     The Gtk.Window
 92: (4)                 """
 93: (4)                 def __init__(self, canvas, num):
 94: (8)                     self._gtk_ver = gtk_ver = Gtk.get_major_version()
 95: (8)                     app = _create_application()
 96: (8)                     self.window = Gtk.Window()
 97: (8)                     app.add_window(self.window)
 98: (8)                     super().__init__(canvas, num)
 99: (8)                     if gtk_ver == 3:
 100: (12)                       self.window.set_wmclass("matplotlib", "Matplotlib")
 101: (12)                       icon_ext = "png" if sys.platform == "win32" else "svg"
 102: (12)                       self.window.set_icon_from_file(
 103: (16)                           str(cbook._get_data_path(f"images/matplotlib.{icon_ext}")))
 104: (8)                    self.vbox = Gtk.Box()
 105: (8)                    self.vbox.set_property("orientation", Gtk.Orientation.VERTICAL)
 106: (8)                    if gtk_ver == 3:
 107: (12)                       self.window.add(self.vbox)
 108: (12)                       self.vbox.show()
 109: (12)                       self.canvas.show()
 110: (12)                       self.vbox.pack_start(self.canvas, True, True, 0)
 111: (8)                    elif gtk_ver == 4:
 112: (12)                       self.window.set_child(self.vbox)
 113: (12)                       self.vbox.prepend(self.canvas)
 114: (8)                    w, h = self.canvas.get_width_height()
 115: (8)                    if self.toolbar is not None:
 116: (12)                       if gtk_ver == 3:
 117: (16)                           self.toolbar.show()
 118: (16)                           self.vbox.pack_end(self.toolbar, False, False, 0)
 119: (12)                       elif gtk_ver == 4:
 120: (16)                           sw =
Gtk.ScrolledWindow(vscrollbar_policy=Gtk.PolicyType.NEVER)
 121: (16)                           sw.set_child(self.toolbar)
 122: (16)                           self.vbox.append(sw)
 123: (12)                       min_size, nat_size = self.toolbar.get_preferred_size()
 124: (12)                       h += nat_size.height
 125: (8)                    self.window.set_default_size(w, h)
 126: (8)                    self._destroying = False
 127: (8)                    self.window.connect("destroy", lambda *args: Gcf.destroy(self))
 128: (8)                    self.window.connect({3: "delete_event", 4: "close-request"}[gtk_ver],
 129: (28)                                       lambda *args: Gcf.destroy(self))
 130: (8)                    if mpl.is_interactive():
 131: (12)                       self.window.show()
 132: (12)                       self.canvas.draw_idle()
 133: (8)                    self.canvas.grab_focus()
 134: (4)                dabout(self, *args):
 135: (8)                    if self._destroying:
 136: (12)                       return
 137: (8)                    self._destroying = True
 138: (8)                    self.window.destroy()
 139: (8)                    self.canvas.destroy()
```

```
(4)                 @classmethod
(4)                 def start_main_loop(cls):
(8)                     global _application
(8)                     if _application is None:
(12)                        return
(8)                     try:
(12)                        _application.run()  # Quits when all added windows close.
(8)                     except KeyboardInterrupt:
(12)                        context = GLib.MainContext.default()
(12)                        while context.pending():
(16)                            context.iteration(True)
(12)                        raise
(8)                     finally:
(12)                        _application = None
(4)                 def show(self):
(8)                     self.window.show()
(8)                     self.canvas.draw()
(8)                     if mpl.rcParams["figure.raise_window"]:
(12)                        meth_name = {3: "get_window", 4: "get_surface"}[self._gtk_ver]
(12)                        if getattr(self.window, meth_name)():
(16)                            self.window.present()
(12)                        else:
(16)                            _api.warn_external("Cannot raise window yet to be setup")
(4)                 def full_screen_toggle(self):
(8)                     is_fullscreen = {
(12)                        3: lambda w: (w.get_window().get_state()
(26)                                      & Gdk.WindowState.FULLSCREEN),
(12)                        4: lambda w: w.is_fullscreen(),
(8)                     }[self._gtk_ver]
(8)                     if is_fullscreen(self.window):
(12)                        self.window.unfullscreen()
(8)                     else:
(12)                        self.window.fullscreen()
(4)                 def get_window_title(self):
(8)                     return self.window.get_title()
(4)                 def set_window_title(self, title):
(8)                     self.window.set_title(title)
(4)                 def resize(self, width, height):
(8)                     width = int(width / self.canvas.device_pixel_ratio)
(8)                     height = int(height / self.canvas.device_pixel_ratio)
(8)                     if self.toolbar:
(12)                        min_size, nat_size = self.toolbar.get_preferred_size()
(12)                        height += nat_size.height
(8)                     canvas_size = self.canvas.get_allocation()
(8)                     if self._gtk_ver >= 4 or canvas_size.width == canvas_size.height == 1:
(12)                        self.window.set_default_size(width, height)
(8)                     else:
(12)                        self.window.resize(width, height)
(0)             class _NavigationToolbar2GTK(NavigationToolbar2):
(4)                 def set_message(self, s):
(8)                     escaped = GLib.markup_escape_text(s)
(8)                     self.message.set_markup(f'<small>{escaped}</small>')
(4)                 def draw_rubberband(self, event, x0, y0, x1, y1):
(8)                     height = self.canvas.figure.bbox.height
(8)                     y1 = height - y1
(8)                     y0 = height - y0
(8)                     rect = [int(val) for val in (x0, y0, x1 - x0, y1 - y0)]
(8)                     self.canvas._draw_rubberband(rect)
(4)                 def remove_rubberband(self):
(8)                     self.canvas._draw_rubberband(None)
(4)                 def _update_buttons_checked(self):
(8)                     for name, active in [("Pan", "PAN"), ("Zoom", "ZOOM")]:
(12)                        button = self._gtk_ids.get(name)
(12)                        if button:
(16)                            with button.handler_block(button._signal_handler):
(20)                                button.set_active(self.mode.name == active)
(4)                 def pan(self, *args):
(8)                     super().pan(*args)
(8)                     self._update_buttons_checked()
```

```
209: (4)                def zoom(self, *args):
210: (8)                    super().zoom(*args)
211: (8)                    self._update_buttons_checked()
212: (4)                def set_history_buttons(self):
213: (8)                    can_backward = self._nav_stack._pos > 0
214: (8)                    can_forward = self._nav_stack._pos < len(self._nav_stack) - 1
215: (8)                    if 'Back' in self._gtk_ids:
216: (12)                       self._gtk_ids['Back'].set_sensitive(can_backward)
217: (8)                    if 'Forward' in self._gtk_ids:
218: (12)                       self._gtk_ids['Forward'].set_sensitive(can_forward)
219: (0)            class RubberbandGTK(backend_tools.RubberbandBase):
220: (4)                def draw_rubberband(self, x0, y0, x1, y1):
221: (8)                    _NavigationToolbar2GTK.draw_rubberband(
222: (12)                       self._make_classic_style_pseudo_toolbar(), None, x0, y0, x1, y1)
223: (4)                def remove_rubberband(self):
224: (8)                    _NavigationToolbar2GTK.remove_rubberband(
225: (12)                       self._make_classic_style_pseudo_toolbar())
226: (0)            class ConfigureSubplotsGTK(backend_tools.ConfigureSubplotsBase):
227: (4)                def trigger(self, *args):
228: (8)                    _NavigationToolbar2GTK.configure_subplots(self, None)
229: (0)            class _BackendGTK(_Backend):
230: (4)                backend_version = "{}.{}.{}".format(
231: (8)                    Gtk.get_major_version(),
232: (8)                    Gtk.get_minor_version(),
233: (8)                    Gtk.get_micro_version(),
234: (4)                )
235: (4)                mainloop = _FigureManagerGTK.start_main_loop
```

File 82 - backend_gtk3.py:

```
1: (0)              import functools
2: (0)              import logging
3: (0)              import os
4: (0)              from pathlib import Path
5: (0)              import matplotlib as mpl
6: (0)              from matplotlib import _api, backend_tools, cbook
7: (0)              from matplotlib.backend_bases import (
8: (4)                  ToolContainerBase, CloseEvent, KeyEvent, LocationEvent, MouseEvent,
9: (4)                  ResizeEvent)
10: (0)             try:
11: (4)                 import gi
12: (0)             except ImportError as err:
13: (4)                 raise ImportError("The GTK3 backends require PyGObject") from err
14: (0)             try:
15: (4)                 gi.require_version("Gtk", "3.0")
16: (0)             except ValueError as e:
17: (4)                 raise ImportError(e) from e
18: (0)             from gi.repository import Gio, GLib, GObject, Gtk, Gdk
19: (0)             from . import _backend_gtk
20: (0)             from ._backend_gtk import (  # noqa: F401 # pylint: disable=W0611
21: (4)                 _BackendGTK, _FigureCanvasGTK, _FigureManagerGTK, _NavigationToolbar2GTK,
22: (4)                 TimerGTK as TimerGTK3,
23: (0)             )
24: (0)             _log = logging.getLogger(__name__)
25: (0)             @functools.cache
26: (0)             def _mpl_to_gtk_cursor(mpl_cursor):
27: (4)                 return Gdk.Cursor.new_from_name(
28: (8)                     Gdk.Display.get_default(),
29: (8)                     _backend_gtk.mpl_to_gtk_cursor_name(mpl_cursor))
30: (0)             class FigureCanvasGTK3(_FigureCanvasGTK, Gtk.DrawingArea):
31: (4)                 required_interactive_framework = "gtk3"
32: (4)                 manager_class = _api.classproperty(lambda cls: FigureManagerGTK3)
33: (4)                 event_mask = (Gdk.EventMask.BUTTON_PRESS_MASK
34: (18)                              | Gdk.EventMask.BUTTON_RELEASE_MASK
35: (18)                              | Gdk.EventMask.EXPOSURE_MASK
36: (18)                              | Gdk.EventMask.KEY_PRESS_MASK
37: (18)                              | Gdk.EventMask.KEY_RELEASE_MASK
```

```
 38: (18)                                | Gdk.EventMask.ENTER_NOTIFY_MASK
 39: (18)                                | Gdk.EventMask.LEAVE_NOTIFY_MASK
 40: (18)                                | Gdk.EventMask.POINTER_MOTION_MASK
 41: (18)                                | Gdk.EventMask.SCROLL_MASK)
 42: (4)                 def __init__(self, figure=None):
 43: (8)                     super().__init__(figure=figure)
 44: (8)                     self._idle_draw_id = 0
 45: (8)                     self._rubberband_rect = None
 46: (8)                     self.connect('scroll_event',         self.scroll_event)
 47: (8)                     self.connect('button_press_event',   self.button_press_event)
 48: (8)                     self.connect('button_release_event', self.button_release_event)
 49: (8)                     self.connect('configure_event',      self.configure_event)
 50: (8)                     self.connect('screen-changed',       self._update_device_pixel_ratio)
 51: (8)                     self.connect('notify::scale-factor', self._update_device_pixel_ratio)
 52: (8)                     self.connect('draw',                 self.on_draw_event)
 53: (8)                     self.connect('draw',                 self._post_draw)
 54: (8)                     self.connect('key_press_event',      self.key_press_event)
 55: (8)                     self.connect('key_release_event',    self.key_release_event)
 56: (8)                     self.connect('motion_notify_event',  self.motion_notify_event)
 57: (8)                     self.connect('enter_notify_event',   self.enter_notify_event)
 58: (8)                     self.connect('leave_notify_event',   self.leave_notify_event)
 59: (8)                     self.connect('size_allocate',        self.size_allocate)
 60: (8)                     self.set_events(self.__class__.event_mask)
 61: (8)                     self.set_can_focus(True)
 62: (8)                     css = Gtk.CssProvider()
 63: (8)                     css.load_from_data(b".matplotlib-canvas { background-color: white; }")
 64: (8)                     style_ctx = self.get_style_context()
 65: (8)                     style_ctx.add_provider(css, Gtk.STYLE_PROVIDER_PRIORITY_APPLICATION)
 66: (8)                     style_ctx.add_class("matplotlib-canvas")
 67: (4)                 def destroy(self):
 68: (8)                     CloseEvent("close_event", self)._process()
 69: (4)                 def set_cursor(self, cursor):
 70: (8)                     window = self.get_property("window")
 71: (8)                     if window is not None:
 72: (12)                        window.set_cursor(_mpl_to_gtk_cursor(cursor))
 73: (12)                        context = GLib.MainContext.default()
 74: (12)                        context.iteration(True)
 75: (4)                 def _mpl_coords(self, event=None):
 76: (8)                     """
 77: (8)                     Convert the position of a GTK event, or of the current cursor position
 78: (8)                     if *event* is None, to Matplotlib coordinates.
 79: (8)                     GTK use logical pixels, but the figure is scaled to physical pixels
for
 80: (8)                     rendering.  Transform to physical pixels so that all of the down-
stream
 81: (8)                     transforms work as expected.
 82: (8)                     Also, the origin is different and needs to be corrected.
 83: (8)                     """
 84: (8)                     if event is None:
 85: (12)                        window = self.get_window()
 86: (12)                        t, x, y, state = window.get_device_position(
 87: (16)
window.get_display().get_device_manager().get_client_pointer())
 88: (8)                     else:
 89: (12)                        x, y = event.x, event.y
 90: (8)                     x = x * self.device_pixel_ratio
 91: (8)                     y = self.figure.bbox.height - y * self.device_pixel_ratio
 92: (8)                     return x, y
 93: (4)                 def scroll_event(self, widget, event):
 94: (8)                     step = 1 if event.direction == Gdk.ScrollDirection.UP else -1
 95: (8)                     MouseEvent("scroll_event", self,
 96: (19)                               *self._mpl_coords(event), step=step,
 97: (19)                               modifiers=self._mpl_modifiers(event.state),
 98: (19)                               guiEvent=event)._process()
 99: (8)                     return False  # finish event propagation?
 100: (4)                 def button_press_event(self, widget, event):
 101: (8)                     MouseEvent("button_press_event", self,
 102: (19)                              *self._mpl_coords(event), event.button,
 103: (19)                              modifiers=self._mpl_modifiers(event.state),
```

```
(19)                                guiEvent=event)._process()
(8)                     return False  # finish event propagation?
(4)                 def button_release_event(self, widget, event):
(8)                     MouseEvent("button_release_event", self,
(19)                                *self._mpl_coords(event), event.button,
(19)                                modifiers=self._mpl_modifiers(event.state),
(19)                                guiEvent=event)._process()
(8)                     return False  # finish event propagation?
(4)                 def key_press_event(self, widget, event):
(8)                     KeyEvent("key_press_event", self,
(17)                             self._get_key(event), *self._mpl_coords(),
(17)                             guiEvent=event)._process()
(8)                     return True  # stop event propagation
(4)                 def key_release_event(self, widget, event):
(8)                     KeyEvent("key_release_event", self,
(17)                             self._get_key(event), *self._mpl_coords(),
(17)                             guiEvent=event)._process()
(8)                     return True  # stop event propagation
(4)                 def motion_notify_event(self, widget, event):
(8)                     MouseEvent("motion_notify_event", self, *self._mpl_coords(event),
(19)                                modifiers=self._mpl_modifiers(event.state),
(19)                                guiEvent=event)._process()
(8)                     return False  # finish event propagation?
(4)                 def enter_notify_event(self, widget, event):
(8)                     gtk_mods = Gdk.Keymap.get_for_display(
(12)                        self.get_display()).get_modifier_state()
(8)                     LocationEvent("figure_enter_event", self, *self._mpl_coords(event),
(22)                                  modifiers=self._mpl_modifiers(gtk_mods),
(22)                                  guiEvent=event)._process()
(4)                 def leave_notify_event(self, widget, event):
(8)                     gtk_mods = Gdk.Keymap.get_for_display(
(12)                        self.get_display()).get_modifier_state()
(8)                     LocationEvent("figure_leave_event", self, *self._mpl_coords(event),
(22)                                  modifiers=self._mpl_modifiers(gtk_mods),
(22)                                  guiEvent=event)._process()
(4)                 def size_allocate(self, widget, allocation):
(8)                     dpival = self.figure.dpi
(8)                     winch = allocation.width * self.device_pixel_ratio / dpival
(8)                     hinch = allocation.height * self.device_pixel_ratio / dpival
(8)                     self.figure.set_size_inches(winch, hinch, forward=False)
(8)                     ResizeEvent("resize_event", self)._process()
(8)                     self.draw_idle()
(4)                 @staticmethod
(4)                 def _mpl_modifiers(event_state, *, exclude=None):
(8)                     modifiers = [
(12)                        ("ctrl", Gdk.ModifierType.CONTROL_MASK, "control"),
(12)                        ("alt", Gdk.ModifierType.MOD1_MASK, "alt"),
(12)                        ("shift", Gdk.ModifierType.SHIFT_MASK, "shift"),
(12)                        ("super", Gdk.ModifierType.MOD4_MASK, "super"),
(8)                     ]
(8)                     return [name for name, mask, key in modifiers
(16)                            if exclude != key and event_state & mask]
(4)                 def _get_key(self, event):
(8)                     unikey = chr(Gdk.keyval_to_unicode(event.keyval))
(8)                     key = cbook._unikey_or_keysym_to_mplkey(
(12)                        unikey, Gdk.keyval_name(event.keyval))
(8)                     mods = self._mpl_modifiers(event.state, exclude=key)
(8)                     if "shift" in mods and unikey.isprintable():
(12)                        mods.remove("shift")
(8)                     return "+".join([*mods, key])
(4)                 def _update_device_pixel_ratio(self, *args, **kwargs):
(8)                     if self._set_device_pixel_ratio(self.get_scale_factor()):
(12)                        self.queue_resize()
(12)                        self.queue_draw()
(4)                 def configure_event(self, widget, event):
(8)                     if widget.get_property("window") is None:
(12)                        return
(8)                     w = event.width * self.device_pixel_ratio
(8)                     h = event.height * self.device_pixel_ratio
```

```
(8)                     if w < 3 or h < 3:
(12)                        return  # empty fig
(8)                     dpi = self.figure.dpi
(8)                     self.figure.set_size_inches(w / dpi, h / dpi, forward=False)
(8)                     return False  # finish event propagation?
(4)                 def _draw_rubberband(self, rect):
(8)                     self._rubberband_rect = rect
(8)                     self.queue_draw()
(4)                 def _post_draw(self, widget, ctx):
(8)                     if self._rubberband_rect is None:
(12)                        return
(8)                     x0, y0, w, h = (dim / self.device_pixel_ratio
(24)                                     for dim in self._rubberband_rect)
(8)                     x1 = x0 + w
(8)                     y1 = y0 + h
(8)                     ctx.move_to(x0, y0)
(8)                     ctx.line_to(x0, y1)
(8)                     ctx.move_to(x0, y0)
(8)                     ctx.line_to(x1, y0)
(8)                     ctx.move_to(x0, y1)
(8)                     ctx.line_to(x1, y1)
(8)                     ctx.move_to(x1, y0)
(8)                     ctx.line_to(x1, y1)
(8)                     ctx.set_antialias(1)
(8)                     ctx.set_line_width(1)
(8)                     ctx.set_dash((3, 3), 0)
(8)                     ctx.set_source_rgb(0, 0, 0)
(8)                     ctx.stroke_preserve()
(8)                     ctx.set_dash((3, 3), 3)
(8)                     ctx.set_source_rgb(1, 1, 1)
(8)                     ctx.stroke()
(4)                 def on_draw_event(self, widget, ctx):
(8)                     pass
(4)                 def draw(self):
(8)                     if self.is_drawable():
(12)                        self.queue_draw()
(4)                 def draw_idle(self):
(8)                     if self._idle_draw_id != 0:
(12)                        return
(8)                     def idle_draw(*args):
(12)                        try:
(16)                            self.draw()
(12)                        finally:
(16)                            self._idle_draw_id = 0
(12)                        return False
(8)                     self._idle_draw_id = GLib.idle_add(idle_draw)
(4)                 def flush_events(self):
(8)                     context = GLib.MainContext.default()
(8)                     while context.pending():
(12)                        context.iteration(True)
(0)             class NavigationToolbar2GTK3(_NavigationToolbar2GTK, Gtk.Toolbar):
(4)                 def __init__(self, canvas):
(8)                     GObject.GObject.__init__(self)
(8)                     self.set_style(Gtk.ToolbarStyle.ICONS)
(8)                     self._gtk_ids = {}
(8)                     for text, tooltip_text, image_file, callback in self.toolitems:
(12)                        if text is None:
(16)                            self.insert(Gtk.SeparatorToolItem(), -1)
(16)                            continue
(12)                        image = Gtk.Image.new_from_gicon(
(16)                            Gio.Icon.new_for_string(
(20)                                str(cbook._get_data_path('images',
(45)                                                         f'{image_file}-symbolic.svg'))),
(16)                            Gtk.IconSize.LARGE_TOOLBAR)
(12)                        self._gtk_ids[text] = button = (
(16)                            Gtk.ToggleToolButton() if callback in ['zoom', 'pan'] else
(16)                            Gtk.ToolButton())
(12)                        button.set_label(text)
(12)                        button.set_icon_widget(image)
```

```
(12)                        button._signal_handler = button.connect(
(16)                            'clicked', getattr(self, callback))
(12)                        button.set_tooltip_text(tooltip_text)
(12)                        self.insert(button, -1)
(8)                     toolitem = Gtk.ToolItem()
(8)                     self.insert(toolitem, -1)
(8)                     label = Gtk.Label()
(8)                     label.set_markup(
(12)                        '<small>\N{NO-BREAK SPACE}\n\N{NO-BREAK SPACE}</small>')
(8)                     toolitem.set_expand(True)  # Push real message to the right.
(8)                     toolitem.add(label)
(8)                     toolitem = Gtk.ToolItem()
(8)                     self.insert(toolitem, -1)
(8)                     self.message = Gtk.Label()
(8)                     self.message.set_justify(Gtk.Justification.RIGHT)
(8)                     toolitem.add(self.message)
(8)                     self.show_all()
(8)                     _NavigationToolbar2GTK.__init__(self, canvas)
(4)                 def save_figure(self, *args):
(8)                     dialog = Gtk.FileChooserDialog(
(12)                        title="Save the figure",
(12)                        parent=self.canvas.get_toplevel(),
(12)                        action=Gtk.FileChooserAction.SAVE,
(12)                        buttons=(Gtk.STOCK_CANCEL, Gtk.ResponseType.CANCEL,
(21)                                 Gtk.STOCK_SAVE,   Gtk.ResponseType.OK),
(8)                     )
(8)                     for name, fmts \
(16)                            in self.canvas.get_supported_filetypes_grouped().items():
(12)                        ff = Gtk.FileFilter()
(12)                        ff.set_name(name)
(12)                        for fmt in fmts:
(16)                            ff.add_pattern(f'*.{fmt}')
(12)                        dialog.add_filter(ff)
(12)                        if self.canvas.get_default_filetype() in fmts:
(16)                            dialog.set_filter(ff)
(8)                     @functools.partial(dialog.connect, "notify::filter")
(8)                     def on_notify_filter(*args):
(12)                        name = dialog.get_filter().get_name()
(12)                        fmt = self.canvas.get_supported_filetypes_grouped()[name][0]
(12)                        dialog.set_current_name(
(16)                            str(Path(dialog.get_current_name()).with_suffix(f'.{fmt}')))
(8)                     dialog.set_current_folder(mpl.rcParams["savefig.directory"])
(8)                     dialog.set_current_name(self.canvas.get_default_filename())
(8)                     dialog.set_do_overwrite_confirmation(True)
(8)                     response = dialog.run()
(8)                     fname = dialog.get_filename()
(8)                     ff = dialog.get_filter()  # Doesn't autoadjust to filename :/
(8)                     fmt = self.canvas.get_supported_filetypes_grouped()[ff.get_name()][0]
(8)                     dialog.destroy()
(8)                     if response != Gtk.ResponseType.OK:
(12)                        return
(8)                     if mpl.rcParams['savefig.directory']:
(12)                        mpl.rcParams['savefig.directory'] = os.path.dirname(fname)
(8)                     try:
(12)                        self.canvas.figure.savefig(fname, format=fmt)
(8)                     except Exception as e:
(12)                        dialog = Gtk.MessageDialog(
(16)                            parent=self.canvas.get_toplevel(), message_format=str(e),
(16)                            type=Gtk.MessageType.ERROR, buttons=Gtk.ButtonsType.OK)
(12)                        dialog.run()
(12)                        dialog.destroy()
(0)             class ToolbarGTK3(ToolContainerBase, Gtk.Box):
(4)                 _icon_extension = '-symbolic.svg'
(4)                 def __init__(self, toolmanager):
(8)                     ToolContainerBase.__init__(self, toolmanager)
(8)                     Gtk.Box.__init__(self)
(8)                     self.set_property('orientation', Gtk.Orientation.HORIZONTAL)
(8)                     self._message = Gtk.Label()
(8)                     self._message.set_justify(Gtk.Justification.RIGHT)
```

```
(8)                 self.pack_end(self._message, False, False, 0)
(8)                 self.show_all()
(8)                 self._groups = {}
(8)                 self._toolitems = {}
(4)             def add_toolitem(self, name, group, position, image_file, description,
(21)                             toggle):
(8)                 if toggle:
(12)                    button = Gtk.ToggleToolButton()
(8)                 else:
(12)                    button = Gtk.ToolButton()
(8)                 button.set_label(name)
(8)                 if image_file is not None:
(12)                    image = Gtk.Image.new_from_gicon(
(16)                        Gio.Icon.new_for_string(image_file),
(16)                        Gtk.IconSize.LARGE_TOOLBAR)
(12)                    button.set_icon_widget(image)
(8)                 if position is None:
(12)                    position = -1
(8)                 self._add_button(button, group, position)
(8)                 signal = button.connect('clicked', self._call_tool, name)
(8)                 button.set_tooltip_text(description)
(8)                 button.show_all()
(8)                 self._toolitems.setdefault(name, [])
(8)                 self._toolitems[name].append((button, signal))
(4)             def _add_button(self, button, group, position):
(8)                 if group not in self._groups:
(12)                    if self._groups:
(16)                        self._add_separator()
(12)                    toolbar = Gtk.Toolbar()
(12)                    toolbar.set_style(Gtk.ToolbarStyle.ICONS)
(12)                    self.pack_start(toolbar, False, False, 0)
(12)                    toolbar.show_all()
(12)                    self._groups[group] = toolbar
(8)                 self._groups[group].insert(button, position)
(4)             def _call_tool(self, btn, name):
(8)                 self.trigger_tool(name)
(4)             def toggle_toolitem(self, name, toggled):
(8)                 if name not in self._toolitems:
(12)                    return
(8)                 for toolitem, signal in self._toolitems[name]:
(12)                    toolitem.handler_block(signal)
(12)                    toolitem.set_active(toggled)
(12)                    toolitem.handler_unblock(signal)
(4)             def remove_toolitem(self, name):
(8)                 for toolitem, _signal in self._toolitems.pop(name, []):
(12)                    for group in self._groups:
(16)                        if toolitem in self._groups[group]:
(20)                            self._groups[group].remove(toolitem)
(4)             def _add_separator(self):
(8)                 sep = Gtk.Separator()
(8)                 sep.set_property("orientation", Gtk.Orientation.VERTICAL)
(8)                 self.pack_start(sep, False, True, 0)
(8)                 sep.show_all()
(4)             def set_message(self, s):
(8)                 self._message.set_label(s)
(0)         @backend_tools._register_tool_class(FigureCanvasGTK3)
(0)         class SaveFigureGTK3(backend_tools.SaveFigureBase):
(4)             def trigger(self, *args, **kwargs):
(8)                 NavigationToolbar2GTK3.save_figure(
(12)                    self._make_classic_style_pseudo_toolbar())
(0)         @backend_tools._register_tool_class(FigureCanvasGTK3)
(0)         class HelpGTK3(backend_tools.ToolHelpBase):
(4)             def _normalize_shortcut(self, key):
(8)                 """
(8)                 Convert Matplotlib key presses to GTK+ accelerator identifiers.
(8)                 Related to `FigureCanvasGTK3._get_key`.
(8)                 """
(8)                 special = {
(12)                    'backspace': 'BackSpace',
```

```
380: (12)                        'pagedown': 'Page_Down',
381: (12)                        'pageup': 'Page_Up',
382: (12)                        'scroll_lock': 'Scroll_Lock',
383: (8)                    }
384: (8)                    parts = key.split('+')
385: (8)                    mods = ['<' + mod + '>' for mod in parts[:-1]]
386: (8)                    key = parts[-1]
387: (8)                    if key in special:
388: (12)                       key = special[key]
389: (8)                    elif len(key) > 1:
390: (12)                       key = key.capitalize()
391: (8)                    elif key.isupper():
392: (12)                       mods += ['<shift>']
393: (8)                    return ''.join(mods) + key
394: (4)                def _is_valid_shortcut(self, key):
395: (8)                    """
396: (8)                    Check for a valid shortcut to be displayed.
397: (8)                    - GTK will never send 'cmd+' (see `FigureCanvasGTK3._get_key`).
398: (8)                    - The shortcut window only shows keyboard shortcuts, not mouse
buttons.
399: (8)                    """
400: (8)                    return 'cmd+' not in key and not key.startswith('MouseButton.')
401: (4)                def _show_shortcuts_window(self):
402: (8)                    section = Gtk.ShortcutsSection()
403: (8)                    for name, tool in sorted(self.toolmanager.tools.items()):
404: (12)                       if not tool.description:
405: (16)                           continue
406: (12)                       group = Gtk.ShortcutsGroup()
407: (12)                       section.add(group)
408: (12)                       group.forall(lambda widget, data: widget.set_visible(False), None)
409: (12)                       shortcut = Gtk.ShortcutsShortcut(
410: (16)                           accelerator=' '.join(
411: (20)                               self._normalize_shortcut(key)
412: (20)                               for key in self.toolmanager.get_tool_keymap(name)
413: (20)                               if self._is_valid_shortcut(key)),
414: (16)                           title=tool.name,
415: (16)                           subtitle=tool.description)
416: (12)                       group.add(shortcut)
417: (8)                    window = Gtk.ShortcutsWindow(
418: (12)                       title='Help',
419: (12)                       modal=True,
420: (12)                       transient_for=self._figure.canvas.get_toplevel())
421: (8)                    section.show()  # Must be done explicitly before add!
422: (8)                    window.add(section)
423: (8)                    window.show_all()
424: (4)                def _show_shortcuts_dialog(self):
425: (8)                    dialog = Gtk.MessageDialog(
426: (12)                       self._figure.canvas.get_toplevel(),
427: (12)                       0, Gtk.MessageType.INFO, Gtk.ButtonsType.OK,
self._get_help_text(),
428: (12)                       title="Help")
429: (8)                    dialog.run()
430: (8)                    dialog.destroy()
431: (4)                def trigger(self, *args):
432: (8)                    if Gtk.check_version(3, 20, 0) is None:
433: (12)                       self._show_shortcuts_window()
434: (8)                    else:
435: (12)                       self._show_shortcuts_dialog()
436: (0)            @backend_tools._register_tool_class(FigureCanvasGTK3)
437: (0)            class ToolCopyToClipboardGTK3(backend_tools.ToolCopyToClipboardBase):
438: (4)                def trigger(self, *args, **kwargs):
439: (8)                    clipboard = Gtk.Clipboard.get(Gdk.SELECTION_CLIPBOARD)
440: (8)                    window = self.canvas.get_window()
441: (8)                    x, y, width, height = window.get_geometry()
442: (8)                    pb = Gdk.pixbuf_get_from_window(window, x, y, width, height)
443: (8)                    clipboard.set_image(pb)
444: (0)            Toolbar = ToolbarGTK3
445: (0)            backend_tools._register_tool_class(
446: (4)                FigureCanvasGTK3, _backend_gtk.ConfigureSubplotsGTK)
```

```
447: (0)            backend_tools._register_tool_class(
448: (4)                FigureCanvasGTK3, _backend_gtk.RubberbandGTK)
449: (0)            class FigureManagerGTK3(_FigureManagerGTK):
450: (4)                _toolbar2_class = NavigationToolbar2GTK3
451: (4)                _toolmanager_toolbar_class = ToolbarGTK3
452: (0)            @_BackendGTK.export
453: (0)            class _BackendGTK3(_BackendGTK):
454: (4)                FigureCanvas = FigureCanvasGTK3
455: (4)                FigureManager = FigureManagerGTK3
```

File 83 - backend_gtk3agg.py:

```
1: (0)              import numpy as np
2: (0)              from .. import cbook, transforms
3: (0)              from . import backend_agg, backend_gtk3
4: (0)              from .backend_gtk3 import GLib, Gtk, _BackendGTK3
5: (0)              import cairo  # Presence of cairo is already checked by _backend_gtk.
6: (0)              class FigureCanvasGTK3Agg(backend_agg.FigureCanvasAgg,
7: (26)                                       backend_gtk3.FigureCanvasGTK3):
8: (4)                  def __init__(self, figure):
9: (8)                      super().__init__(figure=figure)
10: (8)                     self._bbox_queue = []
11: (4)                 def on_draw_event(self, widget, ctx):
12: (8)                     if self._idle_draw_id:
13: (12)                        GLib.source_remove(self._idle_draw_id)
14: (12)                        self._idle_draw_id = 0
15: (12)                        self.draw()
16: (8)                     scale = self.device_pixel_ratio
17: (8)                     allocation = self.get_allocation()
18: (8)                     w = allocation.width * scale
19: (8)                     h = allocation.height * scale
20: (8)                     if not len(self._bbox_queue):
21: (12)                        Gtk.render_background(
22: (16)                            self.get_style_context(), ctx,
23: (16)                            allocation.x, allocation.y,
24: (16)                            allocation.width, allocation.height)
25: (12)                        bbox_queue = [transforms.Bbox([[0, 0], [w, h]])]
26: (8)                     else:
27: (12)                        bbox_queue = self._bbox_queue
28: (8)                     for bbox in bbox_queue:
29: (12)                        x = int(bbox.x0)
30: (12)                        y = h - int(bbox.y1)
31: (12)                        width = int(bbox.x1) - int(bbox.x0)
32: (12)                        height = int(bbox.y1) - int(bbox.y0)
33: (12)                        buf = cbook._unmultiplied_rgba8888_to_premultiplied_argb32(
34: (16)                            np.asarray(self.copy_from_bbox(bbox)))
35: (12)                        image = cairo.ImageSurface.create_for_data(
36: (16)                            buf.ravel().data, cairo.FORMAT_ARGB32, width, height)
37: (12)                        image.set_device_scale(scale, scale)
38: (12)                        ctx.set_source_surface(image, x / scale, y / scale)
39: (12)                        ctx.paint()
40: (8)                     if len(self._bbox_queue):
41: (12)                        self._bbox_queue = []
42: (8)                     return False
43: (4)                 def blit(self, bbox=None):
44: (8)                     if bbox is None:
45: (12)                        bbox = self.figure.bbox
46: (8)                     scale = self.device_pixel_ratio
47: (8)                     allocation = self.get_allocation()
48: (8)                     x = int(bbox.x0 / scale)
49: (8)                     y = allocation.height - int(bbox.y1 / scale)
50: (8)                     width = (int(bbox.x1) - int(bbox.x0)) // scale
51: (8)                     height = (int(bbox.y1) - int(bbox.y0)) // scale
52: (8)                     self._bbox_queue.append(bbox)
53: (8)                     self.queue_draw_area(x, y, width, height)
54: (0)             @_BackendGTK3.export
55: (0)             class _BackendGTK3Cairo(_BackendGTK3):
```

```
56: (4)                 FigureCanvas = FigureCanvasGTK3Agg

----------------------------------------

File 84 - backend_gtk3cairo.py:

1: (0)              from contextlib import nullcontext
2: (0)              from .backend_cairo import FigureCanvasCairo
3: (0)              from .backend_gtk3 import GLib, Gtk, FigureCanvasGTK3, _BackendGTK3
4: (0)              class FigureCanvasGTK3Cairo(FigureCanvasCairo, FigureCanvasGTK3):
5: (4)                  def on_draw_event(self, widget, ctx):
6: (8)                      if self._idle_draw_id:
7: (12)                         GLib.source_remove(self._idle_draw_id)
8: (12)                         self._idle_draw_id = 0
9: (12)                         self.draw()
10: (8)                     with (self.toolbar._wait_cursor_for_draw_cm() if self.toolbar
11: (14)                          else nullcontext()):
12: (12)                        allocation = self.get_allocation()
13: (12)                        Gtk.render_background(
14: (16)                            self.get_style_context(), ctx,
15: (16)                            0, 0, allocation.width, allocation.height)
16: (12)                        scale = self.device_pixel_ratio
17: (12)                        ctx.scale(1 / scale, 1 / scale)
18: (12)                        self._renderer.set_context(ctx)
19: (12)                        self._renderer.width = allocation.width * scale
20: (12)                        self._renderer.height = allocation.height * scale
21: (12)                        self._renderer.dpi = self.figure.dpi
22: (12)                        self.figure.draw(self._renderer)
23: (0)             @_BackendGTK3.export
24: (0)             class _BackendGTK3Cairo(_BackendGTK3):
25: (4)                 FigureCanvas = FigureCanvasGTK3Cairo

----------------------------------------

File 85 - backend_gtk4.py:

1: (0)              import functools
2: (0)              import io
3: (0)              import os
4: (0)              import matplotlib as mpl
5: (0)              from matplotlib import _api, backend_tools, cbook
6: (0)              from matplotlib.backend_bases import (
7: (4)                  ToolContainerBase, KeyEvent, LocationEvent, MouseEvent, ResizeEvent,
8: (4)                  CloseEvent)
9: (0)              try:
10: (4)                 import gi
11: (0)             except ImportError as err:
12: (4)                 raise ImportError("The GTK4 backends require PyGObject") from err
13: (0)             try:
14: (4)                 gi.require_version("Gtk", "4.0")
15: (0)             except ValueError as e:
16: (4)                 raise ImportError(e) from e
17: (0)             from gi.repository import Gio, GLib, Gtk, Gdk, GdkPixbuf
18: (0)             from . import _backend_gtk
19: (0)             from ._backend_gtk import (  # noqa: F401 # pylint: disable=W0611
20: (4)                 _BackendGTK, _FigureCanvasGTK, _FigureManagerGTK, _NavigationToolbar2GTK,
21: (4)                 TimerGTK as TimerGTK4,
22: (0)             )
23: (0)             class FigureCanvasGTK4(_FigureCanvasGTK, Gtk.DrawingArea):
24: (4)                 required_interactive_framework = "gtk4"
25: (4)                 supports_blit = False
26: (4)                 manager_class = _api.classproperty(lambda cls: FigureManagerGTK4)
27: (4)                 def __init__(self, figure=None):
28: (8)                     super().__init__(figure=figure)
29: (8)                     self.set_hexpand(True)
30: (8)                     self.set_vexpand(True)
31: (8)                     self._idle_draw_id = 0
32: (8)                     self._rubberband_rect = None
33: (8)                     self.set_draw_func(self._draw_func)
```

```
34: (8)                     self.connect('resize', self.resize_event)
35: (8)                     self.connect('notify::scale-factor', self._update_device_pixel_ratio)
36: (8)                     click = Gtk.GestureClick()
37: (8)                     click.set_button(0)  # All buttons.
38: (8)                     click.connect('pressed', self.button_press_event)
39: (8)                     click.connect('released', self.button_release_event)
40: (8)                     self.add_controller(click)
41: (8)                     key = Gtk.EventControllerKey()
42: (8)                     key.connect('key-pressed', self.key_press_event)
43: (8)                     key.connect('key-released', self.key_release_event)
44: (8)                     self.add_controller(key)
45: (8)                     motion = Gtk.EventControllerMotion()
46: (8)                     motion.connect('motion', self.motion_notify_event)
47: (8)                     motion.connect('enter', self.enter_notify_event)
48: (8)                     motion.connect('leave', self.leave_notify_event)
49: (8)                     self.add_controller(motion)
50: (8)                     scroll = Gtk.EventControllerScroll.new(
51: (12)                        Gtk.EventControllerScrollFlags.VERTICAL)
52: (8)                     scroll.connect('scroll', self.scroll_event)
53: (8)                     self.add_controller(scroll)
54: (8)                     self.set_focusable(True)
55: (8)                     css = Gtk.CssProvider()
56: (8)                     style = '.matplotlib-canvas { background-color: white; }'
57: (8)                     if Gtk.check_version(4, 9, 3) is None:
58: (12)                        css.load_from_data(style, -1)
59: (8)                     else:
60: (12)                        css.load_from_data(style.encode('utf-8'))
61: (8)                     style_ctx = self.get_style_context()
62: (8)                     style_ctx.add_provider(css, Gtk.STYLE_PROVIDER_PRIORITY_APPLICATION)
63: (8)                     style_ctx.add_class("matplotlib-canvas")
64: (4)                 def destroy(self):
65: (8)                     CloseEvent("close_event", self)._process()
66: (4)                 def set_cursor(self, cursor):
67: (8)                     self.set_cursor_from_name(_backend_gtk.mpl_to_gtk_cursor_name(cursor))
68: (4)                 def _mpl_coords(self, xy=None):
69: (8)                     """
70: (8)                     Convert the *xy* position of a GTK event, or of the current cursor
71: (8)                     position if *xy* is None, to Matplotlib coordinates.
72: (8)                     GTK use logical pixels, but the figure is scaled to physical pixels
for
73: (8)                     rendering.  Transform to physical pixels so that all of the down-
stream
74: (8)                     transforms work as expected.
75: (8)                     Also, the origin is different and needs to be corrected.
76: (8)                     """
77: (8)                     if xy is None:
78: (12)                        surface = self.get_native().get_surface()
79: (12)                        is_over, x, y, mask = surface.get_device_position(
80: (16)                            self.get_display().get_default_seat().get_pointer())
81: (8)                     else:
82: (12)                        x, y = xy
83: (8)                     x = x * self.device_pixel_ratio
84: (8)                     y = self.figure.bbox.height - y * self.device_pixel_ratio
85: (8)                     return x, y
86: (4)                 def scroll_event(self, controller, dx, dy):
87: (8)                     MouseEvent(
88: (12)                        "scroll_event", self, *self._mpl_coords(), step=dy,
89: (12)                        modifiers=self._mpl_modifiers(controller),
90: (8)                     )._process()
91: (8)                     return True
92: (4)                 def button_press_event(self, controller, n_press, x, y):
93: (8)                     MouseEvent(
94: (12)                        "button_press_event", self, *self._mpl_coords((x, y)),
95: (12)                        controller.get_current_button(),
96: (12)                        modifiers=self._mpl_modifiers(controller),
97: (8)                     )._process()
98: (8)                     self.grab_focus()
99: (4)                 def button_release_event(self, controller, n_press, x, y):
100: (8)                    MouseEvent(
```

```
101: (12)                        "button_release_event", self, *self._mpl_coords((x, y)),
102: (12)                        controller.get_current_button(),
103: (12)                        modifiers=self._mpl_modifiers(controller),
104: (8)                    )._process()
105: (4)                def key_press_event(self, controller, keyval, keycode, state):
106: (8)                    KeyEvent(
107: (12)                        "key_press_event", self, self._get_key(keyval, keycode, state),
108: (12)                        *self._mpl_coords(),
109: (8)                    )._process()
110: (8)                    return True
111: (4)                def key_release_event(self, controller, keyval, keycode, state):
112: (8)                    KeyEvent(
113: (12)                        "key_release_event", self, self._get_key(keyval, keycode, state),
114: (12)                        *self._mpl_coords(),
115: (8)                    )._process()
116: (8)                    return True
117: (4)                def motion_notify_event(self, controller, x, y):
118: (8)                    MouseEvent(
119: (12)                        "motion_notify_event", self, *self._mpl_coords((x, y)),
120: (12)                        modifiers=self._mpl_modifiers(controller),
121: (8)                    )._process()
122: (4)                def enter_notify_event(self, controller, x, y):
123: (8)                    LocationEvent(
124: (12)                        "figure_enter_event", self, *self._mpl_coords((x, y)),
125: (12)                        modifiers=self._mpl_modifiers(),
126: (8)                    )._process()
127: (4)                def leave_notify_event(self, controller):
128: (8)                    LocationEvent(
129: (12)                        "figure_leave_event", self, *self._mpl_coords(),
130: (12)                        modifiers=self._mpl_modifiers(),
131: (8)                    )._process()
132: (4)                def resize_event(self, area, width, height):
133: (8)                    self._update_device_pixel_ratio()
134: (8)                    dpi = self.figure.dpi
135: (8)                    winch = width * self.device_pixel_ratio / dpi
136: (8)                    hinch = height * self.device_pixel_ratio / dpi
137: (8)                    self.figure.set_size_inches(winch, hinch, forward=False)
138: (8)                    ResizeEvent("resize_event", self)._process()
139: (8)                    self.draw_idle()
140: (4)                def _mpl_modifiers(self, controller=None):
141: (8)                    if controller is None:
142: (12)                        surface = self.get_native().get_surface()
143: (12)                        is_over, x, y, event_state = surface.get_device_position(
144: (16)                            self.get_display().get_default_seat().get_pointer())
145: (8)                    else:
146: (12)                        event_state = controller.get_current_event_state()
147: (8)                    mod_table = [
148: (12)                        ("ctrl", Gdk.ModifierType.CONTROL_MASK),
149: (12)                        ("alt", Gdk.ModifierType.ALT_MASK),
150: (12)                        ("shift", Gdk.ModifierType.SHIFT_MASK),
151: (12)                        ("super", Gdk.ModifierType.SUPER_MASK),
152: (8)                    ]
153: (8)                    return [name for name, mask in mod_table if event_state & mask]
154: (4)                def _get_key(self, keyval, keycode, state):
155: (8)                    unikey = chr(Gdk.keyval_to_unicode(keyval))
156: (8)                    key = cbook._unikey_or_keysym_to_mplkey(
157: (12)                        unikey,
158: (12)                        Gdk.keyval_name(keyval))
159: (8)                    modifiers = [
160: (12)                        ("ctrl", Gdk.ModifierType.CONTROL_MASK, "control"),
161: (12)                        ("alt", Gdk.ModifierType.ALT_MASK, "alt"),
162: (12)                        ("shift", Gdk.ModifierType.SHIFT_MASK, "shift"),
163: (12)                        ("super", Gdk.ModifierType.SUPER_MASK, "super"),
164: (8)                    ]
165: (8)                    mods = [
166: (12)                        mod for mod, mask, mod_key in modifiers
167: (12)                        if (mod_key != key and state & mask
168: (16)                            and not (mod == "shift" and unikey.isprintable()))]
169: (8)                    return "+".join([*mods, key])
```

```
(4)                 def _update_device_pixel_ratio(self, *args, **kwargs):
(8)                     if self._set_device_pixel_ratio(self.get_scale_factor()):
(12)                        self.draw()
(4)                 def _draw_rubberband(self, rect):
(8)                     self._rubberband_rect = rect
(8)                     self.queue_draw()
(4)                 def _draw_func(self, drawing_area, ctx, width, height):
(8)                     self.on_draw_event(self, ctx)
(8)                     self._post_draw(self, ctx)
(4)                 def _post_draw(self, widget, ctx):
(8)                     if self._rubberband_rect is None:
(12)                        return
(8)                     lw = 1
(8)                     dash = 3
(8)                     x0, y0, w, h = (dim / self.device_pixel_ratio
(24)                                    for dim in self._rubberband_rect)
(8)                     x1 = x0 + w
(8)                     y1 = y0 + h
(8)                     ctx.move_to(x0, y0)
(8)                     ctx.line_to(x0, y1)
(8)                     ctx.move_to(x0, y0)
(8)                     ctx.line_to(x1, y0)
(8)                     ctx.move_to(x0, y1)
(8)                     ctx.line_to(x1, y1)
(8)                     ctx.move_to(x1, y0)
(8)                     ctx.line_to(x1, y1)
(8)                     ctx.set_antialias(1)
(8)                     ctx.set_line_width(lw)
(8)                     ctx.set_dash((dash, dash), 0)
(8)                     ctx.set_source_rgb(0, 0, 0)
(8)                     ctx.stroke_preserve()
(8)                     ctx.set_dash((dash, dash), dash)
(8)                     ctx.set_source_rgb(1, 1, 1)
(8)                     ctx.stroke()
(4)                 def on_draw_event(self, widget, ctx):
(8)                     pass
(4)                 def draw(self):
(8)                     if self.is_drawable():
(12)                        self.queue_draw()
(4)                 def draw_idle(self):
(8)                     if self._idle_draw_id != 0:
(12)                        return
(8)                     def idle_draw(*args):
(12)                        try:
(16)                            self.draw()
(12)                        finally:
(16)                            self._idle_draw_id = 0
(12)                        return False
(8)                     self._idle_draw_id = GLib.idle_add(idle_draw)
(4)                 def flush_events(self):
(8)                     context = GLib.MainContext.default()
(8)                     while context.pending():
(12)                        context.iteration(True)
(0)             class NavigationToolbar2GTK4(_NavigationToolbar2GTK, Gtk.Box):
(4)                 def __init__(self, canvas):
(8)                     Gtk.Box.__init__(self)
(8)                     self.add_css_class('toolbar')
(8)                     self._gtk_ids = {}
(8)                     for text, tooltip_text, image_file, callback in self.toolitems:
(12)                        if text is None:
(16)                            self.append(Gtk.Separator())
(16)                            continue
(12)                        image = Gtk.Image.new_from_gicon(
(16)                            Gio.Icon.new_for_string(
(20)                                str(cbook._get_data_path('images',
(45)                                                        f'{image_file}-symbolic.svg'))))
(12)                        self._gtk_ids[text] = button = (
(16)                            Gtk.ToggleButton() if callback in ['zoom', 'pan'] else
(16)                            Gtk.Button())
```

```
(12)                        button.set_child(image)
(12)                        button.add_css_class('flat')
(12)                        button.add_css_class('image-button')
(12)                        button._signal_handler = button.connect(
(16)                            'clicked', getattr(self, callback))
(12)                        button.set_tooltip_text(tooltip_text)
(12)                        self.append(button)
(8)                     label = Gtk.Label()
(8)                     label.set_markup(
(12)                        '<small>\N{NO-BREAK SPACE}\n\N{NO-BREAK SPACE}</small>')
(8)                     label.set_hexpand(True)  # Push real message to the right.
(8)                     self.append(label)
(8)                     self.message = Gtk.Label()
(8)                     self.message.set_justify(Gtk.Justification.RIGHT)
(8)                     self.append(self.message)
(8)                     _NavigationToolbar2GTK.__init__(self, canvas)
(4)                 def save_figure(self, *args):
(8)                     dialog = Gtk.FileChooserNative(
(12)                        title='Save the figure',
(12)                        transient_for=self.canvas.get_root(),
(12)                        action=Gtk.FileChooserAction.SAVE,
(12)                        modal=True)
(8)                     self._save_dialog = dialog  # Must keep a reference.
(8)                     ff = Gtk.FileFilter()
(8)                     ff.set_name('All files')
(8)                     ff.add_pattern('*')
(8)                     dialog.add_filter(ff)
(8)                     dialog.set_filter(ff)
(8)                     formats = []
(8)                     default_format = None
(8)                     for i, (name, fmts) in enumerate(
(16)                            self.canvas.get_supported_filetypes_grouped().items()):
(12)                        ff = Gtk.FileFilter()
(12)                        ff.set_name(name)
(12)                        for fmt in fmts:
(16)                            ff.add_pattern(f'*.{fmt}')
(12)                        dialog.add_filter(ff)
(12)                        formats.append(name)
(12)                        if self.canvas.get_default_filetype() in fmts:
(16)                            default_format = i
(8)                     formats = [formats[default_format], *formats[:default_format],
(19)                               *formats[default_format+1:]]
(8)                     dialog.add_choice('format', 'File format', formats, formats)
(8)                     dialog.set_choice('format', formats[0])
(8)                     dialog.set_current_folder(Gio.File.new_for_path(
(12)                        os.path.expanduser(mpl.rcParams['savefig.directory'])))
(8)                     dialog.set_current_name(self.canvas.get_default_filename())
(8)                     @functools.partial(dialog.connect, 'response')
(8)                     def on_response(dialog, response):
(12)                        file = dialog.get_file()
(12)                        fmt = dialog.get_choice('format')
(12)                        fmt = self.canvas.get_supported_filetypes_grouped()[fmt][0]
(12)                        dialog.destroy()
(12)                        self._save_dialog = None
(12)                        if response != Gtk.ResponseType.ACCEPT:
(16)                            return
(12)                        if mpl.rcParams['savefig.directory']:
(16)                            parent = file.get_parent()
(16)                            mpl.rcParams['savefig.directory'] = parent.get_path()
(12)                        try:
(16)                            self.canvas.figure.savefig(file.get_path(), format=fmt)
(12)                        except Exception as e:
(16)                            msg = Gtk.MessageDialog(
(20)                                transient_for=self.canvas.get_root(),
(20)                                message_type=Gtk.MessageType.ERROR,
(20)                                buttons=Gtk.ButtonsType.OK, modal=True,
(20)                                text=str(e))
(16)                            msg.show()
(8)                     dialog.show()
```

```
(0)  class ToolbarGTK4(ToolContainerBase, Gtk.Box):
(4)      _icon_extension = '-symbolic.svg'
(4)      def __init__(self, toolmanager):
(8)          ToolContainerBase.__init__(self, toolmanager)
(8)          Gtk.Box.__init__(self)
(8)          self.set_property('orientation', Gtk.Orientation.HORIZONTAL)
(8)          self._tool_box = Gtk.Box()
(8)          self.append(self._tool_box)
(8)          self._groups = {}
(8)          self._toolitems = {}
(8)          label = Gtk.Label()
(8)          label.set_markup(
(12)             '<small>\N{NO-BREAK SPACE}\n\N{NO-BREAK SPACE}</small>')
(8)          label.set_hexpand(True)  # Push real message to the right.
(8)          self.append(label)
(8)          self._message = Gtk.Label()
(8)          self._message.set_justify(Gtk.Justification.RIGHT)
(8)          self.append(self._message)
(4)      def add_toolitem(self, name, group, position, image_file, description,
(21)                      toggle):
(8)          if toggle:
(12)             button = Gtk.ToggleButton()
(8)          else:
(12)             button = Gtk.Button()
(8)          button.set_label(name)
(8)          button.add_css_class('flat')
(8)          if image_file is not None:
(12)             image = Gtk.Image.new_from_gicon(
(16)                 Gio.Icon.new_for_string(image_file))
(12)             button.set_child(image)
(12)             button.add_css_class('image-button')
(8)          if position is None:
(12)             position = -1
(8)          self._add_button(button, group, position)
(8)          signal = button.connect('clicked', self._call_tool, name)
(8)          button.set_tooltip_text(description)
(8)          self._toolitems.setdefault(name, [])
(8)          self._toolitems[name].append((button, signal))
(4)      def _find_child_at_position(self, group, position):
(8)          children = [None]
(8)          child = self._groups[group].get_first_child()
(8)          while child is not None:
(12)             children.append(child)
(12)             child = child.get_next_sibling()
(8)          return children[position]
(4)      def _add_button(self, button, group, position):
(8)          if group not in self._groups:
(12)             if self._groups:
(16)                 self._add_separator()
(12)             group_box = Gtk.Box()
(12)             self._tool_box.append(group_box)
(12)             self._groups[group] = group_box
(8)          self._groups[group].insert_child_after(
(12)             button, self._find_child_at_position(group, position))
(4)      def _call_tool(self, btn, name):
(8)          self.trigger_tool(name)
(4)      def toggle_toolitem(self, name, toggled):
(8)          if name not in self._toolitems:
(12)             return
(8)          for toolitem, signal in self._toolitems[name]:
(12)             toolitem.handler_block(signal)
(12)             toolitem.set_active(toggled)
(12)             toolitem.handler_unblock(signal)
(4)      def remove_toolitem(self, name):
(8)          for toolitem, _signal in self._toolitems.pop(name, []):
(12)             for group in self._groups:
(16)                 if toolitem in self._groups[group]:
(20)                     self._groups[group].remove(toolitem)
(4)      def _add_separator(self):
```

```
377: (8)                    sep = Gtk.Separator()
378: (8)                    sep.set_property("orientation", Gtk.Orientation.VERTICAL)
379: (8)                    self._tool_box.append(sep)
380: (4)                def set_message(self, s):
381: (8)                    self._message.set_label(s)
382: (0)            @backend_tools._register_tool_class(FigureCanvasGTK4)
383: (0)            class SaveFigureGTK4(backend_tools.SaveFigureBase):
384: (4)                def trigger(self, *args, **kwargs):
385: (8)                    NavigationToolbar2GTK4.save_figure(
386: (12)                       self._make_classic_style_pseudo_toolbar())
387: (0)            @backend_tools._register_tool_class(FigureCanvasGTK4)
388: (0)            class HelpGTK4(backend_tools.ToolHelpBase):
389: (4)                def _normalize_shortcut(self, key):
390: (8)                    """
391: (8)                    Convert Matplotlib key presses to GTK+ accelerator identifiers.
392: (8)                    Related to `FigureCanvasGTK4._get_key`.
393: (8)                    """
394: (8)                    special = {
395: (12)                       'backspace': 'BackSpace',
396: (12)                       'pagedown': 'Page_Down',
397: (12)                       'pageup': 'Page_Up',
398: (12)                       'scroll_lock': 'Scroll_Lock',
399: (8)                    }
400: (8)                    parts = key.split('+')
401: (8)                    mods = ['<' + mod + '>' for mod in parts[:-1]]
402: (8)                    key = parts[-1]
403: (8)                    if key in special:
404: (12)                       key = special[key]
405: (8)                    elif len(key) > 1:
406: (12)                       key = key.capitalize()
407: (8)                    elif key.isupper():
408: (12)                       mods += ['<shift>']
409: (8)                    return ''.join(mods) + key
410: (4)                def _is_valid_shortcut(self, key):
411: (8)                    """
412: (8)                    Check for a valid shortcut to be displayed.
413: (8)                    - GTK will never send 'cmd+' (see `FigureCanvasGTK4._get_key`).
414: (8)                    - The shortcut window only shows keyboard shortcuts, not mouse
buttons.
415: (8)                    """
416: (8)                    return 'cmd+' not in key and not key.startswith('MouseButton.')
417: (4)                def trigger(self, *args):
418: (8)                    section = Gtk.ShortcutsSection()
419: (8)                    for name, tool in sorted(self.toolmanager.tools.items()):
420: (12)                       if not tool.description:
421: (16)                           continue
422: (12)                       group = Gtk.ShortcutsGroup()
423: (12)                       section.append(group)
424: (12)                       child = group.get_first_child()
425: (12)                       while child is not None:
426: (16)                           child.set_visible(False)
427: (16)                           child = child.get_next_sibling()
428: (12)                       shortcut = Gtk.ShortcutsShortcut(
429: (16)                           accelerator=' '.join(
430: (20)                               self._normalize_shortcut(key)
431: (20)                               for key in self.toolmanager.get_tool_keymap(name)
432: (20)                               if self._is_valid_shortcut(key)),
433: (16)                           title=tool.name,
434: (16)                           subtitle=tool.description)
435: (12)                       group.append(shortcut)
436: (8)                    window = Gtk.ShortcutsWindow(
437: (12)                       title='Help',
438: (12)                       modal=True,
439: (12)                       transient_for=self._figure.canvas.get_root())
440: (8)                    window.set_child(section)
441: (8)                    window.show()
442: (0)            @backend_tools._register_tool_class(FigureCanvasGTK4)
443: (0)            class ToolCopyToClipboardGTK4(backend_tools.ToolCopyToClipboardBase):
444: (4)                def trigger(self, *args, **kwargs):
```

```
445: (8)                    with io.BytesIO() as f:
446: (12)                       self.canvas.print_rgba(f)
447: (12)                       w, h = self.canvas.get_width_height()
448: (12)                       pb = GdkPixbuf.Pixbuf.new_from_data(f.getbuffer(),
449: (48)                                                           GdkPixbuf.Colorspace.RGB,
True,
450: (48)                                                           8, w, h, w*4)
451: (8)                    clipboard = self.canvas.get_clipboard()
452: (8)                    clipboard.set(pb)
453: (0)            backend_tools._register_tool_class(
454: (4)                FigureCanvasGTK4, _backend_gtk.ConfigureSubplotsGTK)
455: (0)            backend_tools._register_tool_class(
456: (4)                FigureCanvasGTK4, _backend_gtk.RubberbandGTK)
457: (0)            Toolbar = ToolbarGTK4
458: (0)            class FigureManagerGTK4(_FigureManagerGTK):
459: (4)                _toolbar2_class = NavigationToolbar2GTK4
460: (4)                _toolmanager_toolbar_class = ToolbarGTK4
461: (0)            @_BackendGTK.export
462: (0)            class _BackendGTK4(_BackendGTK):
463: (4)                FigureCanvas = FigureCanvasGTK4
464: (4)                FigureManager = FigureManagerGTK4

----------------------------------------

File 86 - backend_gtk4agg.py:

1: (0)              import numpy as np
2: (0)              from .. import cbook
3: (0)              from . import backend_agg, backend_gtk4
4: (0)              from .backend_gtk4 import GLib, Gtk, _BackendGTK4
5: (0)              import cairo  # Presence of cairo is already checked by _backend_gtk.
6: (0)              class FigureCanvasGTK4Agg(backend_agg.FigureCanvasAgg,
7: (26)                                       backend_gtk4.FigureCanvasGTK4):
8: (4)                  def on_draw_event(self, widget, ctx):
9: (8)                      if self._idle_draw_id:
10: (12)                        GLib.source_remove(self._idle_draw_id)
11: (12)                        self._idle_draw_id = 0
12: (12)                        self.draw()
13: (8)                     scale = self.device_pixel_ratio
14: (8)                     allocation = self.get_allocation()
15: (8)                     Gtk.render_background(
16: (12)                        self.get_style_context(), ctx,
17: (12)                        allocation.x, allocation.y,
18: (12)                        allocation.width, allocation.height)
19: (8)                     buf = cbook._unmultiplied_rgba8888_to_premultiplied_argb32(
20: (12)                        np.asarray(self.get_renderer().buffer_rgba()))
21: (8)                     height, width, _ = buf.shape
22: (8)                     image = cairo.ImageSurface.create_for_data(
23: (12)                        buf.ravel().data, cairo.FORMAT_ARGB32, width, height)
24: (8)                     image.set_device_scale(scale, scale)
25: (8)                     ctx.set_source_surface(image, 0, 0)
26: (8)                     ctx.paint()
27: (8)                     return False
28: (0)             @_BackendGTK4.export
29: (0)             class _BackendGTK4Agg(_BackendGTK4):
30: (4)                 FigureCanvas = FigureCanvasGTK4Agg

----------------------------------------

File 87 - backend_gtk4cairo.py:

1: (0)              from contextlib import nullcontext
2: (0)              from .backend_cairo import FigureCanvasCairo
3: (0)              from .backend_gtk4 import GLib, Gtk, FigureCanvasGTK4, _BackendGTK4
4: (0)              class FigureCanvasGTK4Cairo(FigureCanvasCairo, FigureCanvasGTK4):
5: (4)                  def _set_device_pixel_ratio(self, ratio):
6: (8)                      return False
7: (4)                  def on_draw_event(self, widget, ctx):
8: (8)                      if self._idle_draw_id:
```

```
9: (12)                         GLib.source_remove(self._idle_draw_id)
10: (12)                        self._idle_draw_id = 0
11: (12)                        self.draw()
12: (8)                     with (self.toolbar._wait_cursor_for_draw_cm() if self.toolbar
13: (14)                          else nullcontext()):
14: (12)                        self._renderer.set_context(ctx)
15: (12)                        allocation = self.get_allocation()
16: (12)                        Gtk.render_background(
17: (16)                            self.get_style_context(), ctx,
18: (16)                            allocation.x, allocation.y,
19: (16)                            allocation.width, allocation.height)
20: (12)                        self.figure.draw(self._renderer)
21: (0)             @_BackendGTK4.export
22: (0)             class _BackendGTK4Cairo(_BackendGTK4):
23: (4)                 FigureCanvas = FigureCanvasGTK4Cairo

----------------------------------------

File 88 - backend_macosx.py:

1: (0)              import os
2: (0)              import matplotlib as mpl
3: (0)              from matplotlib import _api, cbook
4: (0)              from matplotlib._pylab_helpers import Gcf
5: (0)              from . import _macosx
6: (0)              from .backend_agg import FigureCanvasAgg
7: (0)              from matplotlib.backend_bases import (
8: (4)                  _Backend, FigureCanvasBase, FigureManagerBase, NavigationToolbar2,
9: (4)                  ResizeEvent, TimerBase, _allow_interrupt)
10: (0)             class TimerMac(_macosx.Timer, TimerBase):
11: (4)                 """Subclass of `.TimerBase` using CFRunLoop timer events."""
12: (0)             def _allow_interrupt_macos():
13: (4)                 """A context manager that allows terminating a plot by sending a
SIGINT."""
14: (4)                 return _allow_interrupt(
15: (8)                     lambda rsock: _macosx.wake_on_fd_write(rsock.fileno()), _macosx.stop)
16: (0)             class FigureCanvasMac(FigureCanvasAgg, _macosx.FigureCanvas,
FigureCanvasBase):
17: (4)                 required_interactive_framework = "macosx"
18: (4)                 _timer_cls = TimerMac
19: (4)                 manager_class = _api.classproperty(lambda cls: FigureManagerMac)
20: (4)                 def __init__(self, figure):
21: (8)                     super().__init__(figure=figure)
22: (8)                     self._draw_pending = False
23: (8)                     self._is_drawing = False
24: (8)                     self._timers = set()
25: (4)                 def draw(self):
26: (8)                     """Render the figure and update the macosx canvas."""
27: (8)                     if self._is_drawing:
28: (12)                        return
29: (8)                     with cbook._setattr_cm(self, _is_drawing=True):
30: (12)                        super().draw()
31: (8)                     self.update()
32: (4)                 def draw_idle(self):
33: (8)                     if not (getattr(self, '_draw_pending', False) or
34: (16)                            getattr(self, '_is_drawing', False)):
35: (12)                        self._draw_pending = True
36: (12)                        self._single_shot_timer(self._draw_idle)
37: (4)                 def _single_shot_timer(self, callback):
38: (8)                     """Add a single shot timer with the given callback"""
39: (8)                     def callback_func(callback, timer):
40: (12)                        callback()
41: (12)                        self._timers.remove(timer)
42: (8)                     timer = self.new_timer(interval=0)
43: (8)                     timer.single_shot = True
44: (8)                     timer.add_callback(callback_func, callback, timer)
45: (8)                     self._timers.add(timer)
46: (8)                     timer.start()
47: (4)                 def _draw_idle(self):
```

```
48: (8)                     """
49: (8)                     Draw method for singleshot timer
50: (8)                     This draw method can be added to a singleshot timer, which can
51: (8)                     accumulate draws while the eventloop is spinning. This method will
52: (8)                     then only draw the first time and short-circuit the others.
53: (8)                     """
54: (8)                     with self._idle_draw_cntx():
55: (12)                        if not self._draw_pending:
56: (16)                            return
57: (12)                        self._draw_pending = False
58: (12)                        self.draw()
59: (4)                 def blit(self, bbox=None):
60: (8)                     super().blit(bbox)
61: (8)                     self.update()
62: (4)                 def resize(self, width, height):
63: (8)                     scale = self.figure.dpi / self.device_pixel_ratio
64: (8)                     width /= scale
65: (8)                     height /= scale
66: (8)                     self.figure.set_size_inches(width, height, forward=False)
67: (8)                     ResizeEvent("resize_event", self)._process()
68: (8)                     self.draw_idle()
69: (4)                 def start_event_loop(self, timeout=0):
70: (8)                     with _allow_interrupt_macos():
71: (12)                        self._start_event_loop(timeout=timeout)  # Forward to ObjC
implementation.
72: (0)             class NavigationToolbar2Mac(_macosx.NavigationToolbar2, NavigationToolbar2):
73: (4)                 def __init__(self, canvas):
74: (8)                     data_path = cbook._get_data_path('images')
75: (8)                     _, tooltips, image_names, _ = zip(*NavigationToolbar2.toolitems)
76: (8)                     _macosx.NavigationToolbar2.__init__(
77: (12)                        self, canvas,
78: (12)                        tuple(str(data_path / image_name) + ".pdf"
79: (18)                              for image_name in image_names if image_name is not None),
80: (12)                        tuple(tooltip for tooltip in tooltips if tooltip is not None))
81: (8)                     NavigationToolbar2.__init__(self, canvas)
82: (4)                 def draw_rubberband(self, event, x0, y0, x1, y1):
83: (8)                     self.canvas.set_rubberband(int(x0), int(y0), int(x1), int(y1))
84: (4)                 def remove_rubberband(self):
85: (8)                     self.canvas.remove_rubberband()
86: (4)                 def save_figure(self, *args):
87: (8)                     directory = os.path.expanduser(mpl.rcParams['savefig.directory'])
88: (8)                     filename = _macosx.choose_save_file('Save the figure',
89: (44)                                                         directory,
90: (44)
self.canvas.get_default_filename())
91: (8)                     if filename is None:  # Cancel
92: (12)                        return
93: (8)                     if mpl.rcParams['savefig.directory']:
94: (12)                        mpl.rcParams['savefig.directory'] = os.path.dirname(filename)
95: (8)                     self.canvas.figure.savefig(filename)
96: (0)             class FigureManagerMac(_macosx.FigureManager, FigureManagerBase):
97: (4)                 _toolbar2_class = NavigationToolbar2Mac
98: (4)                 def __init__(self, canvas, num):
99: (8)                     self._shown = False
100: (8)                    _macosx.FigureManager.__init__(self, canvas)
101: (8)                    icon_path = str(cbook._get_data_path('images/matplotlib.pdf'))
102: (8)                    _macosx.FigureManager.set_icon(icon_path)
103: (8)                    FigureManagerBase.__init__(self, canvas, num)
104: (8)                    self._set_window_mode(mpl.rcParams["macosx.window_mode"])
105: (8)                    if self.toolbar is not None:
106: (12)                       self.toolbar.update()
107: (8)                    if mpl.is_interactive():
108: (12)                       self.show()
109: (12)                       self.canvas.draw_idle()
110: (4)                1 def _close_button_pressed(self):
111: (8)                    Gcf.destroy(self)
112: (8)                    self.canvas.flush_events()
113: (4)                def destroy(self):
114: (8)                    while self.canvas._timers:
```

```
115: (12)                        timer = self.canvas._timers.pop()
116: (12)                        timer.stop()
117: (8)                    super().destroy()
118: (4)                @classmethod
119: (4)                def start_main_loop(cls):
120: (8)                    with _allow_interrupt_macos():
121: (12)                       _macosx.show()
122: (4)                def show(self):
123: (8)                    if self.canvas.figure.stale:
124: (12)                       self.canvas.draw_idle()
125: (8)                    if not self._shown:
126: (12)                       self._show()
127: (12)                       self._shown = True
128: (8)                    if mpl.rcParams["figure.raise_window"]:
129: (12)                       self._raise()
130: (0)            @_Backend.export
131: (0)            class _BackendMac(_Backend):
132: (4)                FigureCanvas = FigureCanvasMac
133: (4)                FigureManager = FigureManagerMac
134: (4)                mainloop = FigureManagerMac.start_main_loop

----------------------------------------

File 89 - backend_mixed.py:

1: (0)              import numpy as np
2: (0)              from matplotlib import cbook
3: (0)              from .backend_agg import RendererAgg
4: (0)              from matplotlib._tight_bbox import process_figure_for_rasterizing
5: (0)              class MixedModeRenderer:
6: (4)                  """
7: (4)                  A helper class to implement a renderer that switches between
8: (4)                  vector and raster drawing.  An example may be a PDF writer, where
9: (4)                  most things are drawn with PDF vector commands, but some very
10: (4)                 complex objects, such as quad meshes, are rasterised and then
11: (4)                 output as images.
12: (4)                 """
13: (4)                 def __init__(self, figure, width, height, dpi, vector_renderer,
14: (17)                         raster_renderer_class=None,
15: (17)                         bbox_inches_restore=None):
16: (8)                     """
17: (8)                     Parameters
18: (8)                     ----------
19: (8)                     figure : `~matplotlib.figure.Figure`
20: (12)                        The figure instance.
21: (8)                     width : scalar
22: (12)                        The width of the canvas in logical units
23: (8)                     height : scalar
24: (12)                        The height of the canvas in logical units
25: (8)                     dpi : float
26: (12)                        The dpi of the canvas
27: (8)                     vector_renderer : `~matplotlib.backend_bases.RendererBase`
28: (12)                        An instance of a subclass of
29: (12)                        `~matplotlib.backend_bases.RendererBase` that will be used for the
30: (12)                        vector drawing.
31: (8)                     raster_renderer_class : `~matplotlib.backend_bases.RendererBase`
32: (12)                        The renderer class to use for the raster drawing.  If not
provided,
33: (12)                        this will use the Agg backend (which is currently the only viable
34: (12)                        option anyway.)
35: (8)                     """
36: (8)                     if raster_renderer_class is None:
37: (12)                        raster_renderer_class = RendererAgg
38: (8)                     self._raster_renderer_class = raster_renderer_class
39: (8)                     self._width = width
40: (8)                     self._height = height
41: (8)                     self.dpi = dpi
42: (8)                     self._vector_renderer = vector_renderer
43: (8)                     self._raster_renderer = None
```

```
44: (8)                     self.figure = figure
45: (8)                     self._figdpi = figure.dpi
46: (8)                     self._bbox_inches_restore = bbox_inches_restore
47: (8)                     self._renderer = vector_renderer
48: (4)                 def __getattr__(self, attr):
49: (8)                     return getattr(self._renderer, attr)
50: (4)                 def start_rasterizing(self):
51: (8)                     """
52: (8)                     Enter "raster" mode.  All subsequent drawing commands (until
53: (8)                     `stop_rasterizing` is called) will be drawn with the raster backend.
54: (8)                     """
55: (8)                     self.figure.dpi = self.dpi
56: (8)                     if self._bbox_inches_restore:  # when tight bbox is used
57: (12)                        r = process_figure_for_rasterizing(self.figure,
58: (47)                                                           self._bbox_inches_restore)
59: (12)                        self._bbox_inches_restore = r
60: (8)                     self._raster_renderer = self._raster_renderer_class(
61: (12)                        self._width*self.dpi, self._height*self.dpi, self.dpi)
62: (8)                     self._renderer = self._raster_renderer
63: (4)                 def stop_rasterizing(self):
64: (8)                     """
65: (8)                     Exit "raster" mode.  All of the drawing that was done since
66: (8)                     the last `start_rasterizing` call will be copied to the
67: (8)                     vector backend by calling draw_image.
68: (8)                     """
69: (8)                     self._renderer = self._vector_renderer
70: (8)                     height = self._height * self.dpi
71: (8)                     img = np.asarray(self._raster_renderer.buffer_rgba())
72: (8)                     slice_y, slice_x = cbook._get_nonzero_slices(img[..., 3])
73: (8)                     cropped_img = img[slice_y, slice_x]
74: (8)                     if cropped_img.size:
75: (12)                        gc = self._renderer.new_gc()
76: (12)                        self._renderer.draw_image(
77: (16)                            gc,
78: (16)                            slice_x.start * self._figdpi / self.dpi,
79: (16)                            (height - slice_y.stop) * self._figdpi / self.dpi,
80: (16)                            cropped_img[::-1])
81: (8)                     self._raster_renderer = None
82: (8)                     self.figure.dpi = self._figdpi
83: (8)                     if self._bbox_inches_restore:  # when tight bbox is used
84: (12)                        r = process_figure_for_rasterizing(self.figure,
85: (47)                                                           self._bbox_inches_restore,
86: (47)                                                           self._figdpi)
87: (12)                        self._bbox_inches_restore = r

----------------------------------------

File 90 - backend_nbagg.py:

1: (0)              """Interactive figures in the IPython notebook."""
2: (0)              from base64 import b64encode
3: (0)              import io
4: (0)              import json
5: (0)              import pathlib
6: (0)              import uuid
7: (0)              from ipykernel.comm import Comm
8: (0)              from IPython.display import display, Javascript, HTML
9: (0)              from matplotlib import is_interactive
10: (0)             from matplotlib._pylab_helpers import Gcf
11: (0)             from matplotlib.backend_bases import _Backend, CloseEvent, NavigationToolbar2
12: (0)             from .backend_webagg_core import (
13: (4)                 FigureCanvasWebAggCore, FigureManagerWebAgg, NavigationToolbar2WebAgg)
14: (0)             from .backend_webagg_core import (  # noqa: F401 # pylint: disable=W0611
15: (4)                 TimerTornado, TimerAsyncio)
16: (0)             def connection_info():
17: (4)                 """
18: (4)                 Return a string showing the figure and connection status for the backend.
19: (4)                 This is intended as a diagnostic tool, and not for general use.
20: (4)                 """
```

```
 21: (4)                 result = [
 22: (8)                     '{fig} - {socket}'.format(
 23: (12)                        fig=(manager.canvas.figure.get_label()
 24: (17)                             or f"Figure {manager.num}"),
 25: (12)                        socket=manager.web_sockets)
 26: (8)                     for manager in Gcf.get_all_fig_managers()
 27: (4)                 ]
 28: (4)                 if not is_interactive():
 29: (8)                     result.append(f'Figures pending show: {len(Gcf.figs)}')
 30: (4)                 return '\n'.join(result)
 31: (0)             _FONT_AWESOME_CLASSES = {  # font-awesome 4 names
 32: (4)                 'home': 'fa fa-home',
 33: (4)                 'back': 'fa fa-arrow-left',
 34: (4)                 'forward': 'fa fa-arrow-right',
 35: (4)                 'zoom_to_rect': 'fa fa-square-o',
 36: (4)                 'move': 'fa fa-arrows',
 37: (4)                 'download': 'fa fa-floppy-o',
 38: (4)                 None: None
 39: (0)             }
 40: (0)             class NavigationIPy(NavigationToolbar2WebAgg):
 41: (4)                 toolitems = [(text, tooltip_text,
 42: (18)                              _FONT_AWESOME_CLASSES[image_file], name_of_method)
 43: (17)                             for text, tooltip_text, image_file, name_of_method
 44: (17)                             in (NavigationToolbar2.toolitems +
 45: (21)                                 (('Download', 'Download plot', 'download', 'download'),))
 46: (17)                             if image_file in _FONT_AWESOME_CLASSES]
 47: (0)             class FigureManagerNbAgg(FigureManagerWebAgg):
 48: (4)                 _toolbar2_class = ToolbarCls = NavigationIPy
 49: (4)                 def __init__(self, canvas, num):
 50: (8)                     self._shown = False
 51: (8)                     super().__init__(canvas, num)
 52: (4)                 @classmethod
 53: (4)                 def create_with_canvas(cls, canvas_class, figure, num):
 54: (8)                     canvas = canvas_class(figure)
 55: (8)                     manager = cls(canvas, num)
 56: (8)                     if is_interactive():
 57: (12)                        manager.show()
 58: (12)                        canvas.draw_idle()
 59: (8)                     def destroy(event):
 60: (12)                        canvas.mpl_disconnect(cid)
 61: (12)                        Gcf.destroy(manager)
 62: (8)                     cid = canvas.mpl_connect('close_event', destroy)
 63: (8)                     return manager
 64: (4)                 def display_js(self):
 65: (8)                     display(Javascript(FigureManagerNbAgg.get_javascript()))
 66: (4)                 def show(self):
 67: (8)                     if not self._shown:
 68: (12)                        self.display_js()
 69: (12)                        self._create_comm()
 70: (8)                     else:
 71: (12)                        self.canvas.draw_idle()
 72: (8)                     self._shown = True
 73: (8)                     if hasattr(self, '_cidgcf'):
 74: (12)                        self.canvas.mpl_disconnect(self._cidgcf)
 75: (8)                     if not is_interactive():
 76: (12)                        from matplotlib._pylab_helpers import Gcf
 77: (12)                        Gcf.figs.pop(self.num, None)
 78: (4)                 def reshow(self):
 79: (8)                     """
 80: (8)                     A special method to re-show the figure in the notebook.
 81: (8)                     """
 82: (8)                     self._shown = False
 83: (8)                     self.show()
 84: (4)                 @property
 85: (4)                 def connected(self):
 86: (8)                     return bool(self.web_sockets)
 87: (4)                 @classmethod
 88: (4)                 def get_javascript(cls, stream=None):
 89: (8)                     if stream is None:
```

```
 90: (12)                        output = io.StringIO()
 91: (8)                     else:
 92: (12)                        output = stream
 93: (8)                     super().get_javascript(stream=output)
 94: (8)                     output.write((pathlib.Path(__file__).parent
 95: (22)                                  / "web_backend/js/nbagg_mpl.js")
 96: (21)                                 .read_text(encoding="utf-8"))
 97: (8)                     if stream is None:
 98: (12)                        return output.getvalue()
 99: (4)                 def _create_comm(self):
 100: (8)                    comm = CommSocket(self)
 101: (8)                    self.add_web_socket(comm)
 102: (8)                    return comm
 103: (4)                def destroy(self):
 104: (8)                    self._send_event('close')
 105: (8)                    for comm in list(self.web_sockets):
 106: (12)                       comm.on_close()
 107: (8)                    self.clearup_closed()
 108: (4)                def clearup_closed(self):
 109: (8)                    """Clear up any closed Comms."""
 110: (8)                    self.web_sockets = {socket for socket in self.web_sockets
 111: (28)                                       if socket.is_open()}
 112: (8)                    if len(self.web_sockets) == 0:
 113: (12)                       CloseEvent("close_event", self.canvas)._process()
 114: (4)                def remove_comm(self, comm_id):
 115: (8)                    self.web_sockets = {socket for socket in self.web_sockets
 116: (28)                                       if socket.comm.comm_id != comm_id}
 117: (0)            class FigureCanvasNbAgg(FigureCanvasWebAggCore):
 118: (4)                manager_class = FigureManagerNbAgg
 119: (0)            class CommSocket:
 120: (4)                """
 121: (4)                Manages the Comm connection between IPython and the browser (client).
 122: (4)                Comms are 2 way, with the CommSocket being able to publish a message
 123: (4)                via the send_json method, and handle a message with on_message. On the
 124: (4)                JS side figure.send_message and figure.ws.onmessage do the sending and
 125: (4)                receiving respectively.
 126: (4)                """
 127: (4)                def __init__(self, manager):
 128: (8)                    self.supports_binary = None
 129: (8)                    self.manager = manager
 130: (8)                    self.uuid = str(uuid.uuid4())
 131: (8)                    display(HTML("<div id=%r></div>" % self.uuid))
 132: (8)                    try:
 133: (12)                       self.comm = Comm('matplotlib', data={'id': self.uuid})
 134: (8)                    except AttributeError as err:
 135: (12)                       raise RuntimeError('Unable to create an IPython notebook Comm '
 136: (31)                                          'instance. Are you in the IPython '
 137: (31)                                          'notebook?') from err
 138: (8)                    self.comm.on_msg(self.on_message)
 139: (8)                    manager = self.manager
 140: (8)                    self._ext_close = False
 141: (8)                    def _on_close(close_message):
 142: (12)                       self._ext_close = True
 143: (12)                       manager.remove_comm(close_message['content']['comm_id'])
 144: (12)                       manager.clearup_closed()
 145: (8)                    self.comm.on_close(_on_close)
 146: (4)                def is_open(self):
 147: (8)                    return not (self._ext_close or self.comm._closed)
 148: (4)                def on_close(self):
 149: (8)                    if self.is_open():
 150: (12)                       try:
 151: (16)                           self.comm.close()
 152: (12)                       except KeyError:
 153: (16)                           pass
 154: (4)                def send_json(self, content):
 155: (8)                    self.comm.send({'data': json.dumps(content)})
 156: (4)                def send_binary(self, blob):
 157: (8)                    if self.supports_binary:
 158: (12)                       self.comm.send({'blob': 'image/png'}, buffers=[blob])
```

```
159: (8)                    else:
160: (12)                       data = b64encode(blob).decode('ascii')
161: (12)                       data_uri = f"{data}"
162: (12)                       self.comm.send({'data': data_uri})
163: (4)                def on_message(self, message):
164: (8)                    message = json.loads(message['content']['data'])
165: (8)                    if message['type'] == 'closing':
166: (12)                       self.on_close()
167: (12)                       self.manager.clearup_closed()
168: (8)                    elif message['type'] == 'supports_binary':
169: (12)                       self.supports_binary = message['value']
170: (8)                    else:
171: (12)                       self.manager.handle_json(message)
172: (0)            @_Backend.export
173: (0)            class _BackendNbAgg(_Backend):
174: (4)                FigureCanvas = FigureCanvasNbAgg
175: (4)                FigureManager = FigureManagerNbAgg

----------------------------------------

File 91 - _backend_pdf_ps.py:

1: (0)              """
2: (0)              Common functionality between the PDF and PS backends.
3: (0)              """
4: (0)              from io import BytesIO
5: (0)              import functools
6: (0)              from fontTools import subset
7: (0)              import matplotlib as mpl
8: (0)              from .. import font_manager, ft2font
9: (0)              from ._afm import AFM
10: (0)             from ..backend_bases import RendererBase
11: (0)             @functools.lru_cache(50)
12: (0)             def _cached_get_afm_from_fname(fname):
13: (4)                 with open(fname, "rb") as fh:
14: (8)                     return AFM(fh)
15: (0)             def get_glyphs_subset(fontfile, characters):
16: (4)                 """
17: (4)                 Subset a TTF font
18: (4)                 Reads the named fontfile and restricts the font to the characters.
19: (4)                 Returns a serialization of the subset font as file-like object.
20: (4)                 Parameters
21: (4)                 ----------
22: (4)                 fontfile : str
23: (8)                     Path to the font file
24: (4)                 characters : str
25: (8)                     Continuous set of characters to include in subset
26: (4)                 """
27: (4)                 options = subset.Options(glyph_names=True, recommended_glyphs=True)
28: (4)                 options.drop_tables += [
29: (8)                     'FFTM',  # FontForge Timestamp.
30: (8)                     'PfEd',  # FontForge personal table.
31: (8)                     'BDF',  # X11 BDF header.
32: (8)                     'meta',  # Metadata stores design/supported languages (meaningless for
subsets).
33: (4)                 ]
34: (4)                 if fontfile.endswith(".ttc"):
35: (8)                     options.font_number = 0
36: (4)                 with subset.load_font(fontfile, options) as font:
37: (8)                     subsetter = subset.Subsetter(options=options)
38: (8)                     subsetter.populate(text=characters)
39: (8)                     subsetter.subset(font)
40: (8)                     fh = BytesIO()
41: (8)                     font.save(fh, reorderTables=False)
42: (8)                     return fh
43: (0)             class CharacterTracker:
44: (4)                 """
45: (4)                 Helper for font subsetting by the pdf and ps backends.
46: (4)                 Maintains a mapping of font paths to the set of character codepoints that
```

```
47: (4)                 are being used from that font.
48: (4)                 """
49: (4)                 def __init__(self):
50: (8)                     self.used = {}
51: (4)                 def track(self, font, s):
52: (8)                     """Record that string *s* is being typeset using font *font*."""
53: (8)                     char_to_font = font._get_fontmap(s)
54: (8)                     for _c, _f in char_to_font.items():
55: (12)                        self.used.setdefault(_f.fname, set()).add(ord(_c))
56: (4)                 def track_glyph(self, font, glyph):
57: (8)                     """Record that codepoint *glyph* is being typeset using font
*font*."""
58: (8)                     self.used.setdefault(font.fname, set()).add(glyph)
59: (0)             class RendererPDFPSBase(RendererBase):
60: (4)                 def __init__(self, width, height):
61: (8)                     super().__init__()
62: (8)                     self.width = width
63: (8)                     self.height = height
64: (4)                 def flipy(self):
65: (8)                     return False  # y increases from bottom to top.
66: (4)                 def option_scale_image(self):
67: (8)                     return True  # PDF and PS support arbitrary image scaling.
68: (4)                 def option_image_nocomposite(self):
69: (8)                     return not mpl.rcParams["image.composite_image"]
70: (4)                 def get_canvas_width_height(self):
71: (8)                     return self.width * 72.0, self.height * 72.0
72: (4)                 def get_text_width_height_descent(self, s, prop, ismath):
73: (8)                     if ismath == "TeX":
74: (12)                        return super().get_text_width_height_descent(s, prop, ismath)
75: (8)                     elif ismath:
76: (12)                        parse = self._text2path.mathtext_parser.parse(s, 72, prop)
77: (12)                        return parse.width, parse.height, parse.depth
78: (8)                     elif mpl.rcParams[self._use_afm_rc_name]:
79: (12)                        font = self._get_font_afm(prop)
80: (12)                        l, b, w, h, d = font.get_str_bbox_and_descent(s)
81: (12)                        scale = prop.get_size_in_points() / 1000
82: (12)                        w *= scale
83: (12)                        h *= scale
84: (12)                        d *= scale
85: (12)                        return w, h, d
86: (8)                     else:
87: (12)                        font = self._get_font_ttf(prop)
88: (12)                        font.set_text(s, 0.0, flags=ft2font.LOAD_NO_HINTING)
89: (12)                        w, h = font.get_width_height()
90: (12)                        d = font.get_descent()
91: (12)                        scale = 1 / 64
92: (12)                        w *= scale
93: (12)                        h *= scale
94: (12)                        d *= scale
95: (12)                        return w, h, d
96: (4)                 def _get_font_afm(self, prop):
97: (8)                     fname = font_manager.findfont(
98: (12)                        prop, fontext="afm", directory=self._afm_font_dir)
99: (8)                     return _cached_get_afm_from_fname(fname)
100: (4)                def _get_font_ttf(self, prop):
101: (8)                    fnames = font_manager.fontManager._find_fonts_by_props(prop)
102: (8)                    font = font_manager.get_font(fnames)
103: (8)                    font.clear()
104: (8)                    font.set_size(prop.get_size_in_points(), 72)
105: (8)                    return font

----------------------------------------

File 92 - backend_pdf.py:

1: (0)              """
2: (0)              A PDF Matplotlib backend.
3: (0)              Author: Jouni K Seppänen <jks@iki.fi> and others.
4: (0)              """
```

```
 5: (0)              import codecs
 6: (0)              from datetime import timezone
 7: (0)              from datetime import datetime
 8: (0)              from enum import Enum
 9: (0)              from functools import total_ordering
10: (0)              from io import BytesIO
11: (0)              import itertools
12: (0)              import logging
13: (0)              import math
14: (0)              import os
15: (0)              import string
16: (0)              import struct
17: (0)              import sys
18: (0)              import time
19: (0)              import types
20: (0)              import warnings
21: (0)              import zlib
22: (0)              import numpy as np
23: (0)              from PIL import Image
24: (0)              import matplotlib as mpl
25: (0)              from matplotlib import _api, _text_helpers, _type1font, cbook, dviread
26: (0)              from matplotlib._pylab_helpers import Gcf
27: (0)              from matplotlib.backend_bases import (
28: (4)                  _Backend, FigureCanvasBase, FigureManagerBase, GraphicsContextBase,
29: (4)                  RendererBase)
30: (0)              from matplotlib.backends.backend_mixed import MixedModeRenderer
31: (0)              from matplotlib.figure import Figure
32: (0)              from matplotlib.font_manager import get_font, fontManager as _fontManager
33: (0)              from matplotlib._afm import AFM
34: (0)              from matplotlib.ft2font import (FIXED_WIDTH, ITALIC, LOAD_NO_SCALE,
35: (32)                                             LOAD_NO_HINTING, KERNING_UNFITTED, FT2Font)
36: (0)              from matplotlib.transforms import Affine2D, BboxBase
37: (0)              from matplotlib.path import Path
38: (0)              from matplotlib.dates import UTC
39: (0)              from matplotlib import _path
40: (0)              from . import _backend_pdf_ps
41: (0)              _log = logging.getLogger(__name__)
42: (0)              def _fill(strings, linelen=75):
43: (4)                  """
44: (4)                  Make one string from sequence of strings, with whitespace in between.
45: (4)                  The whitespace is chosen to form lines of at most *linelen* characters,
46: (4)                  if possible.
47: (4)                  """
48: (4)                  currpos = 0
49: (4)                  lasti = 0
50: (4)                  result = []
51: (4)                  for i, s in enumerate(strings):
52: (8)                      length = len(s)
53: (8)                      if currpos + length < linelen:
54: (12)                         currpos += length + 1
55: (8)                      else:
56: (12)                         result.append(b' '.join(strings[lasti:i]))
57: (12)                         lasti = i
58: (12)                         currpos = length
59: (4)                  result.append(b' '.join(strings[lasti:]))
60: (4)                  return b'\n'.join(result)
61: (0)              def _create_pdf_info_dict(backend, metadata):
62: (4)                  """
63: (4)                  Create a PDF infoDict based on user-supplied metadata.
64: (4)                  A default ``Creator``, ``Producer``, and ``CreationDate`` are added,
though
65: (4)                  the user metadata may override it. The date may be the current time, or a
66: (4)                  time set by the ``SOURCE_DATE_EPOCH`` environment variable.
67: (4)                  Metadata is verified to have the correct keys and their expected types.
Any
68: (4)                  unknown keys/types will raise a warning.
69: (4)                  Parameters
70: (4)                  ----------
71: (4)                  backend : str
```

```
72: (8)                     The name of the backend to use in the Producer value.
73: (4)                 metadata : dict[str, Union[str, datetime, Name]]
74: (8)                     A dictionary of metadata supplied by the user with information
75: (8)                     following the PDF specification, also defined in
76: (8)                     `~.backend_pdf.PdfPages` below.
77: (8)                     If any value is *None*, then the key will be removed. This can be used
78: (8)                     to remove any pre-defined values.
79: (4)                 Returns
80: (4)                 -------
81: (4)                 dict[str, Union[str, datetime, Name]]
82: (8)                     A validated dictionary of metadata.
83: (4)                 """
84: (4)                 source_date_epoch = os.getenv("SOURCE_DATE_EPOCH")
85: (4)                 if source_date_epoch:
86: (8)                     source_date = datetime.fromtimestamp(int(source_date_epoch),
timezone.utc)
87: (8)                     source_date = source_date.replace(tzinfo=UTC)
88: (4)                 else:
89: (8)                     source_date = datetime.today()
90: (4)                 info = {
91: (8)                     'Creator': f'Matplotlib v{mpl.__version__}, https://matplotlib.org',
92: (8)                     'Producer': f'Matplotlib {backend} backend v{mpl.__version__}',
93: (8)                     'CreationDate': source_date,
94: (8)                     **metadata
95: (4)                 }
96: (4)                 info = {k: v for (k, v) in info.items() if v is not None}
97: (4)                 def is_string_like(x):
98: (8)                     return isinstance(x, str)
99: (4)                 is_string_like.text_for_warning = "an instance of str"
100: (4)                def is_date(x):
101: (8)                    return isinstance(x, datetime)
102: (4)                is_date.text_for_warning = "an instance of datetime.datetime"
103: (4)                def check_trapped(x):
104: (8)                    if isinstance(x, Name):
105: (12)                       return x.name in (b'True', b'False', b'Unknown')
106: (8)                    else:
107: (12)                       return x in ('True', 'False', 'Unknown')
108: (4)                check_trapped.text_for_warning = 'one of {"True", "False", "Unknown"}'
109: (4)                keywords = {
110: (8)                    'Title': is_string_like,
111: (8)                    'Author': is_string_like,
112: (8)                    'Subject': is_string_like,
113: (8)                    'Keywords': is_string_like,
114: (8)                    'Creator': is_string_like,
115: (8)                    'Producer': is_string_like,
116: (8)                    'CreationDate': is_date,
117: (8)                    'ModDate': is_date,
118: (8)                    'Trapped': check_trapped,
119: (4)                }
120: (4)                for k in info:
121: (8)                    if k not in keywords:
122: (12)                       _api.warn_external(f'Unknown infodict keyword: {k!r}. '
123: (31)                                          f'Must be one of {set(keywords)!r}.')
124: (8)                    elif not keywords[k](info[k]):
125: (12)                       _api.warn_external(f'Bad value for infodict keyword {k}. '
126: (31)                                          f'Got {info[k]!r} which is not '
127: (31)                                          f'{keywords[k].text_for_warning}.')
128: (4)                if 'Trapped' in info:
129: (8)                    info['Trapped'] = Name(info['Trapped'])
130: (4)                return info
131: (0)            def _datetime_to_pdf(d):
132: (4)                """
133: (4)                Convert a datetime to a PDF string representing it.
134: (4)                Used for PDF and PGF.
135: (4)                """
136: (4)                r = d.strftime('D:%Y%m%d%H%M%S')
137: (4)                z = d.utcoffset()
138: (4)                if z is not None:
139: (8)                    z = z.seconds
```

```
140: (4)                else:
141: (8)                    if time.daylight:
142: (12)                       z = time.altzone
143: (8)                    else:
144: (12)                       z = time.timezone
145: (4)                if z == 0:
146: (8)                    r += 'Z'
147: (4)                elif z < 0:
148: (8)                    r += "+%02d'%02d'" % ((-z) // 3600, (-z) % 3600)
149: (4)                else:
150: (8)                    r += "-%02d'%02d'" % (z // 3600, z % 3600)
151: (4)                return r
152: (0)            def _calculate_quad_point_coordinates(x, y, width, height, angle=0):
153: (4)                """
154: (4)                Calculate the coordinates of rectangle when rotated by angle around x, y
155: (4)                """
156: (4)                angle = math.radians(-angle)
157: (4)                sin_angle = math.sin(angle)
158: (4)                cos_angle = math.cos(angle)
159: (4)                a = x + height * sin_angle
160: (4)                b = y + height * cos_angle
161: (4)                c = x + width * cos_angle + height * sin_angle
162: (4)                d = y - width * sin_angle + height * cos_angle
163: (4)                e = x + width * cos_angle
164: (4)                f = y - width * sin_angle
165: (4)                return ((x, y), (e, f), (c, d), (a, b))
166: (0)            def _get_coordinates_of_block(x, y, width, height, angle=0):
167: (4)                """
168: (4)                Get the coordinates of rotated rectangle and rectangle that covers the
169: (4)                rotated rectangle.
170: (4)                """
171: (4)                vertices = _calculate_quad_point_coordinates(x, y, width,
172: (49)                                                            height, angle)
173: (4)                pad = 0.00001 if angle % 90 else 0
174: (4)                min_x = min(v[0] for v in vertices) - pad
175: (4)                min_y = min(v[1] for v in vertices) - pad
176: (4)                max_x = max(v[0] for v in vertices) + pad
177: (4)                max_y = max(v[1] for v in vertices) + pad
178: (4)                return (tuple(itertools.chain.from_iterable(vertices)),
179: (12)                       (min_x, min_y, max_x, max_y))
180: (0)            def _get_link_annotation(gc, x, y, width, height, angle=0):
181: (4)                """
182: (4)                Create a link annotation object for embedding URLs.
183: (4)                """
184: (4)                quadpoints, rect = _get_coordinates_of_block(x, y, width, height, angle)
185: (4)                link_annotation = {
186: (8)                    'Type': Name('Annot'),
187: (8)                    'Subtype': Name('Link'),
188: (8)                    'Rect': rect,
189: (8)                    'Border': [0, 0, 0],
190: (8)                    'A': {
191: (12)                       'S': Name('URI'),
192: (12)                       'URI': gc.get_url(),
193: (8)                    },
194: (4)                }
195: (4)                if angle % 90:
196: (8)                    link_annotation['QuadPoints'] = quadpoints
197: (4)                return link_annotation
198: (0)            _str_escapes = str.maketrans({
199: (4)                '\\': '\\\\', '(': '\\(', ')': '\\)', '\n': '\\n', '\r': '\\r'})
200: (0)            def pdfRepr(obj):
201: (4)                """Map Python objects to PDF syntax."""
202: (4)                if hasattr(obj, 'pdfRepr'):
203: (8)                    return obj.pdfRepr()
204: (4)                elif isinstance(obj, (float, np.floating)):
205: (8)                    if not np.isfinite(obj):
206: (12)                       raise ValueError("Can only output finite numbers in PDF")
207: (8)                    r = b"%.10f" % obj
208: (8)                    return r.rstrip(b'0').rstrip(b'.')
```

```
209: (4)                elif isinstance(obj, bool):
210: (8)                    return [b'false', b'true'][obj]
211: (4)                elif isinstance(obj, (int, np.integer)):
212: (8)                    return b"%d" % obj
213: (4)                elif isinstance(obj, str):
214: (8)                    return pdfRepr(obj.encode('ascii') if obj.isascii()
215: (23)                                  else codecs.BOM_UTF16_BE + obj.encode('UTF-16BE'))
216: (4)                elif isinstance(obj, bytes):
217: (8)                    return (
218: (12)                       b'(' +
219: (12)                       obj.decode('latin-1').translate(_str_escapes).encode('latin-1')
220: (12)                       + b')')
221: (4)                elif isinstance(obj, dict):
222: (8)                    return _fill([
223: (12)                       b"<<",
224: (12)                       *[Name(k).pdfRepr() + b" " + pdfRepr(v) for k, v in obj.items()],
225: (12)                       b">>",
226: (8)                    ])
227: (4)                elif isinstance(obj, (list, tuple)):
228: (8)                    return _fill([b"[", *[pdfRepr(val) for val in obj], b"]"])
229: (4)                elif obj is None:
230: (8)                    return b'null'
231: (4)                elif isinstance(obj, datetime):
232: (8)                    return pdfRepr(_datetime_to_pdf(obj))
233: (4)                elif isinstance(obj, BboxBase):
234: (8)                    return _fill([pdfRepr(val) for val in obj.bounds])
235: (4)                else:
236: (8)                    raise TypeError(f"Don't know a PDF representation for {type(obj)} "
237: (24)                                   "objects")
238: (0)            def _font_supports_glyph(fonttype, glyph):
239: (4)                """
240: (4)                Returns True if the font is able to provide codepoint *glyph* in a PDF.
241: (4)                For a Type 3 font, this method returns True only for single-byte
242: (4)                characters. For Type 42 fonts this method return True if the character is
243: (4)                from the Basic Multilingual Plane.
244: (4)                """
245: (4)                if fonttype == 3:
246: (8)                    return glyph <= 255
247: (4)                if fonttype == 42:
248: (8)                    return glyph <= 65535
249: (4)                raise NotImplementedError()
250: (0)            class Reference:
251: (4)                """
252: (4)                PDF reference object.
253: (4)                Use PdfFile.reserveObject() to create References.
254: (4)                """
255: (4)                def __init__(self, id):
256: (8)                    self.id = id
257: (4)                def __repr__(self):
258: (8)                    return "<Reference %d>" % self.id
259: (4)                def pdfRepr(self):
260: (8)                    return b"%d 0 R" % self.id
261: (4)                def write(self, contents, file):
262: (8)                    write = file.write
263: (8)                    write(b"%d 0 obj\n" % self.id)
264: (8)                    write(pdfRepr(contents))
265: (8)                    write(b"\nendobj\n")
266: (0)            @total_ordering
267: (0)            class Name:
268: (4)                """PDF name object."""
269: (4)                __slots__ = ('name',)
270: (4)                _hexify = {c: '#%02x' % c
271: (15)                          for c in {*range(256)} - {*range(ord('!'), ord('~') + 1)}}
272: (4)                def __init__(self, name):
273: (8)                    if isinstance(name, Name):
274: (12)                       self.name = name.name
275: (8)                    else:
276: (12)                       if isinstance(name, bytes):
277: (16)                           name = name.decode('ascii')
```

```
278: (12)                        self.name = name.translate(self._hexify).encode('ascii')
279: (4)                    def __repr__(self):
280: (8)                        return "<Name %s>" % self.name
281: (4)                    def __str__(self):
282: (8)                        return '/' + self.name.decode('ascii')
283: (4)                    def __eq__(self, other):
284: (8)                        return isinstance(other, Name) and self.name == other.name
285: (4)                    def __lt__(self, other):
286: (8)                        return isinstance(other, Name) and self.name < other.name
287: (4)                    def __hash__(self):
288: (8)                        return hash(self.name)
289: (4)                    def pdfRepr(self):
290: (8)                        return b'/' + self.name
291: (0)            class Verbatim:
292: (4)                """Store verbatim PDF command content for later inclusion in the
stream."""
293: (4)                    def __init__(self, x):
294: (8)                        self._x = x
295: (4)                    def pdfRepr(self):
296: (8)                        return self._x
297: (0)            class Op(Enum):
298: (4)                """PDF operators (not an exhaustive list)."""
299: (4)                    close_fill_stroke = b'b'
300: (4)                    fill_stroke = b'B'
301: (4)                    fill = b'f'
302: (4)                    closepath = b'h'
303: (4)                    close_stroke = b's'
304: (4)                    stroke = b'S'
305: (4)                    endpath = b'n'
306: (4)                    begin_text = b'BT'
307: (4)                    end_text = b'ET'
308: (4)                    curveto = b'c'
309: (4)                    rectangle = b're'
310: (4)                    lineto = b'l'
311: (4)                    moveto = b'm'
312: (4)                    concat_matrix = b'cm'
313: (4)                    use_xobject = b'Do'
314: (4)                    setgray_stroke = b'G'
315: (4)                    setgray_nonstroke = b'g'
316: (4)                    setrgb_stroke = b'RG'
317: (4)                    setrgb_nonstroke = b'rg'
318: (4)                    setcolorspace_stroke = b'CS'
319: (4)                    setcolorspace_nonstroke = b'cs'
320: (4)                    setcolor_stroke = b'SCN'
321: (4)                    setcolor_nonstroke = b'scn'
322: (4)                    setdash = b'd'
323: (4)                    setlinejoin = b'j'
324: (4)                    setlinecap = b'J'
325: (4)                    setgstate = b'gs'
326: (4)                    gsave = b'q'
327: (4)                    grestore = b'Q'
328: (4)                    textpos = b'Td'
329: (4)                    selectfont = b'Tf'
330: (4)                    textmatrix = b'Tm'
331: (4)                    show = b'Tj'
332: (4)                    showkern = b'TJ'
333: (4)                    setlinewidth = b'w'
334: (4)                    clip = b'W'
335: (4)                    shading = b'sh'
336: (4)                    def pdfRepr(self):
337: (8)                        return self.value
338: (4)                    @classmethod
339: (4)                    def paint_path(cls, fill, stroke):
340: (8)                        """
341: (8)                        Return the PDF operator to paint a path.
342: (8)                        Parameters
343: (8)                        ----------
344: (8)                        fill : bool
345: (12)                           Fill the path with the fill color.
```

```
346: (8)                    stroke : bool
347: (12)                       Stroke the outline of the path with the line color.
348: (8)                    """
349: (8)                    if stroke:
350: (12)                       if fill:
351: (16)                           return cls.fill_stroke
352: (12)                       else:
353: (16)                           return cls.stroke
354: (8)                    else:
355: (12)                       if fill:
356: (16)                           return cls.fill
357: (12)                       else:
358: (16)                           return cls.endpath
359: (0)            class Stream:
360: (4)                """
361: (4)                PDF stream object.
362: (4)                This has no pdfRepr method. Instead, call begin(), then output the
363: (4)                contents of the stream by calling write(), and finally call end().
364: (4)                """
365: (4)                __slots__ = ('id', 'len', 'pdfFile', 'file', 'compressobj', 'extra',
'pos')
366: (4)                def __init__(self, id, len, file, extra=None, png=None):
367: (8)                    """
368: (8)                    Parameters
369: (8)                    ----------
370: (8)                    id : int
371: (12)                       Object id of the stream.
372: (8)                    len : Reference or None
373: (12)                       An unused Reference object for the length of the stream;
374: (12)                       None means to use a memory buffer so the length can be inlined.
375: (8)                    file : PdfFile
376: (12)                       The underlying object to write the stream to.
377: (8)                    extra : dict from Name to anything, or None
378: (12)                       Extra key-value pairs to include in the stream header.
379: (8)                    png : dict or None
380: (12)                       If the data is already png encoded, the decode parameters.
381: (8)                    """
382: (8)                    self.id = id            # object id
383: (8)                    self.len = len          # id of length object
384: (8)                    self.pdfFile = file
385: (8)                    self.file = file.fh      # file to which the stream is written
386: (8)                    self.compressobj = None  # compression object
387: (8)                    if extra is None:
388: (12)                       self.extra = dict()
389: (8)                    else:
390: (12)                       self.extra = extra.copy()
391: (8)                    if png is not None:
392: (12)                       self.extra.update({'Filter':      Name('FlateDecode'),
393: (31)                                          'DecodeParms': png})
394: (8)                    self.pdfFile.recordXref(self.id)
395: (8)                    if mpl.rcParams['pdf.compression'] and not png:
396: (12)                       self.compressobj = zlib.compressobj(
397: (16)                           mpl.rcParams['pdf.compression'])
398: (8)                    if self.len is None:
399: (12)                       self.file = BytesIO()
400: (8)                    else:
401: (12)                       self._writeHeader()
402: (12)                       self.pos = self.file.tell()
403: (4)                def _writeHeader(self):
404: (8)                    write = self.file.write
405: (8)                    write(b"%d 0 obj\n" % self.id)
406: (8)                    dict = self.extra
407: (8)                    dict['Length'] = self.len
408: (8)                    if mpl.rcParams['pdf.compression']:
409: (12)                       dict['Filter'] = Name('FlateDecode')
410: (8)                    write(pdfRepr(dict))
411: (8)                    write(b"\nstream\n")
412: (4)                def end(self):
413: (8)                    """Finalize stream."""
```

```
414: (8)                    self._flush()
415: (8)                    if self.len is None:
416: (12)                       contents = self.file.getvalue()
417: (12)                       self.len = len(contents)
418: (12)                       self.file = self.pdfFile.fh
419: (12)                       self._writeHeader()
420: (12)                       self.file.write(contents)
421: (12)                       self.file.write(b"\nendstream\nendobj\n")
422: (8)                    else:
423: (12)                       length = self.file.tell() - self.pos
424: (12)                       self.file.write(b"\nendstream\nendobj\n")
425: (12)                       self.pdfFile.writeObject(self.len, length)
426: (4)                def write(self, data):
427: (8)                    """Write some data on the stream."""
428: (8)                    if self.compressobj is None:
429: (12)                       self.file.write(data)
430: (8)                    else:
431: (12)                       compressed = self.compressobj.compress(data)
432: (12)                       self.file.write(compressed)
433: (4)                def _flush(self):
434: (8)                    """Flush the compression object."""
435: (8)                    if self.compressobj is not None:
436: (12)                       compressed = self.compressobj.flush()
437: (12)                       self.file.write(compressed)
438: (12)                       self.compressobj = None
439: (0)            def _get_pdf_charprocs(font_path, glyph_ids):
440: (4)                font = get_font(font_path, hinting_factor=1)
441: (4)                conv = 1000 / font.units_per_EM  # Conversion to PS units (1/1000's).
442: (4)                procs = {}
443: (4)                for glyph_id in glyph_ids:
444: (8)                    g = font.load_glyph(glyph_id, LOAD_NO_SCALE)
445: (8)                    d1 = (np.array([g.horiAdvance, 0, *g.bbox]) * conv + .5).astype(int)
446: (8)                    v, c = font.get_path()
447: (8)                    v = (v * 64).astype(int)  # Back to TrueType's internal units
(1/64's).
448: (8)                    quads, = np.nonzero(c == 3)
449: (8)                    quads_on = quads[1::2]
450: (8)                    quads_mid_on = np.array(
451: (12)                       sorted({*quads_on} & {*(quads - 1)} & {*(quads + 1)}), int)
452: (8)                    implicit = quads_mid_on[
453: (12)                       (v[quads_mid_on]  # As above, use astype(int), not // division
454: (13)                        == ((v[quads_mid_on - 1] + v[quads_mid_on + 1]) / 2).astype(int))
455: (12)                       .all(axis=1)]
456: (8)                    if (font.postscript_name, glyph_id) in [
457: (16)                           ("DejaVuSerif-Italic", 77),  # j
458: (16)                           ("DejaVuSerif-Italic", 135),  # \AA
459: (8)                    ]:
460: (12)                       v[:, 0] -= 1  # Hard-coded backcompat (FreeType shifts glyph by
1).
461: (8)                    v = (v * conv + .5).astype(int)  # As above re: truncation vs
rounding.
462: (8)                    v[implicit] = ((  # Fix implicit points; again, truncate.
463: (12)                       (v[implicit - 1] + v[implicit + 1]) / 2).astype(int))
464: (8)                    procs[font.get_glyph_name(glyph_id)] = (
465: (12)                       " ".join(map(str, d1)).encode("ascii") + b" d1\n"
466: (12)                       + _path.convert_to_string(
467: (16)                           Path(v, c), None, None, False, None, -1,
468: (16)                           [b"m", b"l", b"", b"c", b"h"], True)
469: (12)                       + b"f")
470: (4)                return procs
471: (0)            class PdfFile:
472: (4)                """PDF file object."""
473: (4)                def __init__(self, filename, metadata=None):
474: (8)                    """
475: (8)                    Parameters
476: (8)                    ----------
477: (8)                    filename : str or path-like or file-like
478: (12)                       Output target; if a string, a file will be opened for writing.
479: (8)                    metadata : dict from strings to strings and dates
```

```
(12)                        Information dictionary object (see PDF reference section 10.2.1
(12)                        'Document Information Dictionary'), e.g.:
(12)                        ``{'Creator': 'My software', 'Author': 'Me', 'Title':
'Awesome'}``.
(12)                        The standard keys are 'Title', 'Author', 'Subject', 'Keywords',
(12)                        'Creator', 'Producer', 'CreationDate', 'ModDate', and
(12)                        'Trapped'. Values have been predefined for 'Creator', 'Producer'
(12)                        and 'CreationDate'. They can be removed by setting them to `None`.
(8)                     """
(8)                     super().__init__()
(8)                     self._object_seq = itertools.count(1)  # consumed by reserveObject
(8)                     self.xrefTable = [[0, 65535, 'the zero object']]
(8)                     self.passed_in_file_object = False
(8)                     self.original_file_like = None
(8)                     self.tell_base = 0
(8)                     fh, opened = cbook.to_filehandle(filename, "wb", return_opened=True)
(8)                     if not opened:
(12)                        try:
(16)                            self.tell_base = filename.tell()
(12)                        except OSError:
(16)                            fh = BytesIO()
(16)                            self.original_file_like = filename
(12)                        else:
(16)                            fh = filename
(16)                            self.passed_in_file_object = True
(8)                     self.fh = fh
(8)                     self.currentstream = None  # stream object to write to, if any
(8)                     fh.write(b"%PDF-1.4\n")    # 1.4 is the first version to have alpha
(8)                     fh.write(b"%\254\334 \253\272\n")
(8)                     self.rootObject = self.reserveObject('root')
(8)                     self.pagesObject = self.reserveObject('pages')
(8)                     self.pageList = []
(8)                     self.fontObject = self.reserveObject('fonts')
(8)                     self._extGStateObject = self.reserveObject('extended graphics states')
(8)                     self.hatchObject = self.reserveObject('tiling patterns')
(8)                     self.gouraudObject = self.reserveObject('Gouraud triangles')
(8)                     self.XObjectObject = self.reserveObject('external objects')
(8)                     self.resourceObject = self.reserveObject('resources')
(8)                     root = {'Type': Name('Catalog'),
(16)                            'Pages': self.pagesObject}
(8)                     self.writeObject(self.rootObject, root)
(8)                     self.infoDict = _create_pdf_info_dict('pdf', metadata or {})
(8)                     self.fontNames = {}     # maps filenames to internal font names
(8)                     self._internal_font_seq = (Name(f'F{i}') for i in itertools.count(1))
(8)                     self.dviFontInfo = {}   # maps dvi font names to embedding information
(8)                     self.type1Descriptors = {}
(8)                     self._character_tracker = _backend_pdf_ps.CharacterTracker()
(8)                     self.alphaStates = {}   # maps alpha values to graphics state objects
(8)                     self._alpha_state_seq = (Name(f'A{i}') for i in itertools.count(1))
(8)                     self._soft_mask_states = {}
(8)                     self._soft_mask_seq = (Name(f'SM{i}') for i in itertools.count(1))
(8)                     self._soft_mask_groups = []
(8)                     self.hatchPatterns = {}
(8)                     self._hatch_pattern_seq = (Name(f'H{i}') for i in itertools.count(1))
(8)                     self.gouraudTriangles = []
(8)                     self._images = {}
(8)                     self._image_seq = (Name(f'I{i}') for i in itertools.count(1))
(8)                     self.markers = {}
(8)                     self.multi_byte_charprocs = {}
(8)                     self.paths = []
(8)                     self._annotations = []
(8)                     self.pageAnnotations = []
(8)                     procsets = [Name(x) for x in "PDF Text ImageB ImageC ImageI".split()]
(8)                     resources = {'Font': self.fontObject,
(21)                                 'XObject': self.XObjectObject,
(21)                                 'ExtGState': self._extGStateObject,
(21)                                 'Pattern': self.hatchObject,
(21)                                 'Shading': self.gouraudObject,
(21)                                 'ProcSet': procsets}
```

```
(8)                 self.writeObject(self.resourceObject, resources)
(4)             def newPage(self, width, height):
(8)                 self.endStream()
(8)                 self.width, self.height = width, height
(8)                 contentObject = self.reserveObject('page contents')
(8)                 annotsObject = self.reserveObject('annotations')
(8)                 thePage = {'Type': Name('Page'),
(19)                           'Parent': self.pagesObject,
(19)                           'Resources': self.resourceObject,
(19)                           'MediaBox': [0, 0, 72 * width, 72 * height],
(19)                           'Contents': contentObject,
(19)                           'Annots': annotsObject,
(19)                           }
(8)                 pageObject = self.reserveObject('page')
(8)                 self.writeObject(pageObject, thePage)
(8)                 self.pageList.append(pageObject)
(8)                 self._annotations.append((annotsObject, self.pageAnnotations))
(8)                 self.beginStream(contentObject.id,
(25)                                 self.reserveObject('length of content stream'))
(8)                 self.output(Name('DeviceRGB'), Op.setcolorspace_stroke)
(8)                 self.output(Name('DeviceRGB'), Op.setcolorspace_nonstroke)
(8)                 self.output(GraphicsContextPdf.joinstyles['round'], Op.setlinejoin)
(8)                 self.pageAnnotations = []
(4)             def newTextnote(self, text, positionRect=[-100, -100, 0, 0]):
(8)                 theNote = {'Type': Name('Annot'),
(19)                           'Subtype': Name('Text'),
(19)                           'Contents': text,
(19)                           'Rect': positionRect,
(19)                           }
(8)                 self.pageAnnotations.append(theNote)
(4)             def _get_subsetted_psname(self, ps_name, charmap):
(8)                 def toStr(n, base):
(12)                    if n < base:
(16)                        return string.ascii_uppercase[n]
(12)                    else:
(16)                        return (
(20)                            toStr(n // base, base) + string.ascii_uppercase[n % base]
(16)                        )
(8)                 hashed = hash(frozenset(charmap.keys())) % ((sys.maxsize + 1) * 2)
(8)                 prefix = toStr(hashed, 26)
(8)                 return prefix[:6] + "+" + ps_name
(4)             def finalize(self):
(8)                 """Write out the various deferred objects and the pdf end matter."""
(8)                 self.endStream()
(8)                 self._write_annotations()
(8)                 self.writeFonts()
(8)                 self.writeExtGSTates()
(8)                 self._write_soft_mask_groups()
(8)                 self.writeHatches()
(8)                 self.writeGouraudTriangles()
(8)                 xobjects = {
(12)                    name: ob for image, name, ob in self._images.values()}
(8)                 for tup in self.markers.values():
(12)                    xobjects[tup[0]] = tup[1]
(8)                 for name, value in self.multi_byte_charprocs.items():
(12)                    xobjects[name] = value
(8)                 for name, path, trans, ob, join, cap, padding, filled, stroked \
(16)                        in self.paths:
(12)                    xobjects[name] = ob
(8)                 self.writeObject(self.XObjectObject, xobjects)
(8)                 self.writeImages()
(8)                 self.writeMarkers()
(8)                 self.writePathCollectionTemplates()
(8)                 self.writeObject(self.pagesObject,
(25)                                 {'Type': Name('Pages'),
(26)                                  'Kids': self.pageList,
(26)                                  'Count': len(self.pageList)})
(8)                 self.writeInfoDict()
(8)                 self.writeXref()
```

```
(8)                 self.writeTrailer()
(4)             def close(self):
(8)                 """Flush all buffers and free all resources."""
(8)                 self.endStream()
(8)                 if self.passed_in_file_object:
(12)                    self.fh.flush()
(8)                 else:
(12)                    if self.original_file_like is not None:
(16)                        self.original_file_like.write(self.fh.getvalue())
(12)                    self.fh.close()
(4)             def write(self, data):
(8)                 if self.currentstream is None:
(12)                    self.fh.write(data)
(8)                 else:
(12)                    self.currentstream.write(data)
(4)             def output(self, *data):
(8)                 self.write(_fill([pdfRepr(x) for x in data]))
(8)                 self.write(b'\n')
(4)             def beginStream(self, id, len, extra=None, png=None):
(8)                 assert self.currentstream is None
(8)                 self.currentstream = Stream(id, len, self, extra, png)
(4)             def endStream(self):
(8)                 if self.currentstream is not None:
(12)                    self.currentstream.end()
(12)                    self.currentstream = None
(4)             def outputStream(self, ref, data, *, extra=None):
(8)                 self.beginStream(ref.id, None, extra)
(8)                 self.currentstream.write(data)
(8)                 self.endStream()
(4)             def _write_annotations(self):
(8)                 for annotsObject, annotations in self._annotations:
(12)                    self.writeObject(annotsObject, annotations)
(4)             def fontName(self, fontprop):
(8)                 """
(8)                 Select a font based on fontprop and return a name suitable for
(8)                 Op.selectfont. If fontprop is a string, it will be interpreted
(8)                 as the filename of the font.
(8)                 """
(8)                 if isinstance(fontprop, str):
(12)                    filenames = [fontprop]
(8)                 elif mpl.rcParams['pdf.use14corefonts']:
(12)                    filenames = _fontManager._find_fonts_by_props(
(16)                        fontprop, fontext='afm', directory=RendererPdf._afm_font_dir
(12)                    )
(8)                 else:
(12)                    filenames = _fontManager._find_fonts_by_props(fontprop)
(8)                 first_Fx = None
(8)                 for fname in filenames:
(12)                    Fx = self.fontNames.get(fname)
(12)                    if not first_Fx:
(16)                        first_Fx = Fx
(12)                    if Fx is None:
(16)                        Fx = next(self._internal_font_seq)
(16)                        self.fontNames[fname] = Fx
(16)                        _log.debug('Assigning font %s = %r', Fx, fname)
(16)                        if not first_Fx:
(20)                            first_Fx = Fx
(8)                 return first_Fx
(4)             def dviFontName(self, dvifont):
(8)                 """
(8)                 Given a dvi font object, return a name suitable for Op.selectfont.
(8)                 This registers the font information in ``self.dviFontInfo`` if not yet
(8)                 registered.
(8)                 """
(8)                 dvi_info = self.dviFontInfo.get(dvifont.texname)
(8)                 if dvi_info is not None:
(12)                    return dvi_info.pdfname
(8)                 tex_font_map = dviread.PsfontsMap(dviread.find_tex_file('pdftex.map'))
(8)                 psfont = tex_font_map[dvifont.texname]
```

```
686: (8)                    if psfont.filename is None:
687: (12)                       raise ValueError(
688: (16)                           "No usable font file found for {} (TeX: {}); "
689: (16)                           "the font may lack a Type-1 version"
690: (16)                           .format(psfont.psname, dvifont.texname))
691: (8)                    pdfname = next(self._internal_font_seq)
692: (8)                    _log.debug('Assigning font %s = %s (dvi)', pdfname, dvifont.texname)
693: (8)                    self.dviFontInfo[dvifont.texname] = types.SimpleNamespace(
694: (12)                       dvifont=dvifont,
695: (12)                       pdfname=pdfname,
696: (12)                       fontfile=psfont.filename,
697: (12)                       basefont=psfont.psname,
698: (12)                       encodingfile=psfont.encoding,
699: (12)                       effects=psfont.effects)
700: (8)                    return pdfname
701: (4)                def writeFonts(self):
702: (8)                    fonts = {}
703: (8)                    for dviname, info in sorted(self.dviFontInfo.items()):
704: (12)                       Fx = info.pdfname
705: (12)                       _log.debug('Embedding Type-1 font %s from dvi.', dviname)
706: (12)                       fonts[Fx] = self._embedTeXFont(info)
707: (8)                    for filename in sorted(self.fontNames):
708: (12)                       Fx = self.fontNames[filename]
709: (12)                       _log.debug('Embedding font %s.', filename)
710: (12)                       if filename.endswith('.afm'):
711: (16)                           _log.debug('Writing AFM font.')
712: (16)                           fonts[Fx] = self._write_afm_font(filename)
713: (12)                       else:
714: (16)                           _log.debug('Writing TrueType font.')
715: (16)                           chars = self._character_tracker.used.get(filename)
716: (16)                           if chars:
717: (20)                               fonts[Fx] = self.embedTTF(filename, chars)
718: (8)                    self.writeObject(self.fontObject, fonts)
719: (4)                def _write_afm_font(self, filename):
720: (8)                    with open(filename, 'rb') as fh:
721: (12)                       font = AFM(fh)
722: (8)                    fontname = font.get_fontname()
723: (8)                    fontdict = {'Type': Name('Font'),
724: (20)                               'Subtype': Name('Type1'),
725: (20)                               'BaseFont': Name(fontname),
726: (20)                               'Encoding': Name('WinAnsiEncoding')}
727: (8)                    fontdictObject = self.reserveObject('font dictionary')
728: (8)                    self.writeObject(fontdictObject, fontdict)
729: (8)                    return fontdictObject
730: (4)                def _embedTeXFont(self, fontinfo):
731: (8)                    _log.debug('Embedding TeX font %s - fontinfo=%s',
732: (19)                              fontinfo.dvifont.texname, fontinfo.__dict__)
733: (8)                    widthsObject = self.reserveObject('font widths')
734: (8)                    self.writeObject(widthsObject, fontinfo.dvifont.widths)
735: (8)                    fontdictObject = self.reserveObject('font dictionary')
736: (8)                    fontdict = {
737: (12)                       'Type':      Name('Font'),
738: (12)                       'Subtype':   Name('Type1'),
739: (12)                       'FirstChar': 0,
740: (12)                       'LastChar':  len(fontinfo.dvifont.widths) - 1,
741: (12)                       'Widths':    widthsObject,
742: (12)                       }
743: (8)                    if fontinfo.encodingfile is not None:
744: (12)                       fontdict['Encoding'] = {
745: (16)                           'Type': Name('Encoding'),
746: (16)                           'Differences': [
747: (20)                               0, *map(Name, dviread._parse_enc(fontinfo.encodingfile))],
748: (12)                       }
749: (8)                    if fontinfo.fontfile is None:
750: (12)                       _log.warning(
751: (16)                           "Because of TeX configuration (pdftex.map, see updmap option "
752: (16)                           "pdftexDownloadBase14) the font %s is not embedded. This is "
753: (16)                           "deprecated as of PDF 1.5 and it may cause the consumer "
754: (16)                           "application to show something that was not intended.",
```

```
(16)                            fontinfo.basefont)
(12)                        fontdict['BaseFont'] = Name(fontinfo.basefont)
(12)                        self.writeObject(fontdictObject, fontdict)
(12)                        return fontdictObject
(8)                     t1font = _type1font.Type1Font(fontinfo.fontfile)
(8)                     if fontinfo.effects:
(12)                        t1font = t1font.transform(fontinfo.effects)
(8)                     fontdict['BaseFont'] = Name(t1font.prop['FontName'])
(8)                     effects = (fontinfo.effects.get('slant', 0.0),
(19)                               fontinfo.effects.get('extend', 1.0))
(8)                     fontdesc = self.type1Descriptors.get((fontinfo.fontfile, effects))
(8)                     if fontdesc is None:
(12)                        fontdesc = self.createType1Descriptor(t1font, fontinfo.fontfile)
(12)                        self.type1Descriptors[(fontinfo.fontfile, effects)] = fontdesc
(8)                     fontdict['FontDescriptor'] = fontdesc
(8)                     self.writeObject(fontdictObject, fontdict)
(8)                     return fontdictObject
(4)                 def createType1Descriptor(self, t1font, fontfile):
(8)                     fontdescObject = self.reserveObject('font descriptor')
(8)                     fontfileObject = self.reserveObject('font file')
(8)                     italic_angle = t1font.prop['ItalicAngle']
(8)                     fixed_pitch = t1font.prop['isFixedPitch']
(8)                     flags = 0
(8)                     if fixed_pitch:
(12)                        flags |= 1 << 0
(8)                     if 0:
(12)                        flags |= 1 << 1
(8)                     if 1:
(12)                        flags |= 1 << 2
(8)                     else:
(12)                        flags |= 1 << 5
(8)                     if italic_angle:
(12)                        flags |= 1 << 6
(8)                     if 0:
(12)                        flags |= 1 << 16
(8)                     if 0:
(12)                        flags |= 1 << 17
(8)                     if 0:
(12)                        flags |= 1 << 18
(8)                     ft2font = get_font(fontfile)
(8)                     descriptor = {
(12)                        'Type':        Name('FontDescriptor'),
(12)                        'FontName':    Name(t1font.prop['FontName']),
(12)                        'Flags':       flags,
(12)                        'FontBBox':    ft2font.bbox,
(12)                        'ItalicAngle': italic_angle,
(12)                        'Ascent':      ft2font.ascender,
(12)                        'Descent':     ft2font.descender,
(12)                        'CapHeight':   1000,  # TODO: find this out
(12)                        'XHeight':     500,  # TODO: this one too
(12)                        'FontFile':    fontfileObject,
(12)                        'FontFamily':  t1font.prop['FamilyName'],
(12)                        'StemV':       50,  # TODO
(12)                        }
(8)                     self.writeObject(fontdescObject, descriptor)
(8)                     self.outputStream(fontfileObject, b"".join(t1font.parts[:2]),
(26)                                      extra={'Length1': len(t1font.parts[0]),
(33)                                             'Length2': len(t1font.parts[1]),
(33)                                             'Length3': 0})
(8)                     return fontdescObject
(4)                 def _get_xobject_glyph_name(self, filename, glyph_name):
(8)                     Fx = self.fontName(filename)
(8)                     return "-".join([
(12)                        Fx.name.decode(),
(12)                        os.path.splitext(os.path.basename(filename))[0],
(12)                        glyph_name])
(4)                 _identityToUnicodeCMap = b"""/CIDInit /ProcSet findresource begin
(0)             12 dict begin
(0)             begincmap
```

```
824: (0)            /CIDSystemInfo
825: (0)            << /Registry (Adobe)
826: (3)               /Ordering (UCS)
827: (3)               /Supplement 0
828: (0)            >> def
829: (0)            /CMapName /Adobe-Identity-UCS def
830: (0)            /CMapType 2 def
831: (0)            1 begincodespacerange
832: (0)            <0000> <ffff>
833: (0)            endcodespacerange
834: (0)            %d beginbfrange
835: (0)            %s
836: (0)            endbfrange
837: (0)            endcmap
838: (0)            CMapName currentdict /CMap defineresource pop
839: (0)            end
840: (0)            end"""
841: (4)                def embedTTF(self, filename, characters):
842: (8)                    """Embed the TTF font from the named file into the document."""
843: (8)                    font = get_font(filename)
844: (8)                    fonttype = mpl.rcParams['pdf.fonttype']
845: (8)                    def cvt(length, upe=font.units_per_EM, nearest=True):
846: (12)                       """Convert font coordinates to PDF glyph coordinates."""
847: (12)                       value = length / upe * 1000
848: (12)                       if nearest:
849: (16)                           return round(value)
850: (12)                       if value < 0:
851: (16)                           return math.floor(value)
852: (12)                       else:
853: (16)                           return math.ceil(value)
854: (8)                    def embedTTFType3(font, characters, descriptor):
855: (12)                       """The Type 3-specific part of embedding a Truetype font"""
856: (12)                       widthsObject = self.reserveObject('font widths')
857: (12)                       fontdescObject = self.reserveObject('font descriptor')
858: (12)                       fontdictObject = self.reserveObject('font dictionary')
859: (12)                       charprocsObject = self.reserveObject('character procs')
860: (12)                       differencesArray = []
861: (12)                       firstchar, lastchar = 0, 255
862: (12)                       bbox = [cvt(x, nearest=False) for x in font.bbox]
863: (12)                       fontdict = {
864: (16)                           'Type': Name('Font'),
865: (16)                           'BaseFont': ps_name,
866: (16)                           'FirstChar': firstchar,
867: (16)                           'LastChar': lastchar,
868: (16)                           'FontDescriptor': fontdescObject,
869: (16)                           'Subtype': Name('Type3'),
870: (16)                           'Name': descriptor['FontName'],
871: (16)                           'FontBBox': bbox,
872: (16)                           'FontMatrix': [.001, 0, 0, .001, 0, 0],
873: (16)                           'CharProcs': charprocsObject,
874: (16)                           'Encoding': {
875: (20)                               'Type': Name('Encoding'),
876: (20)                               'Differences': differencesArray},
877: (16)                           'Widths': widthsObject
878: (16)                           }
879: (12)                       from encodings import cp1252
880: (12)                       def get_char_width(charcode):
881: (16)                           s = ord(cp1252.decoding_table[charcode])
882: (16)                           width = font.load_char(
883: (20)                               s, flags=LOAD_NO_SCALE | LOAD_NO_HINTING).horiAdvance
884: (16)                           return cvt(width)
885: (12)                       with warnings.catch_warnings():
886: (16)                           warnings.filterwarnings("ignore")
887: (16)                           widths = [get_char_width(charcode)
888: (26)                                     for charcode in range(firstchar, lastchar+1)]
889: (12)                       descriptor['MaxWidth'] = max(widths)
890: (12)                       glyph_ids = []
891: (12)                       differences = []
892: (12)                       multi_byte_chars = set()
```

```
(12)                        for c in characters:
(16)                            ccode = c
(16)                            gind = font.get_char_index(ccode)
(16)                            glyph_ids.append(gind)
(16)                            glyph_name = font.get_glyph_name(gind)
(16)                            if ccode <= 255:
(20)                                differences.append((ccode, glyph_name))
(16)                            else:
(20)                                multi_byte_chars.add(glyph_name)
(12)                        differences.sort()
(12)                        last_c = -2
(12)                        for c, name in differences:
(16)                            if c != last_c + 1:
(20)                                differencesArray.append(c)
(16)                            differencesArray.append(Name(name))
(16)                            last_c = c
(12)                        rawcharprocs = _get_pdf_charprocs(filename, glyph_ids)
(12)                        charprocs = {}
(12)                        for charname in sorted(rawcharprocs):
(16)                            stream = rawcharprocs[charname]
(16)                            charprocDict = {}
(16)                            if charname in multi_byte_chars:
(20)                                charprocDict = {'Type': Name('XObject'),
(36)                                                'Subtype': Name('Form'),
(36)                                                'BBox': bbox}
(20)                                stream = stream[stream.find(b"d1") + 2:]
(16)                            charprocObject = self.reserveObject('charProc')
(16)                            self.outputStream(charprocObject, stream, extra=charprocDict)
(16)                            if charname in multi_byte_chars:
(20)                                name = self._get_xobject_glyph_name(filename, charname)
(20)                                self.multi_byte_charprocs[name] = charprocObject
(16)                            else:
(20)                                charprocs[charname] = charprocObject
(12)                        self.writeObject(fontdictObject, fontdict)
(12)                        self.writeObject(fontdescObject, descriptor)
(12)                        self.writeObject(widthsObject, widths)
(12)                        self.writeObject(charprocsObject, charprocs)
(12)                        return fontdictObject
(8)                     def embedTTFType42(font, characters, descriptor):
(12)                        """The Type 42-specific part of embedding a Truetype font"""
(12)                        fontdescObject = self.reserveObject('font descriptor')
(12)                        cidFontDictObject = self.reserveObject('CID font dictionary')
(12)                        type0FontDictObject = self.reserveObject('Type 0 font dictionary')
(12)                        cidToGidMapObject = self.reserveObject('CIDToGIDMap stream')
(12)                        fontfileObject = self.reserveObject('font file stream')
(12)                        wObject = self.reserveObject('Type 0 widths')
(12)                        toUnicodeMapObject = self.reserveObject('ToUnicode map')
(12)                        subset_str = "".join(chr(c) for c in characters)
(12)                        _log.debug("SUBSET %s characters: %s", filename, subset_str)
(12)                        fontdata = _backend_pdf_ps.get_glyphs_subset(filename, subset_str)
(12)                        _log.debug(
(16)                            "SUBSET %s %d -> %d", filename,
(16)                            os.stat(filename).st_size, fontdata.getbuffer().nbytes
(12)                        )
(12)                        full_font = font
(12)                        font = FT2Font(fontdata)
(12)                        cidFontDict = {
(16)                            'Type': Name('Font'),
(16)                            'Subtype': Name('CIDFontType2'),
(16)                            'BaseFont': ps_name,
(16)                            'CIDSystemInfo': {
(20)                                'Registry': 'Adobe',
(20)                                'Ordering': 'Identity',
(20)                                'Supplement': 0},
(16)                            'FontDescriptor': fontdescObject,
(16)                            'W': wObject,
(16)                            'CIDToGIDMap': cidToGidMapObject
(16)                            }
(12)                        type0FontDict = {
```

```
962: (16)                            'Type': Name('Font'),
963: (16)                            'Subtype': Name('Type0'),
964: (16)                            'BaseFont': ps_name,
965: (16)                            'Encoding': Name('Identity-H'),
966: (16)                            'DescendantFonts': [cidFontDictObject],
967: (16)                            'ToUnicode': toUnicodeMapObject
968: (16)                            }
969: (12)                        descriptor['FontFile2'] = fontfileObject
970: (12)                        self.outputStream(
971: (16)                            fontfileObject, fontdata.getvalue(),
972: (16)                            extra={'Length1': fontdata.getbuffer().nbytes})
973: (12)                        cid_to_gid_map = ['\0'] * 65536
974: (12)                        widths = []
975: (12)                        max_ccode = 0
976: (12)                        for c in characters:
977: (16)                            ccode = c
978: (16)                            gind = font.get_char_index(ccode)
979: (16)                            glyph = font.load_char(ccode,
980: (39)                                                   flags=LOAD_NO_SCALE | LOAD_NO_HINTING)
981: (16)                            widths.append((ccode, cvt(glyph.horiAdvance)))
982: (16)                            if ccode < 65536:
983: (20)                                cid_to_gid_map[ccode] = chr(gind)
984: (16)                            max_ccode = max(ccode, max_ccode)
985: (12)                        widths.sort()
986: (12)                        cid_to_gid_map = cid_to_gid_map[:max_ccode + 1]
987: (12)                        last_ccode = -2
988: (12)                        w = []
989: (12)                        max_width = 0
990: (12)                        unicode_groups = []
991: (12)                        for ccode, width in widths:
992: (16)                            if ccode != last_ccode + 1:
993: (20)                                w.append(ccode)
994: (20)                                w.append([width])
995: (20)                                unicode_groups.append([ccode, ccode])
996: (16)                            else:
997: (20)                                w[-1].append(width)
998: (20)                                unicode_groups[-1][1] = ccode
999: (16)                            max_width = max(max_width, width)
1000: (16)                           last_ccode = ccode
1001: (12)                       unicode_bfrange = []
1002: (12)                       for start, end in unicode_groups:
1003: (16)                           if start > 65535:
1004: (20)                               continue
1005: (16)                           end = min(65535, end)
1006: (16)                           unicode_bfrange.append(
1007: (20)                               b"<%04x> <%04x> [%s]" %
1008: (20)                               (start, end,
1009: (21)                                b" ".join(b"<%04x>" % x for x in range(start, end+1))))
1010: (12)                       unicode_cmap = (self._identityToUnicodeCMap %
1011: (28)                                       (len(unicode_groups),
b"\n".join(unicode_bfrange)))
1012: (12)                       glyph_ids = []
1013: (12)                       for ccode in characters:
1014: (16)                           if not _font_supports_glyph(fonttype, ccode):
1015: (20)                               gind = full_font.get_char_index(ccode)
1016: (20)                               glyph_ids.append(gind)
1017: (12)                       bbox = [cvt(x, nearest=False) for x in full_font.bbox]
1018: (12)                       rawcharprocs = _get_pdf_charprocs(filename, glyph_ids)
1019: (12)                       for charname in sorted(rawcharprocs):
1020: (16)                           stream = rawcharprocs[charname]
1021: (16)                           charprocDict = {'Type': Name('XObject'),
1022: (32)                                           'Subtype': Name('Form'),
1023: (32)                                           'BBox': bbox}
1024: (16)                           stream = stream[stream.find(b"d1") + 2:]
1025: (16)                           charprocObject = self.reserveObject('charProc')
1026: (16)                           self.outputStream(charprocObject, stream, extra=charprocDict)
1027: (16)                           name = self._get_xobject_glyph_name(filename, charname)
1028: (16)                           self.multi_byte_charprocs[name] = charprocObject
1029: (12)                       cid_to_gid_map = "".join(cid_to_gid_map).encode("utf-16be")
```

```
1030: (12)                      self.outputStream(cidToGidMapObject, cid_to_gid_map)
1031: (12)                      self.outputStream(toUnicodeMapObject, unicode_cmap)
1032: (12)                      descriptor['MaxWidth'] = max_width
1033: (12)                      self.writeObject(cidFontDictObject, cidFontDict)
1034: (12)                      self.writeObject(type0FontDictObject, type0FontDict)
1035: (12)                      self.writeObject(fontdescObject, descriptor)
1036: (12)                      self.writeObject(wObject, w)
1037: (12)                      return type0FontDictObject
1038: (8)                   ps_name = self._get_subsetted_psname(
1039: (12)                      font.postscript_name,
1040: (12)                      font.get_charmap()
1041: (8)                   )
1042: (8)                   ps_name = ps_name.encode('ascii', 'replace')
1043: (8)                   ps_name = Name(ps_name)
1044: (8)                   pclt = font.get_sfnt_table('pclt') or {'capHeight': 0, 'xHeight': 0}
1045: (8)                   post = font.get_sfnt_table('post') or {'italicAngle': (0, 0)}
1046: (8)                   ff = font.face_flags
1047: (8)                   sf = font.style_flags
1048: (8)                   flags = 0
1049: (8)                   symbolic = False  # ps_name.name in ('Cmsy10', 'Cmmi10', 'Cmex10')
1050: (8)                   if ff & FIXED_WIDTH:
1051: (12)                      flags |= 1 << 0
1052: (8)                   if 0:  # TODO: serif
1053: (12)                      flags |= 1 << 1
1054: (8)                   if symbolic:
1055: (12)                      flags |= 1 << 2
1056: (8)                   else:
1057: (12)                      flags |= 1 << 5
1058: (8)                   if sf & ITALIC:
1059: (12)                      flags |= 1 << 6
1060: (8)                   if 0:  # TODO: all caps
1061: (12)                      flags |= 1 << 16
1062: (8)                   if 0:  # TODO: small caps
1063: (12)                      flags |= 1 << 17
1064: (8)                   if 0:  # TODO: force bold
1065: (12)                      flags |= 1 << 18
1066: (8)                   descriptor = {
1067: (12)                      'Type': Name('FontDescriptor'),
1068: (12)                      'FontName': ps_name,
1069: (12)                      'Flags': flags,
1070: (12)                      'FontBBox': [cvt(x, nearest=False) for x in font.bbox],
1071: (12)                      'Ascent': cvt(font.ascender, nearest=False),
1072: (12)                      'Descent': cvt(font.descender, nearest=False),
1073: (12)                      'CapHeight': cvt(pclt['capHeight'], nearest=False),
1074: (12)                      'XHeight': cvt(pclt['xHeight']),
1075: (12)                      'ItalicAngle': post['italicAngle'][1],  # ???
1076: (12)                      'StemV': 0  # ???
1077: (12)                      }
1078: (8)                   if fonttype == 3:
1079: (12)                      return embedTTFType3(font, characters, descriptor)
1080: (8)                   elif fonttype == 42:
1081: (12)                      return embedTTFType42(font, characters, descriptor)
1082: (4)               def alphaState(self, alpha):
1083: (8)                   """Return name of an ExtGState that sets alpha to the given value."""
1084: (8)                   state = self.alphaStates.get(alpha, None)
1085: (8)                   if state is not None:
1086: (12)                      return state[0]
1087: (8)                   name = next(self._alpha_state_seq)
1088: (8)                   self.alphaStates[alpha] = \
1089: (12)                      (name, {'Type': Name('ExtGState'),
1090: (20)                              'CA': alpha[0], 'ca': alpha[1]})
1091: (8)                   return name
1092: (4)               def _soft_mask_state(self, smask):
1093: (8)                   """
1094: (8)                   Return an ExtGState that sets the soft mask to the given shading.
1095: (8)                   Parameters
1096: (8)                   ----------
1097: (8)                   smask : Reference
1098: (12)                      Reference to a shading in DeviceGray color space, whose luminosity
```

```
(12)                    is to be used as the alpha channel.
(8)                 Returns
(8)                 -------
(8)                 Name
(8)                 """
(8)                 state = self._soft_mask_states.get(smask, None)
(8)                 if state is not None:
(12)                    return state[0]
(8)                 name = next(self._soft_mask_seq)
(8)                 groupOb = self.reserveObject('transparency group for soft mask')
(8)                 self._soft_mask_states[smask] = (
(12)                    name,
(12)                    {
(16)                        'Type': Name('ExtGState'),
(16)                        'AIS': False,
(16)                        'SMask': {
(20)                            'Type': Name('Mask'),
(20)                            'S': Name('Luminosity'),
(20)                            'BC': [1],
(20)                            'G': groupOb
(16)                        }
(12)                    }
(8)                 )
(8)                 self._soft_mask_groups.append((
(12)                    groupOb,
(12)                    {
(16)                        'Type': Name('XObject'),
(16)                        'Subtype': Name('Form'),
(16)                        'FormType': 1,
(16)                        'Group': {
(20)                            'S': Name('Transparency'),
(20)                            'CS': Name('DeviceGray')
(16)                        },
(16)                        'Matrix': [1, 0, 0, 1, 0, 0],
(16)                        'Resources': {'Shading': {'S': smask}},
(16)                        'BBox': [0, 0, 1, 1]
(12)                    },
(12)                    [Name('S'), Op.shading]
(8)                 ))
(8)                 return name
(4)             def writeExtGSTates(self):
(8)                 self.writeObject(
(12)                    self._extGStateObject,
(12)                    dict([
(16)                        *self.alphaStates.values(),
(16)                        *self._soft_mask_states.values()
(12)                    ])
(8)                 )
(4)             def _write_soft_mask_groups(self):
(8)                 for ob, attributes, content in self._soft_mask_groups:
(12)                    self.beginStream(ob.id, None, attributes)
(12)                    self.output(*content)
(12)                    self.endStream()
(4)             def hatchPattern(self, hatch_style):
(8)                 if hatch_style is not None:
(12)                    edge, face, hatch = hatch_style
(12)                    if edge is not None:
(16)                        edge = tuple(edge)
(12)                    if face is not None:
(16)                        face = tuple(face)
(12)                    hatch_style = (edge, face, hatch)
(8)                 pattern = self.hatchPatterns.get(hatch_style, None)
(8)                 if pattern is not None:
(12)                    return pattern
(8)                 name = next(self._hatch_pattern_seq)
(8)                 self.hatchPatterns[hatch_style] = name
(8)                 return name
(4)             def writeHatches(self):
(8)                 hatchDict = dict()
```

```
(8)                 sidelen = 72.0
(8)                 for hatch_style, name in self.hatchPatterns.items():
(12)                    ob = self.reserveObject('hatch pattern')
(12)                    hatchDict[name] = ob
(12)                    res = {'Procsets':
(19)                           [Name(x) for x in "PDF Text ImageB ImageC ImageI".split()]}
(12)                    self.beginStream(
(16)                        ob.id, None,
(16)                        {'Type': Name('Pattern'),
(17)                         'PatternType': 1, 'PaintType': 1, 'TilingType': 1,
(17)                         'BBox': [0, 0, sidelen, sidelen],
(17)                         'XStep': sidelen, 'YStep': sidelen,
(17)                         'Resources': res,
(17)                         'Matrix': [1, 0, 0, 1, 0, self.height * 72]})
(12)                    stroke_rgb, fill_rgb, hatch = hatch_style
(12)                    self.output(stroke_rgb[0], stroke_rgb[1], stroke_rgb[2],
(24)                                Op.setrgb_stroke)
(12)                    if fill_rgb is not None:
(16)                        self.output(fill_rgb[0], fill_rgb[1], fill_rgb[2],
(28)                                    Op.setrgb_nonstroke,
(28)                                    0, 0, sidelen, sidelen, Op.rectangle,
(28)                                    Op.fill)
(12)                    self.output(mpl.rcParams['hatch.linewidth'], Op.setlinewidth)
(12)                    self.output(*self.pathOperations(
(16)                        Path.hatch(hatch),
(16)                        Affine2D().scale(sidelen),
(16)                        simplify=False))
(12)                    self.output(Op.fill_stroke)
(12)                    self.endStream()
(8)                 self.writeObject(self.hatchObject, hatchDict)
(4)             def addGouraudTriangles(self, points, colors):
(8)                 """
(8)                 Add a Gouraud triangle shading.
(8)                 Parameters
(8)                 ----------
(8)                 points : np.ndarray
(12)                    Triangle vertices, shape (n, 3, 2)
(12)                    where n = number of triangles, 3 = vertices, 2 = x, y.
(8)                 colors : np.ndarray
(12)                    Vertex colors, shape (n, 3, 1) or (n, 3, 4)
(12)                    as with points, but last dimension is either (gray,)
(12)                    or (r, g, b, alpha).
(8)                 Returns
(8)                 -------
(8)                 Name, Reference
(8)                 """
(8)                 name = Name('GT%d' % len(self.gouraudTriangles))
(8)                 ob = self.reserveObject(f'Gouraud triangle {name}')
(8)                 self.gouraudTriangles.append((name, ob, points, colors))
(8)                 return name, ob
(4)             def writeGouraudTriangles(self):
(8)                 gouraudDict = dict()
(8)                 for name, ob, points, colors in self.gouraudTriangles:
(12)                    gouraudDict[name] = ob
(12)                    shape = points.shape
(12)                    flat_points = points.reshape((shape[0] * shape[1], 2))
(12)                    colordim = colors.shape[2]
(12)                    assert colordim in (1, 4)
(12)                    flat_colors = colors.reshape((shape[0] * shape[1], colordim))
(12)                    if colordim == 4:
(16)                        colordim = 3
(12)                    points_min = np.min(flat_points, axis=0) - (1 << 8)
(12)                    points_max = np.max(flat_points, axis=0) + (1 << 8)
(12)                    factor = 0xffffffff / (points_max - points_min)
(12)                    self.beginStream(
(16)                        ob.id, None,
(16)                        {'ShadingType': 4,
(17)                         'BitsPerCoordinate': 32,
(17)                         'BitsPerComponent': 8,
```

```
(17)                            'BitsPerFlag': 8,
(17)                            'ColorSpace': Name(
(21)                                'DeviceRGB' if colordim == 3 else 'DeviceGray'
(17)                            ),
(17)                            'AntiAlias': False,
(17)                            'Decode': ([points_min[0], points_max[0],
(29)                                       points_min[1], points_max[1]]
(28)                                      + [0, 1] * colordim),
(17)                            })
(12)                        streamarr = np.empty(
(16)                            (shape[0] * shape[1],),
(16)                            dtype=[('flags', 'u1'),
(23)                                   ('points', '>u4', (2,)),
(23)                                   ('colors', 'u1', (colordim,))])
(12)                        streamarr['flags'] = 0
(12)                        streamarr['points'] = (flat_points - points_min) * factor
(12)                        streamarr['colors'] = flat_colors[:, :colordim] * 255.0
(12)                        self.write(streamarr.tobytes())
(12)                        self.endStream()
(8)                     self.writeObject(self.gouraudObject, gouraudDict)
(4)                 def imageObject(self, image):
(8)                     """Return name of an image XObject representing the given image."""
(8)                     entry = self._images.get(id(image), None)
(8)                     if entry is not None:
(12)                        return entry[1]
(8)                     name = next(self._image_seq)
(8)                     ob = self.reserveObject(f'image {name}')
(8)                     self._images[id(image)] = (image, name, ob)
(8)                     return name
(4)                 def _unpack(self, im):
(8)                     """
(8)                     Unpack image array *im* into ``(data, alpha)``, which have shape
(8)                     ``(height, width, 3)`` (RGB) or ``(height, width, 1)`` (grayscale or
(8)                     alpha), except that alpha is None if the image is fully opaque.
(8)                     """
(8)                     im = im[::-1]
(8)                     if im.ndim == 2:
(12)                        return im, None
(8)                     else:
(12)                        rgb = im[:, :, :3]
(12)                        rgb = np.array(rgb, order='C')
(12)                        if im.shape[2] == 4:
(16)                            alpha = im[:, :, 3][..., None]
(16)                            if np.all(alpha == 255):
(20)                                alpha = None
(16)                            else:
(20)                                alpha = np.array(alpha, order='C')
(12)                        else:
(16)                            alpha = None
(12)                        return rgb, alpha
(4)                 def _writePng(self, img):
(8)                     """
(8)                     Write the image *img* into the pdf file using png
(8)                     predictors with Flate compression.
(8)                     """
(8)                     buffer = BytesIO()
(8)                     img.save(buffer, format="png")
(8)                     buffer.seek(8)
(8)                     png_data = b''
(8)                     bit_depth = palette = None
(8)                     while True:
(12)                        length, type = struct.unpack(b'!L4s', buffer.read(8))
(12)                        if type in [b'IHDR', b'PLTE', b'IDAT']:
(16)                            data = buffer.read(length)
(16)                            if len(data) != length:
(20)                                raise RuntimeError("truncated data")
(16)                            if type == b'IHDR':
(20)                                bit_depth = int(data[8])
(16)                            elif type == b'PLTE':
```

```
1306: (20)                                palette = data
1307: (16)                            elif type == b'IDAT':
1308: (20)                                png_data += data
1309: (12)                        elif type == b'IEND':
1310: (16)                            break
1311: (12)                        else:
1312: (16)                            buffer.seek(length, 1)
1313: (12)                        buffer.seek(4, 1)   # skip CRC
1314: (8)                     return png_data, bit_depth, palette
1315: (4)                 def _writeImg(self, data, id, smask=None):
1316: (8)                     """
1317: (8)                     Write the image *data*, of shape ``(height, width, 1)`` (grayscale) or
1318: (8)                     ``(height, width, 3)`` (RGB), as pdf object *id* and with the soft
mask
1319: (8)                     (alpha channel) *smask*, which should be either None or a ``(height,
1320: (8)                     width, 1)`` array.
1321: (8)                     """
1322: (8)                     height, width, color_channels = data.shape
1323: (8)                     obj = {'Type': Name('XObject'),
1324: (15)                            'Subtype': Name('Image'),
1325: (15)                            'Width': width,
1326: (15)                            'Height': height,
1327: (15)                            'ColorSpace': Name({1: 'DeviceGray', 3: 'DeviceRGB'}
[color_channels]),
1328: (15)                            'BitsPerComponent': 8}
1329: (8)                     if smask:
1330: (12)                        obj['SMask'] = smask
1331: (8)                     if mpl.rcParams['pdf.compression']:
1332: (12)                        if data.shape[-1] == 1:
1333: (16)                            data = data.squeeze(axis=-1)
1334: (12)                        png = {'Predictor': 10, 'Colors': color_channels, 'Columns':
width}
1335: (12)                        img = Image.fromarray(data)
1336: (12)                        img_colors = img.getcolors(maxcolors=256)
1337: (12)                        if color_channels == 3 and img_colors is not None:
1338: (16)                            num_colors = len(img_colors)
1339: (16)                            palette = np.array([comp for _, color in img_colors for comp
in color],
1340: (35)                                               dtype=np.uint8)
1341: (16)                            palette24 = ((palette[0::3].astype(np.uint32) << 16) |
1342: (29)                                         (palette[1::3].astype(np.uint32) << 8) |
1343: (29)                                         palette[2::3])
1344: (16)                            rgb24 = ((data[:, :, 0].astype(np.uint32) << 16) |
1345: (25)                                     (data[:, :, 1].astype(np.uint32) << 8) |
1346: (25)                                     data[:, :, 2])
1347: (16)                            indices = np.argsort(palette24).astype(np.uint8)
1348: (16)                            rgb8 = indices[np.searchsorted(palette24, rgb24,
sorter=indices)]
1349: (16)                            img = Image.fromarray(rgb8, mode='P')
1350: (16)                            img.putpalette(palette)
1351: (16)                            png_data, bit_depth, palette = self._writePng(img)
1352: (16)                            if bit_depth is None or palette is None:
1353: (20)                                raise RuntimeError("invalid PNG header")
1354: (16)                            palette = palette[:num_colors * 3]  # Trim padding; remove for
Pillow>=9
1355: (16)                            obj['ColorSpace'] = [Name('Indexed'), Name('DeviceRGB'),
1356: (37)                                                 num_colors - 1, palette]
1357: (16)                            obj['BitsPerComponent'] = bit_depth
1358: (16)                            png['Colors'] = 1
1359: (16)                            png['BitsPerComponent'] = bit_depth
1360: (12)                        else:
1361: (16)                            png_data, _, _ = self._writePng(img)
1362: (8)                     else:
1363: (12)                        png = None
1364: (8)                     self.beginStream(
1365: (12)                        id,
1366: (12)                        self.reserveObject('length of image stream'),
1367: (12)                        obj,
1368: (12)                        png=png
```

```
(12)                        )
(8)                     if png:
(12)                        self.currentstream.write(png_data)
(8)                     else:
(12)                        self.currentstream.write(data.tobytes())
(8)                     self.endStream()
(4)                 def writeImages(self):
(8)                     for img, name, ob in self._images.values():
(12)                        data, adata = self._unpack(img)
(12)                        if adata is not None:
(16)                            smaskObject = self.reserveObject("smask")
(16)                            self._writeImg(adata, smaskObject.id)
(12)                        else:
(16)                            smaskObject = None
(12)                        self._writeImg(data, ob.id, smaskObject)
(4)                 def markerObject(self, path, trans, fill, stroke, lw, joinstyle,
(21)                                 capstyle):
(8)                     """Return name of a marker XObject representing the given path."""
(8)                     pathops = self.pathOperations(path, trans, simplify=False)
(8)                     key = (tuple(pathops), bool(fill), bool(stroke), joinstyle, capstyle)
(8)                     result = self.markers.get(key)
(8)                     if result is None:
(12)                        name = Name('M%d' % len(self.markers))
(12)                        ob = self.reserveObject('marker %d' % len(self.markers))
(12)                        bbox = path.get_extents(trans)
(12)                        self.markers[key] = [name, ob, bbox, lw]
(8)                     else:
(12)                        if result[-1] < lw:
(16)                            result[-1] = lw
(12)                        name = result[0]
(8)                     return name
(4)                 def writeMarkers(self):
(8)                     for ((pathops, fill, stroke, joinstyle, capstyle),
(13)                          (name, ob, bbox, lw)) in self.markers.items():
(12)                        bbox = bbox.padded(lw * 5)
(12)                        self.beginStream(
(16)                            ob.id, None,
(16)                            {'Type': Name('XObject'), 'Subtype': Name('Form'),
(17)                             'BBox': list(bbox.extents)})
(12)                        self.output(GraphicsContextPdf.joinstyles[joinstyle],
(24)                                    Op.setlinejoin)
(12)                        self.output(GraphicsContextPdf.capstyles[capstyle], Op.setlinecap)
(12)                        self.output(*pathops)
(12)                        self.output(Op.paint_path(fill, stroke))
(12)                        self.endStream()
(4)                 def pathCollectionObject(self, gc, path, trans, padding, filled, stroked):
(8)                     name = Name('P%d' % len(self.paths))
(8)                     ob = self.reserveObject('path %d' % len(self.paths))
(8)                     self.paths.append(
(12)                        (name, path, trans, ob, gc.get_joinstyle(), gc.get_capstyle(),
(13)                         padding, filled, stroked))
(8)                     return name
(4)                 def writePathCollectionTemplates(self):
(8)                     for (name, path, trans, ob, joinstyle, capstyle, padding, filled,
(13)                         stroked) in self.paths:
(12)                        pathops = self.pathOperations(path, trans, simplify=False)
(12)                        bbox = path.get_extents(trans)
(12)                        if not np.all(np.isfinite(bbox.extents)):
(16)                            extents = [0, 0, 0, 0]
(12)                        else:
(16)                            bbox = bbox.padded(padding)
(16)                            extents = list(bbox.extents)
(12)                        self.beginStream(
(16)                            ob.id, None,
(16)                            {'Type': Name('XObject'), 'Subtype': Name('Form'),
(17)                             'BBox': extents})
(12)                        self.output(GraphicsContextPdf.joinstyles[joinstyle],
(24)                                    Op.setlinejoin)
(12)                        self.output(GraphicsContextPdf.capstyles[capstyle], Op.setlinecap)
```

```
1438: (12)                      self.output(*pathops)
1439: (12)                      self.output(Op.paint_path(filled, stroked))
1440: (12)                      self.endStream()
1441: (4)               @staticmethod
1442: (4)               def pathOperations(path, transform, clip=None, simplify=None,
sketch=None):
1443: (8)                   return [Verbatim(_path.convert_to_string(
1444: (12)                      path, transform, clip, simplify, sketch,
1445: (12)                      6,
1446: (12)                      [Op.moveto.value, Op.lineto.value, b'', Op.curveto.value,
1447: (13)                       Op.closepath.value],
1448: (12)                      True))]
1449: (4)               def writePath(self, path, transform, clip=False, sketch=None):
1450: (8)                   if clip:
1451: (12)                      clip = (0.0, 0.0, self.width * 72, self.height * 72)
1452: (12)                      simplify = path.should_simplify
1453: (8)                   else:
1454: (12)                      clip = None
1455: (12)                      simplify = False
1456: (8)                   cmds = self.pathOperations(path, transform, clip, simplify=simplify,
1457: (35)                                             sketch=sketch)
1458: (8)                   self.output(*cmds)
1459: (4)               def reserveObject(self, name=''):
1460: (8)                   """
1461: (8)                   Reserve an ID for an indirect object.
1462: (8)                   The name is used for debugging in case we forget to print out
1463: (8)                   the object with writeObject.
1464: (8)                   """
1465: (8)                   id = next(self._object_seq)
1466: (8)                   self.xrefTable.append([None, 0, name])
1467: (8)                   return Reference(id)
1468: (4)               def recordXref(self, id):
1469: (8)                   self.xrefTable[id][0] = self.fh.tell() - self.tell_base
1470: (4)               def writeObject(self, object, contents):
1471: (8)                   self.recordXref(object.id)
1472: (8)                   object.write(contents, self)
1473: (4)               def writeXref(self):
1474: (8)                   """Write out the xref table."""
1475: (8)                   self.startxref = self.fh.tell() - self.tell_base
1476: (8)                   self.write(b"xref\n0 %d\n" % len(self.xrefTable))
1477: (8)                   for i, (offset, generation, name) in enumerate(self.xrefTable):
1478: (12)                      if offset is None:
1479: (16)                          raise AssertionError(
1480: (20)                              'No offset for object %d (%s)' % (i, name))
1481: (12)                      else:
1482: (16)                          key = b"f" if name == 'the zero object' else b"n"
1483: (16)                          text = b"%010d %05d %b \n" % (offset, generation, key)
1484: (16)                          self.write(text)
1485: (4)               def writeInfoDict(self):
1486: (8)                   """Write out the info dictionary, checking it for good form"""
1487: (8)                   self.infoObject = self.reserveObject('info')
1488: (8)                   self.writeObject(self.infoObject, self.infoDict)
1489: (4)               def writeTrailer(self):
1490: (8)                   """Write out the PDF trailer."""
1491: (8)                   self.write(b"trailer\n")
1492: (8)                   self.write(pdfRepr(
1493: (12)                      {'Size': len(self.xrefTable),
1494: (13)                       'Root': self.rootObject,
1495: (13)                       'Info': self.infoObject}))
1496: (8)                   self.write(b"\nstartxref\n%d\n%%%%EOF\n" % self.startxref)
1497: (0)           class RendererPdf(_backend_pdf_ps.RendererPDFPSBase):
1498: (4)               _afm_font_dir = cbook._get_data_path("fonts/pdfcorefonts")
1499: (4)               _use_afm_rc_name = "pdf.use14corefonts"
1500: (4)               def __init__(self, file, image_dpi, height, width):
1501: (8)                   super().__init__(width, height)
1502: (8)                   self.file = file
1503: (8)                   self.gc = self.new_gc()
1504: (8)                   self.image_dpi = image_dpi
1505: (4)               def finalize(self):
```

```
1506: (8)                    self.file.output(*self.gc.finalize())
1507: (4)                def check_gc(self, gc, fillcolor=None):
1508: (8)                    orig_fill = getattr(gc, '_fillcolor', (0., 0., 0.))
1509: (8)                    gc._fillcolor = fillcolor
1510: (8)                    orig_alphas = getattr(gc, '_effective_alphas', (1.0, 1.0))
1511: (8)                    if gc.get_rgb() is None:
1512: (12)                       gc.set_foreground((0, 0, 0, 0), isRGBA=True)
1513: (8)                    if gc._forced_alpha:
1514: (12)                       gc._effective_alphas = (gc._alpha, gc._alpha)
1515: (8)                    elif fillcolor is None or len(fillcolor) < 4:
1516: (12)                       gc._effective_alphas = (gc._rgb[3], 1.0)
1517: (8)                    else:
1518: (12)                       gc._effective_alphas = (gc._rgb[3], fillcolor[3])
1519: (8)                    delta = self.gc.delta(gc)
1520: (8)                    if delta:
1521: (12)                       self.file.output(*delta)
1522: (8)                    gc._fillcolor = orig_fill
1523: (8)                    gc._effective_alphas = orig_alphas
1524: (4)                def get_image_magnification(self):
1525: (8)                    return self.image_dpi/72.0
1526: (4)                def draw_image(self, gc, x, y, im, transform=None):
1527: (8)                    h, w = im.shape[:2]
1528: (8)                    if w == 0 or h == 0:
1529: (12)                       return
1530: (8)                    if transform is None:
1531: (12)                       gc.set_alpha(1.0)
1532: (8)                    self.check_gc(gc)
1533: (8)                    w = 72.0 * w / self.image_dpi
1534: (8)                    h = 72.0 * h / self.image_dpi
1535: (8)                    imob = self.file.imageObject(im)
1536: (8)                    if transform is None:
1537: (12)                       self.file.output(Op.gsave,
1538: (29)                                        w, 0, 0, h, x, y, Op.concat_matrix,
1539: (29)                                        imob, Op.use_xobject, Op.grestore)
1540: (8)                    else:
1541: (12)                       tr1, tr2, tr3, tr4, tr5, tr6 = transform.frozen().to_values()
1542: (12)                       self.file.output(Op.gsave,
1543: (29)                                        1, 0, 0, 1, x, y, Op.concat_matrix,
1544: (29)                                        tr1, tr2, tr3, tr4, tr5, tr6, Op.concat_matrix,
1545: (29)                                        imob, Op.use_xobject, Op.grestore)
1546: (4)                def draw_path(self, gc, path, transform, rgbFace=None):
1547: (8)                    self.check_gc(gc, rgbFace)
1548: (8)                    self.file.writePath(
1549: (12)                       path, transform,
1550: (12)                       rgbFace is None and gc.get_hatch_path() is None,
1551: (12)                       gc.get_sketch_params())
1552: (8)                    self.file.output(self.gc.paint())
1553: (4)                def draw_path_collection(self, gc, master_transform, paths,
all_transforms,
1554: (29)                                        offsets, offset_trans, facecolors, edgecolors,
1555: (29)                                        linewidths, linestyles, antialiaseds, urls,
1556: (29)                                        offset_position):
1557: (8)                    can_do_optimization = True
1558: (8)                    facecolors = np.asarray(facecolors)
1559: (8)                    edgecolors = np.asarray(edgecolors)
1560: (8)                    if not len(facecolors):
1561: (12)                       filled = False
1562: (12)                       can_do_optimization = not gc.get_hatch()
1563: (8)                    else:
1564: (12)                       if np.all(facecolors[:, 3] == facecolors[0, 3]):
1565: (16)                           filled = facecolors[0, 3] != 0.0
1566: (12)                       else:
1567: (16)                           can_do_optimization = False
1568: (8)                    if not len(edgecolors):
1569: (12)                       stroked = False
1570: (8)                    else:
1571: (12)                       if np.all(np.asarray(linewidths) == 0.0):
1572: (16)                           stroked = False
1573: (12)                       elif np.all(edgecolors[:, 3] == edgecolors[0, 3]):
```

```
(16)                            stroked = edgecolors[0, 3] != 0.0
(12)                        else:
(16)                            can_do_optimization = False
(8)                     len_path = len(paths[0].vertices) if len(paths) > 0 else 0
(8)                     uses_per_path = self._iter_collection_uses_per_path(
(12)                        paths, all_transforms, offsets, facecolors, edgecolors)
(8)                     should_do_optimization = \
(12)                        len_path + uses_per_path + 5 < len_path * uses_per_path
(8)                     if (not can_do_optimization) or (not should_do_optimization):
(12)                        return RendererBase.draw_path_collection(
(16)                            self, gc, master_transform, paths, all_transforms,
(16)                            offsets, offset_trans, facecolors, edgecolors,
(16)                            linewidths, linestyles, antialiaseds, urls,
(16)                            offset_position)
(8)                     padding = np.max(linewidths)
(8)                     path_codes = []
(8)                     for i, (path, transform) in enumerate(self._iter_collection_raw_paths(
(16)                            master_transform, paths, all_transforms)):
(12)                        name = self.file.pathCollectionObject(
(16)                            gc, path, transform, padding, filled, stroked)
(12)                        path_codes.append(name)
(8)                     output = self.file.output
(8)                     output(*self.gc.push())
(8)                     lastx, lasty = 0, 0
(8)                     for xo, yo, path_id, gc0, rgbFace in self._iter_collection(
(16)                            gc, path_codes, offsets, offset_trans,
(16)                            facecolors, edgecolors, linewidths, linestyles,
(16)                            antialiaseds, urls, offset_position):
(12)                        self.check_gc(gc0, rgbFace)
(12)                        dx, dy = xo - lastx, yo - lasty
(12)                        output(1, 0, 0, 1, dx, dy, Op.concat_matrix, path_id,
(19)                               Op.use_xobject)
(12)                        lastx, lasty = xo, yo
(8)                     output(*self.gc.pop())
(4)                 def draw_markers(self, gc, marker_path, marker_trans, path, trans,
(21)                                 rgbFace=None):
(8)                     len_marker_path = len(marker_path)
(8)                     uses = len(path)
(8)                     if len_marker_path * uses < len_marker_path + uses + 5:
(12)                        RendererBase.draw_markers(self, gc, marker_path, marker_trans,
(38)                                                  path, trans, rgbFace)
(12)                        return
(8)                     self.check_gc(gc, rgbFace)
(8)                     fill = gc.fill(rgbFace)
(8)                     stroke = gc.stroke()
(8)                     output = self.file.output
(8)                     marker = self.file.markerObject(
(12)                        marker_path, marker_trans, fill, stroke, self.gc._linewidth,
(12)                        gc.get_joinstyle(), gc.get_capstyle())
(8)                     output(Op.gsave)
(8)                     lastx, lasty = 0, 0
(8)                     for vertices, code in path.iter_segments(
(16)                            trans,
(16)                            clip=(0, 0, self.file.width*72, self.file.height*72),
(16)                            simplify=False):
(12)                        if len(vertices):
(16)                            x, y = vertices[-2:]
(16)                            if not (0 <= x <= self.file.width * 72
(24)                                    and 0 <= y <= self.file.height * 72):
(20)                                continue
(16)                            dx, dy = x - lastx, y - lasty
(16)                            output(1, 0, 0, 1, dx, dy, Op.concat_matrix,
(23)                                   marker, Op.use_xobject)
(16)                            lastx, lasty = x, y
(8)                     output(Op.grestore)
(4)                 def draw_gouraud_triangles(self, gc, points, colors, trans):
(8)                     assert len(points) == len(colors)
(8)                     if len(points) == 0:
(12)                        return
```

```
(8)                 assert points.ndim == 3
(8)                 assert points.shape[1] == 3
(8)                 assert points.shape[2] == 2
(8)                 assert colors.ndim == 3
(8)                 assert colors.shape[1] == 3
(8)                 assert colors.shape[2] in (1, 4)
(8)                 shape = points.shape
(8)                 points = points.reshape((shape[0] * shape[1], 2))
(8)                 tpoints = trans.transform(points)
(8)                 tpoints = tpoints.reshape(shape)
(8)                 name, _ = self.file.addGouraudTriangles(tpoints, colors)
(8)                 output = self.file.output
(8)                 if colors.shape[2] == 1:
(12)                    gc.set_alpha(1.0)
(12)                    self.check_gc(gc)
(12)                    output(name, Op.shading)
(12)                    return
(8)                 alpha = colors[0, 0, 3]
(8)                 if np.allclose(alpha, colors[:, :, 3]):
(12)                    gc.set_alpha(alpha)
(12)                    self.check_gc(gc)
(12)                    output(name, Op.shading)
(8)                 else:
(12)                    alpha = colors[:, :, 3][:, :, None]
(12)                    _, smask_ob = self.file.addGouraudTriangles(tpoints, alpha)
(12)                    gstate = self.file._soft_mask_state(smask_ob)
(12)                    output(Op.gsave, gstate, Op.setgstate,
(19)                           name, Op.shading,
(19)                           Op.grestore)
(4)             def _setup_textpos(self, x, y, angle, oldx=0, oldy=0, oldangle=0):
(8)                 if angle == oldangle == 0:
(12)                    self.file.output(x - oldx, y - oldy, Op.textpos)
(8)                 else:
(12)                    angle = math.radians(angle)
(12)                    self.file.output(math.cos(angle), math.sin(angle),
(29)                                     -math.sin(angle), math.cos(angle),
(29)                                     x, y, Op.textmatrix)
(12)                    self.file.output(0, 0, Op.textpos)
(4)             def draw_mathtext(self, gc, x, y, s, prop, angle):
(8)                 width, height, descent, glyphs, rects = \
(12)                    self._text2path.mathtext_parser.parse(s, 72, prop)
(8)                 if gc.get_url() is not None:
(12)                    self.file._annotations[-1][1].append(_get_link_annotation(
(16)                        gc, x, y, width, height, angle))
(8)                 fonttype = mpl.rcParams['pdf.fonttype']
(8)                 a = math.radians(angle)
(8)                 self.file.output(Op.gsave)
(8)                 self.file.output(math.cos(a), math.sin(a),
(25)                                 -math.sin(a), math.cos(a),
(25)                                 x, y, Op.concat_matrix)
(8)                 self.check_gc(gc, gc._rgb)
(8)                 prev_font = None, None
(8)                 oldx, oldy = 0, 0
(8)                 unsupported_chars = []
(8)                 self.file.output(Op.begin_text)
(8)                 for font, fontsize, num, ox, oy in glyphs:
(12)                    self.file._character_tracker.track_glyph(font, num)
(12)                    fontname = font.fname
(12)                    if not _font_supports_glyph(fonttype, num):
(16)                        unsupported_chars.append((font, fontsize, ox, oy, num))
(12)                    else:
(16)                        self._setup_textpos(ox, oy, 0, oldx, oldy)
(16)                        oldx, oldy = ox, oy
(16)                        if (fontname, fontsize) != prev_font:
(20)                            self.file.output(self.file.fontName(fontname), fontsize,
(37)                                             Op.selectfont)
(20)                            prev_font = fontname, fontsize
(16)                        self.file.output(self.encode_string(chr(num), fonttype),
(33)                                         Op.show)
```

```
(8)                 self.file.output(Op.end_text)
(8)                 for font, fontsize, ox, oy, num in unsupported_chars:
(12)                    self._draw_xobject_glyph(
(16)                        font, fontsize, font.get_char_index(num), ox, oy)
(8)                 for ox, oy, width, height in rects:
(12)                    self.file.output(Op.gsave, ox, oy, width, height,
(29)                                     Op.rectangle, Op.fill, Op.grestore)
(8)                 self.file.output(Op.grestore)
(4)             def draw_tex(self, gc, x, y, s, prop, angle, *, mtext=None):
(8)                 texmanager = self.get_texmanager()
(8)                 fontsize = prop.get_size_in_points()
(8)                 dvifile = texmanager.make_dvi(s, fontsize)
(8)                 with dviread.Dvi(dvifile, 72) as dvi:
(12)                    page, = dvi
(8)                 if gc.get_url() is not None:
(12)                    self.file._annotations[-1][1].append(_get_link_annotation(
(16)                        gc, x, y, page.width, page.height, angle))
(8)                 oldfont, seq = None, []
(8)                 for x1, y1, dvifont, glyph, width in page.text:
(12)                    if dvifont != oldfont:
(16)                        pdfname = self.file.dviFontName(dvifont)
(16)                        seq += [['font', pdfname, dvifont.size]]
(16)                        oldfont = dvifont
(12)                    seq += [['text', x1, y1, [bytes([glyph])], x1+width]]
(8)                 i, curx, fontsize = 0, 0, None
(8)                 while i < len(seq)-1:
(12)                    elt, nxt = seq[i:i+2]
(12)                    if elt[0] == 'font':
(16)                        fontsize = elt[2]
(12)                    elif elt[0] == nxt[0] == 'text' and elt[2] == nxt[2]:
(16)                        offset = elt[4] - nxt[1]
(16)                        if abs(offset) < 0.1:
(20)                            elt[3][-1] += nxt[3][0]
(20)                            elt[4] += nxt[4]-nxt[1]
(16)                        else:
(20)                            elt[3] += [offset*1000.0/fontsize, nxt[3][0]]
(20)                            elt[4] = nxt[4]
(16)                        del seq[i+1]
(16)                        continue
(12)                    i += 1
(8)                 mytrans = Affine2D().rotate_deg(angle).translate(x, y)
(8)                 self.check_gc(gc, gc._rgb)
(8)                 self.file.output(Op.begin_text)
(8)                 curx, cury, oldx, oldy = 0, 0, 0, 0
(8)                 for elt in seq:
(12)                    if elt[0] == 'font':
(16)                        self.file.output(elt[1], elt[2], Op.selectfont)
(12)                    elif elt[0] == 'text':
(16)                        curx, cury = mytrans.transform((elt[1], elt[2]))
(16)                        self._setup_textpos(curx, cury, angle, oldx, oldy)
(16)                        oldx, oldy = curx, cury
(16)                        if len(elt[3]) == 1:
(20)                            self.file.output(elt[3][0], Op.show)
(16)                        else:
(20)                            self.file.output(elt[3], Op.showkern)
(12)                    else:
(16)                        assert False
(8)                 self.file.output(Op.end_text)
(8)                 boxgc = self.new_gc()
(8)                 boxgc.copy_properties(gc)
(8)                 boxgc.set_linewidth(0)
(8)                 pathops = [Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO,
(19)                           Path.CLOSEPOLY]
(8)                 for x1, y1, h, w in page.boxes:
(12)                    path = Path([[x1, y1], [x1+w, y1], [x1+w, y1+h], [x1, y1+h],
(25)                                 [0, 0]], pathops)
(12)                    self.draw_path(boxgc, path, mytrans, gc._rgb)
(4)             def encode_string(self, s, fonttype):
(8)                 if fonttype in (1, 3):
```

```
1781: (12)                        return s.encode('cp1252', 'replace')
1782: (8)                     return s.encode('utf-16be', 'replace')
1783: (4)                 def draw_text(self, gc, x, y, s, prop, angle, ismath=False, mtext=None):
1784: (8)                     self.check_gc(gc, gc._rgb)
1785: (8)                     if ismath:
1786: (12)                        return self.draw_mathtext(gc, x, y, s, prop, angle)
1787: (8)                     fontsize = prop.get_size_in_points()
1788: (8)                     if mpl.rcParams['pdf.use14corefonts']:
1789: (12)                        font = self._get_font_afm(prop)
1790: (12)                        fonttype = 1
1791: (8)                     else:
1792: (12)                        font = self._get_font_ttf(prop)
1793: (12)                        self.file._character_tracker.track(font, s)
1794: (12)                        fonttype = mpl.rcParams['pdf.fonttype']
1795: (8)                     if gc.get_url() is not None:
1796: (12)                        font.set_text(s)
1797: (12)                        width, height = font.get_width_height()
1798: (12)                        self.file._annotations[-1][1].append(_get_link_annotation(
1799: (16)                            gc, x, y, width / 64, height / 64, angle))
1800: (8)                     if fonttype not in [3, 42]:
1801: (12)                        self.file.output(Op.begin_text,
1802: (29)                                         self.file.fontName(prop), fontsize,
Op.selectfont)
1803: (12)                        self._setup_textpos(x, y, angle)
1804: (12)                        self.file.output(self.encode_string(s, fonttype),
1805: (29)                                         Op.show, Op.end_text)
1806: (8)                     else:
1807: (12)                        singlebyte_chunks = []
1808: (12)                        multibyte_glyphs = []
1809: (12)                        prev_was_multibyte = True
1810: (12)                        prev_font = font
1811: (12)                        for item in _text_helpers.layout(s, font,
kern_mode=KERNING_UNFITTED):
1812: (16)                            if _font_supports_glyph(fonttype, ord(item.char)):
1813: (20)                                if prev_was_multibyte or item.ft_object != prev_font:
1814: (24)                                    singlebyte_chunks.append((item.ft_object, item.x, []))
1815: (24)                                    prev_font = item.ft_object
1816: (20)                                if item.prev_kern:
1817: (24)                                    singlebyte_chunks[-1][2].append(item.prev_kern)
1818: (20)                                singlebyte_chunks[-1][2].append(item.char)
1819: (20)                                prev_was_multibyte = False
1820: (16)                            else:
1821: (20)                                multibyte_glyphs.append((item.ft_object, item.x,
item.glyph_idx))
1822: (20)                                prev_was_multibyte = True
1823: (12)                        self.file.output(Op.gsave)
1824: (12)                        a = math.radians(angle)
1825: (12)                        self.file.output(math.cos(a), math.sin(a),
1826: (29)                                         -math.sin(a), math.cos(a),
1827: (29)                                         x, y, Op.concat_matrix)
1828: (12)                        self.file.output(Op.begin_text)
1829: (12)                        prev_start_x = 0
1830: (12)                        for ft_object, start_x, kerns_or_chars in singlebyte_chunks:
1831: (16)                            ft_name = self.file.fontName(ft_object.fname)
1832: (16)                            self.file.output(ft_name, fontsize, Op.selectfont)
1833: (16)                            self._setup_textpos(start_x, 0, 0, prev_start_x, 0, 0)
1834: (16)                            self.file.output(
1835: (20)                                [-1000 * next(group) / fontsize if tp == float  # a kern
1836: (21)                                 else self.encode_string("".join(group), fonttype)
1837: (21)                                 for tp, group in itertools.groupby(kerns_or_chars,
type)],
1838: (20)                                Op.showkern)
1839: (16)                            prev_start_x = start_x
1840: (12)                        self.file.output(Op.end_text)
1841: (12)                        for ft_object, start_x, glyph_idx in multibyte_glyphs:
1842: (16)                            self._draw_xobject_glyph(
1843: (20)                                ft_object, fontsize, glyph_idx, start_x, 0
1844: (16)                            )
1845: (12)                        self.file.output(Op.grestore)
```

```
(4)     def _draw_xobject_glyph(self, font, fontsize, glyph_idx, x, y):
(8)         """Draw a multibyte character from a Type 3 font as an XObject."""
(8)         glyph_name = font.get_glyph_name(glyph_idx)
(8)         name = self.file._get_xobject_glyph_name(font.fname, glyph_name)
(8)         self.file.output(
(12)            Op.gsave,
(12)            0.001 * fontsize, 0, 0, 0.001 * fontsize, x, y, Op.concat_matrix,
(12)            Name(name), Op.use_xobject,
(12)            Op.grestore,
(8)         )
(4)     def new_gc(self):
(8)         return GraphicsContextPdf(self.file)
(0)class GraphicsContextPdf(GraphicsContextBase):
(4)     def __init__(self, file):
(8)         super().__init__()
(8)         self._fillcolor = (0.0, 0.0, 0.0)
(8)         self._effective_alphas = (1.0, 1.0)
(8)         self.file = file
(8)         self.parent = None
(4)     def __repr__(self):
(8)         d = dict(self.__dict__)
(8)         del d['file']
(8)         del d['parent']
(8)         return repr(d)
(4)     def stroke(self):
(8)         """
(8)         Predicate: does the path need to be stroked (its outline drawn)?
(8)         This tests for the various conditions that disable stroking
(8)         the path, in which case it would presumably be filled.
(8)         """
(8)         return (self._linewidth > 0 and self._alpha > 0 and
(16)                (len(self._rgb) <= 3 or self._rgb[3] != 0.0))
(4)     def fill(self, *args):
(8)         """
(8)         Predicate: does the path need to be filled?
(8)         An optional argument can be used to specify an alternative
(8)         _fillcolor, as needed by RendererPdf.draw_markers.
(8)         """
(8)         if len(args):
(12)            _fillcolor = args[0]
(8)         else:
(12)            _fillcolor = self._fillcolor
(8)         return (self._hatch or
(16)                (_fillcolor is not None and
(17)                 (len(_fillcolor) <= 3 or _fillcolor[3] != 0.0)))
(4)     def paint(self):
(8)         """
(8)         Return the appropriate pdf operator to cause the path to be
(8)         stroked, filled, or both.
(8)         """
(8)         return Op.paint_path(self.fill(), self.stroke())
(4)     capstyles = {'butt': 0, 'round': 1, 'projecting': 2}
(4)     joinstyles = {'miter': 0, 'round': 1, 'bevel': 2}
(4)     def capstyle_cmd(self, style):
(8)         return [self.capstyles[style], Op.setlinecap]
(4)     def joinstyle_cmd(self, style):
(8)         return [self.joinstyles[style], Op.setlinejoin]
(4)     def linewidth_cmd(self, width):
(8)         return [width, Op.setlinewidth]
(4)     def dash_cmd(self, dashes):
(8)         offset, dash = dashes
(8)         if dash is None:
(12)            dash = []
(12)            offset = 0
(8)         return [list(dash), offset, Op.setdash]
(4)     def alpha_cmd(self, alpha, forced, effective_alphas):
(8)         name = self.file.alphaState(effective_alphas)
(8)         return [name, Op.setgstate]
(4)     def hatch_cmd(self, hatch, hatch_color):
```

```
(8)                 if not hatch:
(12)                    if self._fillcolor is not None:
(16)                        return self.fillcolor_cmd(self._fillcolor)
(12)                    else:
(16)                        return [Name('DeviceRGB'), Op.setcolorspace_nonstroke]
(8)                 else:
(12)                    hatch_style = (hatch_color, self._fillcolor, hatch)
(12)                    name = self.file.hatchPattern(hatch_style)
(12)                    return [Name('Pattern'), Op.setcolorspace_nonstroke,
(20)                            name, Op.setcolor_nonstroke]
(4)             def rgb_cmd(self, rgb):
(8)                 if mpl.rcParams['pdf.inheritcolor']:
(12)                    return []
(8)                 if rgb[0] == rgb[1] == rgb[2]:
(12)                    return [rgb[0], Op.setgray_stroke]
(8)                 else:
(12)                    return [*rgb[:3], Op.setrgb_stroke]
(4)             def fillcolor_cmd(self, rgb):
(8)                 if rgb is None or mpl.rcParams['pdf.inheritcolor']:
(12)                    return []
(8)                 elif rgb[0] == rgb[1] == rgb[2]:
(12)                    return [rgb[0], Op.setgray_nonstroke]
(8)                 else:
(12)                    return [*rgb[:3], Op.setrgb_nonstroke]
(4)             def push(self):
(8)                 parent = GraphicsContextPdf(self.file)
(8)                 parent.copy_properties(self)
(8)                 parent.parent = self.parent
(8)                 self.parent = parent
(8)                 return [Op.gsave]
(4)             def pop(self):
(8)                 assert self.parent is not None
(8)                 self.copy_properties(self.parent)
(8)                 self.parent = self.parent.parent
(8)                 return [Op.grestore]
(4)             def clip_cmd(self, cliprect, clippath):
(8)                 """Set clip rectangle. Calls `.pop()` and `.push()`."""
(8)                 cmds = []
(8)                 while ((self._cliprect, self._clippath) != (cliprect, clippath)
(16)                        and self.parent is not None):
(12)                    cmds.extend(self.pop())
(8)                 if ((self._cliprect, self._clippath) != (cliprect, clippath) or
(16)                        self.parent is None):
(12)                    cmds.extend(self.push())
(12)                    if self._cliprect != cliprect:
(16)                        cmds.extend([cliprect, Op.rectangle, Op.clip, Op.endpath])
(12)                    if self._clippath != clippath:
(16)                        path, affine = clippath.get_transformed_path_and_affine()
(16)                        cmds.extend(
(20)                            PdfFile.pathOperations(path, affine, simplify=False) +
(20)                            [Op.clip, Op.endpath])
(8)                 return cmds
(4)             commands = (
(8)                 (('_cliprect', '_clippath'), clip_cmd),
(8)                 (('_alpha', '_forced_alpha', '_effective_alphas'), alpha_cmd),
(8)                 (('_capstyle',), capstyle_cmd),
(8)                 (('_fillcolor',), fillcolor_cmd),
(8)                 (('_joinstyle',), joinstyle_cmd),
(8)                 (('_linewidth',), linewidth_cmd),
(8)                 (('_dashes',), dash_cmd),
(8)                 (('_rgb',), rgb_cmd),
(8)                 (('_hatch', '_hatch_color'), hatch_cmd),
(8)                 )
(4)             def delta(self, other):
(8)                 """
(8)                 Copy properties of other into self and return PDF commands
(8)                 needed to transform *self* into *other*.
(8)                 """
(8)                 cmds = []
```

```
1984: (8)                   fill_performed = False
1985: (8)                   for params, cmd in self.commands:
1986: (12)                      different = False
1987: (12)                      for p in params:
1988: (16)                          ours = getattr(self, p)
1989: (16)                          theirs = getattr(other, p)
1990: (16)                          try:
1991: (20)                              if ours is None or theirs is None:
1992: (24)                                  different = ours is not theirs
1993: (20)                              else:
1994: (24)                                  different = bool(ours != theirs)
1995: (16)                          except ValueError:
1996: (20)                              ours = np.asarray(ours)
1997: (20)                              theirs = np.asarray(theirs)
1998: (20)                              different = (ours.shape != theirs.shape or
1999: (33)                                           np.any(ours != theirs))
2000: (16)                          if different:
2001: (20)                              break
2002: (12)                      if params == ('_hatch', '_hatch_color') and fill_performed:
2003: (16)                          different = True
2004: (12)                      if different:
2005: (16)                          if params == ('_fillcolor',):
2006: (20)                              fill_performed = True
2007: (16)                          theirs = [getattr(other, p) for p in params]
2008: (16)                          cmds.extend(cmd(self, *theirs))
2009: (16)                          for p in params:
2010: (20)                              setattr(self, p, getattr(other, p))
2011: (8)                   return cmds
2012: (4)               def copy_properties(self, other):
2013: (8)                   """
2014: (8)                   Copy properties of other into self.
2015: (8)                   """
2016: (8)                   super().copy_properties(other)
2017: (8)                   fillcolor = getattr(other, '_fillcolor', self._fillcolor)
2018: (8)                   effective_alphas = getattr(other, '_effective_alphas',
2019: (35)                                             self._effective_alphas)
2020: (8)                   self._fillcolor = fillcolor
2021: (8)                   self._effective_alphas = effective_alphas
2022: (4)               def finalize(self):
2023: (8)                   """
2024: (8)                   Make sure every pushed graphics state is popped.
2025: (8)                   """
2026: (8)                   cmds = []
2027: (8)                   while self.parent is not None:
2028: (12)                      cmds.extend(self.pop())
2029: (8)                   return cmds
2030: (0)           class PdfPages:
2031: (4)               """
2032: (4)               A multi-page PDF file.
2033: (4)               Examples
2034: (4)               --------
2035: (4)               >>> import matplotlib.pyplot as plt
2036: (4)               >>> # Initialize:
2037: (4)               >>> with PdfPages('foo.pdf') as pdf:
2038: (4)               ...     # As many times as you like, create a figure fig and save it:
2039: (4)               ...     fig = plt.figure()
2040: (4)               ...     pdf.savefig(fig)
2041: (4)               ...     # When no figure is specified the current figure is saved
2042: (4)               ...     pdf.savefig()
2043: (4)               Notes
2044: (4)               -----
2045: (4)               In reality `PdfPages` is a thin wrapper around `PdfFile`, in order to
avoid
2046: (4)               confusion when using `~.pyplot.savefig` and forgetting the format
argument.
2047: (4)               """
2048: (4)               _UNSET = object()
2049: (4)               def __init__(self, filename, keep_empty=_UNSET, metadata=None):
2050: (8)                   """
```

```
(8)                 Create a new PdfPages object.
(8)                 Parameters
(8)                 ----------
(8)                 filename : str or path-like or file-like
(12)                    Plots using `PdfPages.savefig` will be written to a file at this location.
(12)                    The file is opened when a figure is saved for the first time (overwriting
(12)                    any older file with the same name).
(8)                 keep_empty : bool, optional
(12)                    If set to False, then empty pdf files will be deleted automatically
(12)                    when closed.
(8)                 metadata : dict, optional
(12)                    Information dictionary object (see PDF reference section 10.2.1
(12)                    'Document Information Dictionary'), e.g.:
(12)                    ``{'Creator': 'My software', 'Author': 'Me', 'Title': 'Awesome'}``.
(12)                    The standard keys are 'Title', 'Author', 'Subject', 'Keywords',
(12)                    'Creator', 'Producer', 'CreationDate', 'ModDate', and
(12)                    'Trapped'. Values have been predefined for 'Creator', 'Producer'
(12)                    and 'CreationDate'. They can be removed by setting them to `None`.
(8)                 """
(8)                 self._filename = filename
(8)                 self._metadata = metadata
(8)                 self._file = None
(8)                 if keep_empty and keep_empty is not self._UNSET:
(12)                    _api.warn_deprecated("3.8", message=(
(16)                        "Keeping empty pdf files is deprecated since %(since)s and support "
(16)                        "will be removed %(removal)s."))
(8)                 self._keep_empty = keep_empty
(4)             keep_empty = _api.deprecate_privatize_attribute("3.8")
(4)             def __enter__(self):
(8)                 return self
(4)             def __exit__(self, exc_type, exc_val, exc_tb):
(8)                 self.close()
(4)             def _ensure_file(self):
(8)                 if self._file is None:
(12)                    self._file = PdfFile(self._filename, metadata=self._metadata)  # init.
(8)                 return self._file
(4)             def close(self):
(8)                 """
(8)                 Finalize this object, making the underlying file a complete
(8)                 PDF file.
(8)                 """
(8)                 if self._file is not None:
(12)                    self._file.finalize()
(12)                    self._file.close()
(12)                    self._file = None
(8)                 elif self._keep_empty:  # True *or* UNSET.
(12)                    _api.warn_deprecated("3.8", message=(
(16)                        "Keeping empty pdf files is deprecated since %(since)s and support "
(16)                        "will be removed %(removal)s."))
(12)                    PdfFile(self._filename, metadata=self._metadata).close()  # touch the file.
(4)             def infodict(self):
(8)                 """
(8)                 Return a modifiable information dictionary object
(8)                 (see PDF reference section 10.2.1 'Document Information
(8)                 Dictionary').
(8)                 """
(8)                 return self._ensure_file().infoDict
(4)             def savefig(self, figure=None, **kwargs):
(8)                 """
(8)                 Save a `.Figure` to this file as a new page.
(8)                 Any other keyword arguments are passed to `~.Figure.savefig`.
```

```
2112: (8)                   Parameters
2113: (8)                   ----------
2114: (8)                   figure : `.Figure` or int, default: the active figure
2115: (12)                      The figure, or index of the figure, that is saved to the file.
2116: (8)                   """
2117: (8)                   if not isinstance(figure, Figure):
2118: (12)                      if figure is None:
2119: (16)                          manager = Gcf.get_active()
2120: (12)                      else:
2121: (16)                          manager = Gcf.get_fig_manager(figure)
2122: (12)                      if manager is None:
2123: (16)                          raise ValueError(f"No figure {figure}")
2124: (12)                      figure = manager.canvas.figure
2125: (8)                   figure.savefig(self, format="pdf", backend="pdf", **kwargs)
2126: (4)               def get_pagecount(self):
2127: (8)                   """Return the current number of pages in the multipage pdf file."""
2128: (8)                   return len(self._ensure_file().pageList)
2129: (4)               def attach_note(self, text, positionRect=[-100, -100, 0, 0]):
2130: (8)                   """
2131: (8)                   Add a new text note to the page to be saved next. The optional
2132: (8)                   positionRect specifies the position of the new note on the
2133: (8)                   page. It is outside the page per default to make sure it is
2134: (8)                   invisible on printouts.
2135: (8)                   """
2136: (8)                   self._ensure_file().newTextnote(text, positionRect)
2137: (0)           class FigureCanvasPdf(FigureCanvasBase):
2138: (4)               fixed_dpi = 72
2139: (4)               filetypes = {'pdf': 'Portable Document Format'}
2140: (4)               def get_default_filetype(self):
2141: (8)                   return 'pdf'
2142: (4)               def print_pdf(self, filename, *,
2143: (18)                            bbox_inches_restore=None, metadata=None):
2144: (8)                   dpi = self.figure.dpi
2145: (8)                   self.figure.dpi = 72  # there are 72 pdf points to an inch
2146: (8)                   width, height = self.figure.get_size_inches()
2147: (8)                   if isinstance(filename, PdfPages):
2148: (12)                      file = filename._ensure_file()
2149: (8)                   else:
2150: (12)                      file = PdfFile(filename, metadata=metadata)
2151: (8)                   try:
2152: (12)                      file.newPage(width, height)
2153: (12)                      renderer = MixedModeRenderer(
2154: (16)                          self.figure, width, height, dpi,
2155: (16)                          RendererPdf(file, dpi, height, width),
2156: (16)                          bbox_inches_restore=bbox_inches_restore)
2157: (12)                      self.figure.draw(renderer)
2158: (12)                      renderer.finalize()
2159: (12)                      if not isinstance(filename, PdfPages):
2160: (16)                          file.finalize()
2161: (8)                   finally:
2162: (12)                      if isinstance(filename, PdfPages):  # finish off this page
2163: (16)                          file.endStream()
2164: (12)                      else:            # we opened the file above; now finish it off
2165: (16)                          file.close()
2166: (4)               def draw(self):
2167: (8)                   self.figure.draw_without_rendering()
2168: (8)                   return super().draw()
2169: (0)           FigureManagerPdf = FigureManagerBase
2170: (0)           @_Backend.export
2171: (0)           class _BackendPdf(_Backend):
2172: (4)               FigureCanvas = FigureCanvasPdf
```

File 93 - backend_pgf.py:

```
1: (0)              import codecs
2: (0)              import datetime
3: (0)              import functools
```

```
4: (0)              from io import BytesIO
5: (0)              import logging
6: (0)              import math
7: (0)              import os
8: (0)              import pathlib
9: (0)              import shutil
10: (0)             import subprocess
11: (0)             from tempfile import TemporaryDirectory
12: (0)             import weakref
13: (0)             from PIL import Image
14: (0)             import matplotlib as mpl
15: (0)             from matplotlib import _api, cbook, font_manager as fm
16: (0)             from matplotlib.backend_bases import (
17: (4)                 _Backend, FigureCanvasBase, FigureManagerBase, RendererBase
18: (0)             )
19: (0)             from matplotlib.backends.backend_mixed import MixedModeRenderer
20: (0)             from matplotlib.backends.backend_pdf import (
21: (4)                 _create_pdf_info_dict, _datetime_to_pdf)
22: (0)             from matplotlib.path import Path
23: (0)             from matplotlib.figure import Figure
24: (0)             from matplotlib._pylab_helpers import Gcf
25: (0)             _log = logging.getLogger(__name__)
26: (0)             def _get_preamble():
27: (4)                 """Prepare a LaTeX preamble based on the rcParams configuration."""
28: (4)                 return "\n".join([
29: (8)                     r"\def\mathdefault#1{#1}",
30: (8)                     r"\everymath=\expandafter{\the\everymath\displaystyle}",
31: (8)                     mpl.rcParams["pgf.preamble"],
32: (8)                     r"\ifdefined\pdftexversion\else  % non-pdftex case.",
33: (8)                     r"  \usepackage{fontspec}",
34: (8)                     *([
35: (12)                        r"  \%s{%s}[Path=\detokenize{%s/}]"
36: (12)                        % (command, path.name, path.parent.as_posix())
37: (12)                        for command, path in zip(
38: (16)                            ["setmainfont", "setsansfont", "setmonofont"],
39: (16)                            [pathlib.Path(fm.findfont(family))
40: (17)                             for family in ["serif", "sans\\-serif", "monospace"]]
41: (12)                        )
42: (8)                     ] if mpl.rcParams["pgf.rcfonts"] else []),
43: (8)                     r"\fi",
44: (8)                     mpl.texmanager._usepackage_if_not_loaded("underscore",
option="strings"),
45: (4)                 ])
46: (0)             latex_pt_to_in = 1. / 72.27
47: (0)             latex_in_to_pt = 1. / latex_pt_to_in
48: (0)             mpl_pt_to_in = 1. / 72.
49: (0)             mpl_in_to_pt = 1. / mpl_pt_to_in
50: (0)             def _tex_escape(text):
51: (4)                 r"""
52: (4)                 Do some necessary and/or useful substitutions for texts to be included in
53: (4)                 LaTeX documents.
54: (4)                 """
55: (4)                 return text.replace("\N{MINUS SIGN}", r"\ensuremath{-}")
56: (0)             def _writeln(fh, line):
57: (4)                 fh.write(line)
58: (4)                 fh.write("%\n")
59: (0)             def _escape_and_apply_props(s, prop):
60: (4)                 """
61: (4)                 Generate a TeX string that renders string *s* with font properties *prop*,
62: (4)                 also applying any required escapes to *s*.
63: (4)                 """
64: (4)                 commands = []
65: (4)                 families = {"serif": r"\rmfamily", "sans": r"\sffamily",
66: (16)                            "sans-serif": r"\sffamily", "monospace": r"\ttfamily"}
67: (4)                 family = prop.get_family()[0]
68: (4)                 if family in families:
69: (8)                     commands.append(families[family])
70: (4)                 elif any(font.name == family for font in fm.fontManager.ttflist):
71: (8)                     commands.append(
```

```
72: (12)                        r"\ifdefined\pdftexversion\else\setmainfont{%s}\rmfamily\fi" %
family)
73: (4)                 else:
74: (8)                     _log.warning("Ignoring unknown font: %s", family)
75: (4)                 size = prop.get_size_in_points()
76: (4)                 commands.append(r"\fontsize{%f}{%f}" % (size, size * 1.2))
77: (4)                 styles = {"normal": r"", "italic": r"\itshape", "oblique": r"\slshape"}
78: (4)                 commands.append(styles[prop.get_style()])
79: (4)                 boldstyles = ["semibold", "demibold", "demi", "bold", "heavy",
80: (18)                              "extra bold", "black"]
81: (4)                 if prop.get_weight() in boldstyles:
82: (8)                     commands.append(r"\bfseries")
83: (4)                 commands.append(r"\selectfont")
84: (4)                 return (
85: (8)                     "{"
86: (8)                     + "".join(commands)
87: (8)                     + r"\catcode`\^=\active\def^{\ifmmode\sp\else\^{}\fi}"
88: (8)                     + r"\catcode`\%=\active\def%{\%}"
89: (8)                     + _tex_escape(s)
90: (8)                     + "}"
91: (4)                 )
92: (0)             def _metadata_to_str(key, value):
93: (4)                 """Convert metadata key/value to a form that hyperref accepts."""
94: (4)                 if isinstance(value, datetime.datetime):
95: (8)                     value = _datetime_to_pdf(value)
96: (4)                 elif key == 'Trapped':
97: (8)                     value = value.name.decode('ascii')
98: (4)                 else:
99: (8)                     value = str(value)
100: (4)                return f'{key}={{{value}}}'
101: (0)            def make_pdf_to_png_converter():
102: (4)                """Return a function that converts a pdf file to a png file."""
103: (4)                try:
104: (8)                    mpl._get_executable_info("pdftocairo")
105: (4)                except mpl.ExecutableNotFoundError:
106: (8)                    pass
107: (4)                else:
108: (8)                    return lambda pdffile, pngfile, dpi: subprocess.check_output(
109: (12)                       ["pdftocairo", "-singlefile", "-transp", "-png", "-r", "%d" % dpi,
110: (13)                        pdffile, os.path.splitext(pngfile)[0]],
111: (12)                       stderr=subprocess.STDOUT)
112: (4)                try:
113: (8)                    gs_info = mpl._get_executable_info("gs")
114: (4)                except mpl.ExecutableNotFoundError:
115: (8)                    pass
116: (4)                else:
117: (8)                    return lambda pdffile, pngfile, dpi: subprocess.check_output(
118: (12)                       [gs_info.executable,
119: (13)                        '-dQUIET', '-dSAFER', '-dBATCH', '-dNOPAUSE', '-dNOPROMPT',
120: (13)                        '-dUseCIEColor', '-dTextAlphaBits=4',
121: (13)                        '-dGraphicsAlphaBits=4', '-dDOINTERPOLATE',
122: (13)                        '-sDEVICE=pngalpha', '-sOutputFile=%s' % pngfile,
123: (13)                        '-r%d' % dpi, pdffile],
124: (12)                       stderr=subprocess.STDOUT)
125: (4)                raise RuntimeError("No suitable pdf to png renderer found.")
126: (0)            class LatexError(Exception):
127: (4)                def __init__(self, message, latex_output=""):
128: (8)                    super().__init__(message)
129: (8)                    self.latex_output = latex_output
130: (4)                def __str__(self):
131: (8)                    s, = self.args
132: (8)                    if self.latex_output:
133: (12)                       s += "\n" + self.latex_output
134: (8)                    return s
135: (0)            class LatexManager:
136: (4)                """
137: (4)                The LatexManager opens an instance of the LaTeX application for
138: (4)                determining the metrics of text elements. The LaTeX environment can be
139: (4)                modified by setting fonts and/or a custom preamble in `.rcParams`.
```

```
140: (4)                """
141: (4)                @staticmethod
142: (4)                def _build_latex_header():
143: (8)                    latex_header = [
144: (12)                       r"\documentclass{article}",
145: (12)                       rf"% !TeX program = {mpl.rcParams['pgf.texsystem']}",
146: (12)                       r"\usepackage{graphicx}",
147: (12)                       _get_preamble(),
148: (12)                       r"\begin{document}",
149: (12)                       r"\typeout{pgf_backend_query_start}",
150: (8)                    ]
151: (8)                    return "\n".join(latex_header)
152: (4)                @classmethod
153: (4)                def _get_cached_or_new(cls):
154: (8)                    """
155: (8)                    Return the previous LatexManager if the header and tex system did not
156: (8)                    change, or a new instance otherwise.
157: (8)                    """
158: (8)                    return cls._get_cached_or_new_impl(cls._build_latex_header())
159: (4)                @classmethod
160: (4)                @functools.lru_cache(1)
161: (4)                def _get_cached_or_new_impl(cls, header):  # Helper for
_get_cached_or_new.
162: (8)                    return cls()
163: (4)                def _stdin_writeln(self, s):
164: (8)                    if self.latex is None:
165: (12)                       self._setup_latex_process()
166: (8)                    self.latex.stdin.write(s)
167: (8)                    self.latex.stdin.write("\n")
168: (8)                    self.latex.stdin.flush()
169: (4)                def _expect(self, s):
170: (8)                    s = list(s)
171: (8)                    chars = []
172: (8)                    while True:
173: (12)                       c = self.latex.stdout.read(1)
174: (12)                       chars.append(c)
175: (12)                       if chars[-len(s):] == s:
176: (16)                           break
177: (12)                       if not c:
178: (16)                           self.latex.kill()
179: (16)                           self.latex = None
180: (16)                           raise LatexError("LaTeX process halted", "".join(chars))
181: (8)                    return "".join(chars)
182: (4)                def _expect_prompt(self):
183: (8)                    return self._expect("\n*")
184: (4)                def __init__(self):
185: (8)                    self._tmpdir = TemporaryDirectory()
186: (8)                    self.tmpdir = self._tmpdir.name
187: (8)                    self._finalize_tmpdir = weakref.finalize(self, self._tmpdir.cleanup)
188: (8)                    self._setup_latex_process(expect_reply=False)
189: (8)                    stdout, stderr = self.latex.communicate("\n\\makeatletter\\@@end\n")
190: (8)                    if self.latex.returncode != 0:
191: (12)                       raise LatexError(
192: (16)                           f"LaTeX errored (probably missing font or error in preamble) "
193: (16)                           f"while processing the following input:\n"
194: (16)                           f"{self._build_latex_header()}",
195: (16)                           stdout)
196: (8)                    self.latex = None  # Will be set up on first use.
197: (8)                    self._get_box_metrics = functools.lru_cache(self._get_box_metrics)
198: (4)                def _setup_latex_process(self, *, expect_reply=True):
199: (8)                    try:
200: (12)                       self.latex = subprocess.Popen(
201: (16)                           [mpl.rcParams["pgf.texsystem"], "-halt-on-error"],
202: (16)                           stdin=subprocess.PIPE, stdout=subprocess.PIPE,
203: (16)                           encoding="utf-8", cwd=self.tmpdir)
204: (8)                    except FileNotFoundError as err:
205: (12)                       raise RuntimeError(
206: (16)                           f"{mpl.rcParams['pgf.texsystem']!r} not found; install it or
change "
```

```
207: (16)                            f"rcParams['pgf.texsystem'] to an available TeX
implementation"
208: (12)                        ) from err
209: (8)                    except OSError as err:
210: (12)                        raise RuntimeError(
211: (16)                            f"Error starting {mpl.rcParams['pgf.texsystem']!r}") from err
212: (8)                    def finalize_latex(latex):
213: (12)                        latex.kill()
214: (12)                        try:
215: (16)                            latex.communicate()
216: (12)                        except RuntimeError:
217: (16)                            latex.wait()
218: (8)                    self._finalize_latex = weakref.finalize(
219: (12)                        self, finalize_latex, self.latex)
220: (8)                    self._stdin_writeln(self._build_latex_header())
221: (8)                    if expect_reply:  # read until 'pgf_backend_query_start' token appears
222: (12)                        self._expect("*pgf_backend_query_start")
223: (12)                        self._expect_prompt()
224: (4)                def get_width_height_descent(self, text, prop):
225: (8)                    """
226: (8)                    Get the width, total height, and descent (in TeX points) for a text
227: (8)                    typeset by the current LaTeX environment.
228: (8)                    """
229: (8)                    return self._get_box_metrics(_escape_and_apply_props(text, prop))
230: (4)                def _get_box_metrics(self, tex):
231: (8)                    """
232: (8)                    Get the width, total height and descent (in TeX points) for a TeX
233: (8)                    command's output in the current LaTeX environment.
234: (8)                    """
235: (8)                    self._stdin_writeln(  # Send textbox to TeX & request metrics typeout.
236: (12)                        r"{\catcode`\^=\active\catcode`\%%=\active\sbox0{%s}"
237: (12)                        r"\typeout{\the\wd0,\the\ht0,\the\dp0}}"
238: (12)                        % tex)
239: (8)                    try:
240: (12)                        answer = self._expect_prompt()
241: (8)                    except LatexError as err:
242: (12)                        raise ValueError("Error measuring {}\nLaTeX Output:\n{}"
243: (29)                                         .format(tex, err.latex_output)) from err
244: (8)                    try:
245: (12)                        width, height, offset = answer.splitlines()[-3].split(",")
246: (8)                    except Exception as err:
247: (12)                        raise ValueError("Error measuring {}\nLaTeX Output:\n{}"
248: (29)                                         .format(tex, answer)) from err
249: (8)                    w, h, o = float(width[:-2]), float(height[:-2]), float(offset[:-2])
250: (8)                    return w, h + o, o
251: (0)            @functools.lru_cache(1)
252: (0)            def _get_image_inclusion_command():
253: (4)                man = LatexManager._get_cached_or_new()
254: (4)                man._stdin_writeln(
255: (8)                    r"\includegraphics[interpolate=true]{%s}"
256: (8)                    % cbook._get_data_path("images/matplotlib.png").as_posix())
257: (4)                try:
258: (8)                    man._expect_prompt()
259: (8)                    return r"\includegraphics"
260: (4)                except LatexError:
261: (8)                    LatexManager._get_cached_or_new_impl.cache_clear()
262: (8)                    return r"\pgfimage"
263: (0)            class RendererPgf(RendererBase):
264: (4)                def __init__(self, figure, fh):
265: (8)                    """
266: (8)                    Create a new PGF renderer that translates any drawing instruction
267: (8)                    into text commands to be interpreted in a latex pgfpicture
environment.
268: (8)                    Attributes
269: (8)                    ----------
270: (8)                    figure : `~matplotlib.figure.Figure`
271: (12)                        Matplotlib figure to initialize height, width and dpi from.
272: (8)                    fh : file-like
273: (12)                        File handle for the output of the drawing commands.
```

```
274: (8)                    """
275: (8)                    super().__init__()
276: (8)                    self.dpi = figure.dpi
277: (8)                    self.fh = fh
278: (8)                    self.figure = figure
279: (8)                    self.image_counter = 0
280: (4)                def draw_markers(self, gc, marker_path, marker_trans, path, trans,
281: (21)                                rgbFace=None):
282: (8)                    _writeln(self.fh, r"\begin{pgfscope}")
283: (8)                    f = 1. / self.dpi
284: (8)                    self._print_pgf_clip(gc)
285: (8)                    self._print_pgf_path_styles(gc, rgbFace)
286: (8)                    bl, tr = marker_path.get_extents(marker_trans).get_points()
287: (8)                    coords = bl[0] * f, bl[1] * f, tr[0] * f, tr[1] * f
288: (8)                    _writeln(self.fh,
289: (17)                             r"\pgfsys@defobject{currentmarker}"
290: (17)                             r"{\pgfqpoint{%fin}{%fin}}{\pgfqpoint{%fin}{%fin}}{" %
coords)
291: (8)                    self._print_pgf_path(None, marker_path, marker_trans)
292: (8)                    self._pgf_path_draw(stroke=gc.get_linewidth() != 0.0,
293: (28)                                       fill=rgbFace is not None)
294: (8)                    _writeln(self.fh, r"}")
295: (8)                    maxcoord = 16383 / 72.27 * self.dpi  # Max dimensions in LaTeX.
296: (8)                    clip = (-maxcoord, -maxcoord, maxcoord, maxcoord)
297: (8)                    for point, code in path.iter_segments(trans, simplify=False,
298: (46)                                                         clip=clip):
299: (12)                       x, y = point[0] * f, point[1] * f
300: (12)                       _writeln(self.fh, r"\begin{pgfscope}")
301: (12)                       _writeln(self.fh, r"\pgfsys@transformshift{%fin}{%fin}" % (x, y))
302: (12)                       _writeln(self.fh, r"\pgfsys@useobject{currentmarker}{}")
303: (12)                       _writeln(self.fh, r"\end{pgfscope}")
304: (8)                    _writeln(self.fh, r"\end{pgfscope}")
305: (4)                def draw_path(self, gc, path, transform, rgbFace=None):
306: (8)                    _writeln(self.fh, r"\begin{pgfscope}")
307: (8)                    self._print_pgf_clip(gc)
308: (8)                    self._print_pgf_path_styles(gc, rgbFace)
309: (8)                    self._print_pgf_path(gc, path, transform, rgbFace)
310: (8)                    self._pgf_path_draw(stroke=gc.get_linewidth() != 0.0,
311: (28)                                       fill=rgbFace is not None)
312: (8)                    _writeln(self.fh, r"\end{pgfscope}")
313: (8)                    if gc.get_hatch():
314: (12)                       _writeln(self.fh, r"\begin{pgfscope}")
315: (12)                       self._print_pgf_path_styles(gc, rgbFace)
316: (12)                       self._print_pgf_clip(gc)
317: (12)                       self._print_pgf_path(gc, path, transform, rgbFace)
318: (12)                       _writeln(self.fh, r"\pgfusepath{clip}")
319: (12)                       _writeln(self.fh,
320: (21)                                r"\pgfsys@defobject{currentpattern}"
321: (21)                                r"{\pgfqpoint{0in}{0in}}{\pgfqpoint{1in}{1in}}{")
322: (12)                       _writeln(self.fh, r"\begin{pgfscope}")
323: (12)                       _writeln(self.fh,
324: (21)                                r"\pgfpathrectangle"
325: (21)                                r"{\pgfqpoint{0in}{0in}}{\pgfqpoint{1in}{1in}}")
326: (12)                       _writeln(self.fh, r"\pgfusepath{clip}")
327: (12)                       scale = mpl.transforms.Affine2D().scale(self.dpi)
328: (12)                       self._print_pgf_path(None, gc.get_hatch_path(), scale)
329: (12)                       self._pgf_path_draw(stroke=True)
330: (12)                       _writeln(self.fh, r"\end{pgfscope}")
331: (12)                       _writeln(self.fh, r"}")
332: (12)                       f = 1. / self.dpi
333: (12)                       (xmin, ymin), (xmax, ymax) = \
334: (16)                           path.get_extents(transform).get_points()
335: (12)                       xmin, xmax = f * xmin, f * xmax
336: (12)                       ymin, ymax = f * ymin, f * ymax
337: (12)                       repx, repy = math.ceil(xmax - xmin), math.ceil(ymax - ymin)
338: (12)                       _writeln(self.fh,
339: (21)                                r"\pgfsys@transformshift{%fin}{%fin}" % (xmin, ymin))
340: (12)                       for iy in range(repy):
341: (16)                           for ix in range(repx):
```

```
342: (20)                                _writeln(self.fh, r"\pgfsys@useobject{currentpattern}{}")
343: (20)                                _writeln(self.fh, r"\pgfsys@transformshift{1in}{0in}")
344: (16)                            _writeln(self.fh, r"\pgfsys@transformshift{-%din}{0in}" %
repx)
345: (16)                            _writeln(self.fh, r"\pgfsys@transformshift{0in}{1in}")
346: (12)                        _writeln(self.fh, r"\end{pgfscope}")
347: (4)                 def _print_pgf_clip(self, gc):
348: (8)                     f = 1. / self.dpi
349: (8)                     bbox = gc.get_clip_rectangle()
350: (8)                     if bbox:
351: (12)                        p1, p2 = bbox.get_points()
352: (12)                        w, h = p2 - p1
353: (12)                        coords = p1[0] * f, p1[1] * f, w * f, h * f
354: (12)                        _writeln(self.fh,
355: (21)                                 r"\pgfpathrectangle"
356: (21)                                 r"{\pgfqpoint{%fin}{%fin}}{\pgfqpoint{%fin}{%fin}}"
357: (21)                                 % coords)
358: (12)                        _writeln(self.fh, r"\pgfusepath{clip}")
359: (8)                     clippath, clippath_trans = gc.get_clip_path()
360: (8)                     if clippath is not None:
361: (12)                        self._print_pgf_path(gc, clippath, clippath_trans)
362: (12)                        _writeln(self.fh, r"\pgfusepath{clip}")
363: (4)                 def _print_pgf_path_styles(self, gc, rgbFace):
364: (8)                     capstyles = {"butt": r"\pgfsetbuttcap",
365: (21)                                  "round": r"\pgfsetroundcap",
366: (21)                                  "projecting": r"\pgfsetrectcap"}
367: (8)                     _writeln(self.fh, capstyles[gc.get_capstyle()])
368: (8)                     joinstyles = {"miter": r"\pgfsetmiterjoin",
369: (22)                                  "round": r"\pgfsetroundjoin",
370: (22)                                  "bevel": r"\pgfsetbeveljoin"}
371: (8)                     _writeln(self.fh, joinstyles[gc.get_joinstyle()])
372: (8)                     has_fill = rgbFace is not None
373: (8)                     if gc.get_forced_alpha():
374: (12)                        fillopacity = strokeopacity = gc.get_alpha()
375: (8)                     else:
376: (12)                        strokeopacity = gc.get_rgb()[3]
377: (12)                        fillopacity = rgbFace[3] if has_fill and len(rgbFace) > 3 else 1.0
378: (8)                     if has_fill:
379: (12)                        _writeln(self.fh,
380: (21)                                 r"\definecolor{currentfill}{rgb}{%f,%f,%f}"
381: (21)                                 % tuple(rgbFace[:3]))
382: (12)                        _writeln(self.fh, r"\pgfsetfillcolor{currentfill}")
383: (8)                     if has_fill and fillopacity != 1.0:
384: (12)                        _writeln(self.fh, r"\pgfsetfillopacity{%f}" % fillopacity)
385: (8)                     lw = gc.get_linewidth() * mpl_pt_to_in * latex_in_to_pt
386: (8)                     stroke_rgba = gc.get_rgb()
387: (8)                     _writeln(self.fh, r"\pgfsetlinewidth{%fpt}" % lw)
388: (8)                     _writeln(self.fh,
389: (17)                             r"\definecolor{currentstroke}{rgb}{%f,%f,%f}"
390: (17)                             % stroke_rgba[:3])
391: (8)                     _writeln(self.fh, r"\pgfsetstrokecolor{currentstroke}")
392: (8)                     if strokeopacity != 1.0:
393: (12)                        _writeln(self.fh, r"\pgfsetstrokeopacity{%f}" % strokeopacity)
394: (8)                     dash_offset, dash_list = gc.get_dashes()
395: (8)                     if dash_list is None:
396: (12)                        _writeln(self.fh, r"\pgfsetdash{}{0pt}")
397: (8)                     else:
398: (12)                        _writeln(self.fh,
399: (21)                                 r"\pgfsetdash{%s}{%fpt}"
400: (21)                                 % ("".join(r"{%fpt}" % dash for dash in dash_list),
401: (24)                                    dash_offset))
402: (4)                 def _print_pgf_path(self, gc, path, transform, rgbFace=None):
403: (8)                     f = 1. / self.dpi
404: (8)                     bbox = gc.get_clip_rectangle() if gc else None
405: (8)                     maxcoord = 16383 / 72.27 * self.dpi  # Max dimensions in LaTeX.
406: (8)                     if bbox and (rgbFace is None):
407: (12)                        p1, p2 = bbox.get_points()
408: (12)                        clip = (max(p1[0], -maxcoord), max(p1[1], -maxcoord),
409: (20)                                min(p2[0], maxcoord), min(p2[1], maxcoord))
```

```
(8)                     else:
(12)                        clip = (-maxcoord, -maxcoord, maxcoord, maxcoord)
(8)                     for points, code in path.iter_segments(transform, clip=clip):
(12)                        if code == Path.MOVETO:
(16)                            x, y = tuple(points)
(16)                            _writeln(self.fh,
(25)                                     r"\pgfpathmoveto{\pgfqpoint{%fin}{%fin}}" %
(25)                                     (f * x, f * y))
(12)                        elif code == Path.CLOSEPOLY:
(16)                            _writeln(self.fh, r"\pgfpathclose")
(12)                        elif code == Path.LINETO:
(16)                            x, y = tuple(points)
(16)                            _writeln(self.fh,
(25)                                     r"\pgfpathlineto{\pgfqpoint{%fin}{%fin}}" %
(25)                                     (f * x, f * y))
(12)                        elif code == Path.CURVE3:
(16)                            cx, cy, px, py = tuple(points)
(16)                            coords = cx * f, cy * f, px * f, py * f
(16)                            _writeln(self.fh,
(25)                                     r"\pgfpathquadraticcurveto"
(25)                                     r"{\pgfqpoint{%fin}{%fin}}{\pgfqpoint{%fin}{%fin}}"
(25)                                     % coords)
(12)                        elif code == Path.CURVE4:
(16)                            c1x, c1y, c2x, c2y, px, py = tuple(points)
(16)                            coords = c1x * f, c1y * f, c2x * f, c2y * f, px * f, py * f
(16)                            _writeln(self.fh,
(25)                                     r"\pgfpathcurveto"
(25)                                     r"{\pgfqpoint{%fin}{%fin}}"
(25)                                     r"{\pgfqpoint{%fin}{%fin}}"
(25)                                     r"{\pgfqpoint{%fin}{%fin}}"
(25)                                     % coords)
(8)                     sketch_params = gc.get_sketch_params() if gc else None
(8)                     if sketch_params is not None:
(12)                        scale, length, randomness = sketch_params
(12)                        if scale is not None:
(16)                            length *= 0.5
(16)                            scale *= 2
(16)                            _writeln(self.fh, r"\usepgfmodule{decorations}")
(16)                            _writeln(self.fh, r"\usepgflibrary{decorations.pathmorphing}")
(16)                            _writeln(self.fh, r"\pgfkeys{/pgf/decoration/.cd, "
(25)                                     f"segment length = {(length * f):f}in, "
(25)                                     f"amplitude = {(scale * f):f}in}}")
(16)                            _writeln(self.fh, f"\\pgfmathsetseed{{{{int(randomness)}}}}")
(16)                            _writeln(self.fh, r"\pgfdecoratecurrentpath{random steps}")
(4)                 def _pgf_path_draw(self, stroke=True, fill=False):
(8)                     actions = []
(8)                     if stroke:
(12)                        actions.append("stroke")
(8)                     if fill:
(12)                        actions.append("fill")
(8)                     _writeln(self.fh, r"\pgfusepath{%s}" % ",".join(actions))
(4)                 def option_scale_image(self):
(8)                     return True
(4)                 def option_image_nocomposite(self):
(8)                     return not mpl.rcParams['image.composite_image']
(4)                 def draw_image(self, gc, x, y, im, transform=None):
(8)                     h, w = im.shape[:2]
(8)                     if w == 0 or h == 0:
(12)                        return
(8)                     if not os.path.exists(getattr(self.fh, "name", "")):
(12)                        raise ValueError(
(16)                            "streamed pgf-code does not support raster graphics, consider
"
(16)                            "using the pgf-to-pdf option")
(8)                     path = pathlib.Path(self.fh.name)
(8)                     fname_img = "%s-img%d.png" % (path.stem, self.image_counter)
(8)                     Image.fromarray(im[::-1]).save(path.parent / fname_img)
(8)                     self.image_counter += 1
(8)                     _writeln(self.fh, r"\begin{pgfscope}")
```

```
(8)                     self._print_pgf_clip(gc)
(8)                     f = 1. / self.dpi  # from display coords to inch
(8)                     if transform is None:
(12)                        _writeln(self.fh,
(21)                                 r"\pgfsys@transformshift{%fin}{%fin}" % (x * f, y * f))
(12)                        w, h = w * f, h * f
(8)                     else:
(12)                        tr1, tr2, tr3, tr4, tr5, tr6 = transform.frozen().to_values()
(12)                        _writeln(self.fh,
(21)                                 r"\pgfsys@transformcm{%f}{%f}{%f}{%f}{%fin}{%fin}" %
(21)                                 (tr1 * f, tr2 * f, tr3 * f, tr4 * f,
(22)                                  (tr5 + x) * f, (tr6 + y) * f))
(12)                        w = h = 1  # scale is already included in the transform
(8)                     interp = str(transform is None).lower()  # interpolation in PDF reader
(8)                     _writeln(self.fh,
(17)                             r"\pgftext[left,bottom]"
(17)                             r"{%s[interpolate=%s,width=%fin,height=%fin]{%s}}" %
(17)                             (_get_image_inclusion_command(),
(18)                              interp, w, h, fname_img))
(8)                     _writeln(self.fh, r"\end{pgfscope}")
(4)                 def draw_tex(self, gc, x, y, s, prop, angle, *, mtext=None):
(8)                     self.draw_text(gc, x, y, s, prop, angle, ismath="TeX", mtext=mtext)
(4)                 def draw_text(self, gc, x, y, s, prop, angle, ismath=False, mtext=None):
(8)                     s = _escape_and_apply_props(s, prop)
(8)                     _writeln(self.fh, r"\begin{pgfscope}")
(8)                     self._print_pgf_clip(gc)
(8)                     alpha = gc.get_alpha()
(8)                     if alpha != 1.0:
(12)                        _writeln(self.fh, r"\pgfsetfillopacity{%f}" % alpha)
(12)                        _writeln(self.fh, r"\pgfsetstrokeopacity{%f}" % alpha)
(8)                     rgb = tuple(gc.get_rgb())[:3]
(8)                     _writeln(self.fh, r"\definecolor{textcolor}{rgb}{%f,%f,%f}" % rgb)
(8)                     _writeln(self.fh, r"\pgfsetstrokecolor{textcolor}")
(8)                     _writeln(self.fh, r"\pgfsetfillcolor{textcolor}")
(8)                     s = r"\color{textcolor}" + s
(8)                     dpi = self.figure.dpi
(8)                     text_args = []
(8)                     if mtext and (
(16)                            (angle == 0 or
(17)                             mtext.get_rotation_mode() == "anchor") and
(16)                            mtext.get_verticalalignment() != "center_baseline"):
(12)                        pos = mtext.get_unitless_position()
(12)                        x, y = mtext.get_transform().transform(pos)
(12)                        halign = {"left": "left", "right": "right", "center": ""}
(12)                        valign = {"top": "top", "bottom": "bottom",
(22)                                  "baseline": "base", "center": ""}
(12)                        text_args.extend([
(16)                            f"x={x/dpi:f}in",
(16)                            f"y={y/dpi:f}in",
(16)                            halign[mtext.get_horizontalalignment()],
(16)                            valign[mtext.get_verticalalignment()],
(12)                        ])
(8)                     else:
(12)                        text_args.append(f"x={x/dpi:f}in, y={y/dpi:f}in, left, base")
(8)                     if angle != 0:
(12)                        text_args.append("rotate=%f" % angle)
(8)                     _writeln(self.fh, r"\pgftext[%s]{%s}" % (",".join(text_args), s))
(8)                     _writeln(self.fh, r"\end{pgfscope}")
(4)                 def get_text_width_height_descent(self, s, prop, ismath):
(8)                     w, h, d = (LatexManager._get_cached_or_new()
(19)                               .get_width_height_descent(s, prop))
(8)                     f = mpl_pt_to_in * self.dpi
(8)                     return w * f, h * f, d * f
(4)                 def flipy(self):
(8)                     return False
(4)                 def get_canvas_width_height(self):
(8)                     return (self.figure.get_figwidth() * self.dpi,
(16)                            self.figure.get_figheight() * self.dpi)
(4)                 def points_to_pixels(self, points):
```

```
547: (8)                    return points * mpl_pt_to_in * self.dpi
548: (0)            class FigureCanvasPgf(FigureCanvasBase):
549: (4)                filetypes = {"pgf": "LaTeX PGF picture",
550: (17)                            "pdf": "LaTeX compiled PGF picture",
551: (17)                            "png": "Portable Network Graphics", }
552: (4)                def get_default_filetype(self):
553: (8)                    return 'pdf'
554: (4)                def _print_pgf_to_fh(self, fh, *, bbox_inches_restore=None):
555: (8)                    header_text = """%% Creator: Matplotlib, PGF backend
556: (0)            %%
557: (0)            %% To include the figure in your LaTeX document, write
558: (0)            %%   \\input{<filename>.pgf}
559: (0)            %%
560: (0)            %% Make sure the required packages are loaded in your preamble
561: (0)            %%   \\usepackage{pgf}
562: (0)            %%
563: (0)            %% Also ensure that all the required font packages are loaded; for instance,
564: (0)            %% the lmodern package is sometimes necessary when using math font.
565: (0)            %%   \\usepackage{lmodern}
566: (0)            %%
567: (0)            %% Figures using additional raster images can only be included by \\input if
568: (0)            %% they are in the same directory as the main LaTeX file. For loading figures
569: (0)            %% from other directories you can use the `import` package
570: (0)            %%   \\usepackage{import}
571: (0)            %%
572: (0)            %% and then include the figures with
573: (0)            %%   \\import{<path to file>}{<filename>.pgf}
574: (0)            %%
575: (0)            """
576: (8)                    header_info_preamble = ["%% Matplotlib used the following preamble"]
577: (8)                    for line in _get_preamble().splitlines():
578: (12)                       header_info_preamble.append("%%   " + line)
579: (8)                    header_info_preamble.append("%%")
580: (8)                    header_info_preamble = "\n".join(header_info_preamble)
581: (8)                    w, h = self.figure.get_figwidth(), self.figure.get_figheight()
582: (8)                    dpi = self.figure.dpi
583: (8)                    fh.write(header_text)
584: (8)                    fh.write(header_info_preamble)
585: (8)                    fh.write("\n")
586: (8)                    _writeln(fh, r"\begingroup")
587: (8)                    _writeln(fh, r"\makeatletter")
588: (8)                    _writeln(fh, r"\begin{pgfpicture}")
589: (8)                    _writeln(fh,
590: (17)                           r"\pgfpathrectangle{\pgfpointorigin}{\pgfqpoint{%fin}{%fin}}"
591: (17)                           % (w, h))
592: (8)                    _writeln(fh, r"\pgfusepath{use as bounding box, clip}")
593: (8)                    renderer = MixedModeRenderer(self.figure, w, h, dpi,
594: (37)                                                RendererPgf(self.figure, fh),
595: (37)                                                bbox_inches_restore=bbox_inches_restore)
596: (8)                    self.figure.draw(renderer)
597: (8)                    _writeln(fh, r"\end{pgfpicture}")
598: (8)                    _writeln(fh, r"\makeatother")
599: (8)                    _writeln(fh, r"\endgroup")
600: (4)                def print_pgf(self, fname_or_fh, **kwargs):
601: (8)                    """
602: (8)                    Output pgf macros for drawing the figure so it can be included and
603: (8)                    rendered in latex documents.
604: (8)                    """
605: (8)                    with cbook.open_file_cm(fname_or_fh, "w", encoding="utf-8") as file:
606: (12)                       if not cbook.file_requires_unicode(file):
607: (16)                           file = codecs.getwriter("utf-8")(file)
608: (12)                       self._print_pgf_to_fh(file, **kwargs)
609: (4)                def print_pdf(self, fname_or_fh, *, metadata=None, **kwargs):
610: (8)                    """Use LaTeX to compile a pgf generated figure to pdf."""
611: (8)                    w, h = self.figure.get_size_inches()
612: (8)                    info_dict = _create_pdf_info_dict('pgf', metadata or {})
613: (8)                    pdfinfo = ','.join(
614: (12)                       _metadata_to_str(k, v) for k, v in info_dict.items())
615: (8)                    with TemporaryDirectory() as tmpdir:
```

```
616: (12)                        tmppath = pathlib.Path(tmpdir)
617: (12)                        self.print_pgf(tmppath / "figure.pgf", **kwargs)
618: (12)                        (tmppath / "figure.tex").write_text(
619: (16)                            "\n".join([
620: (20)                                r"\documentclass[12pt]{article}",
621: (20)                                r"\usepackage[pdfinfo={%s}]{hyperref}" % pdfinfo,
622: (20)                                r"\usepackage[papersize={%fin,%fin}, margin=0in]
{geometry}"
623: (20)                                % (w, h),
624: (20)                                r"\usepackage{pgf}",
625: (20)                                _get_preamble(),
626: (20)                                r"\begin{document}",
627: (20)                                r"\centering",
628: (20)                                r"\input{figure.pgf}",
629: (20)                                r"\end{document}",
630: (16)                            ]), encoding="utf-8")
631: (12)                        texcommand = mpl.rcParams["pgf.texsystem"]
632: (12)                        cbook._check_and_log_subprocess(
633: (16)                            [texcommand, "-interaction=nonstopmode", "-halt-on-error",
634: (17)                             "figure.tex"], _log, cwd=tmpdir)
635: (12)                        with (tmppath / "figure.pdf").open("rb") as orig, \
636: (17)                             cbook.open_file_cm(fname_or_fh, "wb") as dest:
637: (16)                            shutil.copyfileobj(orig, dest)  # copy file contents to target
638: (4)                    def print_png(self, fname_or_fh, **kwargs):
639: (8)                        """Use LaTeX to compile a pgf figure to pdf and convert it to png."""
640: (8)                        converter = make_pdf_to_png_converter()
641: (8)                        with TemporaryDirectory() as tmpdir:
642: (12)                        tmppath = pathlib.Path(tmpdir)
643: (12)                        pdf_path = tmppath / "figure.pdf"
644: (12)                        png_path = tmppath / "figure.png"
645: (12)                        self.print_pdf(pdf_path, **kwargs)
646: (12)                        converter(pdf_path, png_path, dpi=self.figure.dpi)
647: (12)                        with png_path.open("rb") as orig, \
648: (17)                             cbook.open_file_cm(fname_or_fh, "wb") as dest:
649: (16)                            shutil.copyfileobj(orig, dest)  # copy file contents to target
650: (4)                    def get_renderer(self):
651: (8)                        return RendererPgf(self.figure, None)
652: (4)                    def draw(self):
653: (8)                        self.figure.draw_without_rendering()
654: (8)                        return super().draw()
655: (0)                FigureManagerPgf = FigureManagerBase
656: (0)                @_Backend.export
657: (0)                class _BackendPgf(_Backend):
658: (4)                    FigureCanvas = FigureCanvasPgf
659: (0)                class PdfPages:
660: (4)                    """
661: (4)                    A multi-page PDF file using the pgf backend
662: (4)                    Examples
663: (4)                    --------
664: (4)                    >>> import matplotlib.pyplot as plt
665: (4)                    >>> # Initialize:
666: (4)                    >>> with PdfPages('foo.pdf') as pdf:
667: (4)                    ...     # As many times as you like, create a figure fig and save it:
668: (4)                    ...     fig = plt.figure()
669: (4)                    ...     pdf.savefig(fig)
670: (4)                    ...     # When no figure is specified the current figure is saved
671: (4)                    ...     pdf.savefig()
672: (4)                    """
673: (4)                    _UNSET = object()
674: (4)                    def __init__(self, filename, *, keep_empty=_UNSET, metadata=None):
675: (8)                        """
676: (8)                        Create a new PdfPages object.
677: (8)                        Parameters
678: (8)                        ----------
679: (8)                        filename : str or path-like
680: (12)                           Plots using `PdfPages.savefig` will be written to a file at this
681: (12)                           location. Any older file with the same name is overwritten.
682: (8)                        keep_empty : bool, default: True
683: (12)                           If set to False, then empty pdf files will be deleted
```

```
automatically
            when closed.
        metadata : dict, optional
            Information dictionary object (see PDF reference section 10.2.1
            'Document Information Dictionary'), e.g.:
            ``{'Creator': 'My software', 'Author': 'Me', 'Title': 'Awesome'}``.
            The standard keys are 'Title', 'Author', 'Subject', 'Keywords',
            'Creator', 'Producer', 'CreationDate', 'ModDate', and
            'Trapped'. Values have been predefined for 'Creator', 'Producer'
            and 'CreationDate'. They can be removed by setting them to `None`.
            Note that some versions of LaTeX engines may ignore the 'Producer'
            key and set it to themselves.
        """
        self._output_name = filename
        self._n_figures = 0
        if keep_empty and keep_empty is not self._UNSET:
            _api.warn_deprecated("3.8", message=(
                "Keeping empty pdf files is deprecated since %(since)s and support "
                "will be removed %(removal)s."))
        self._keep_empty = keep_empty
        self._metadata = (metadata or {}).copy()
        self._info_dict = _create_pdf_info_dict('pgf', self._metadata)
        self._file = BytesIO()
    keep_empty = _api.deprecate_privatize_attribute("3.8")
    def _write_header(self, width_inches, height_inches):
        pdfinfo = ','.join(
            _metadata_to_str(k, v) for k, v in self._info_dict.items())
        latex_header = "\n".join([
            r"\documentclass[12pt]{article}",
            r"\usepackage[pdfinfo={%s}]{hyperref}" % pdfinfo,
            r"\usepackage[papersize={%fin,%fin}, margin=0in]{geometry}"
            % (width_inches, height_inches),
            r"\usepackage{pgf}",
            _get_preamble(),
            r"\setlength{\parindent}{0pt}",
            r"\begin{document}%",
        ])
        self._file.write(latex_header.encode('utf-8'))
    def __enter__(self):
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        self.close()
    def close(self):
        """
        Finalize this object, running LaTeX in a temporary directory
        and moving the final pdf file to *filename*.
        """
        self._file.write(rb'\end{document}\n')
        if self._n_figures > 0:
            self._run_latex()
        elif self._keep_empty:
            _api.warn_deprecated("3.8", message=(
                "Keeping empty pdf files is deprecated since %(since)s and support "
                "will be removed %(removal)s."))
            open(self._output_name, 'wb').close()
        self._file.close()
    def _run_latex(self):
        texcommand = mpl.rcParams["pgf.texsystem"]
        with TemporaryDirectory() as tmpdir:
            tex_source = pathlib.Path(tmpdir, "pdf_pages.tex")
            tex_source.write_bytes(self._file.getvalue())
            cbook._check_and_log_subprocess(
                [texcommand, "-interaction=nonstopmode", "-halt-on-error",
                 tex_source],
                _log, cwd=tmpdir)
            shutil.move(tex_source.with_suffix(".pdf"), self._output_name)
```

```
(4)     def savefig(self, figure=None, **kwargs):
(8)         """
(8)         Save a `.Figure` to this file as a new page.
(8)         Any other keyword arguments are passed to `~.Figure.savefig`.
(8)         Parameters
(8)         ----------
(8)         figure : `.Figure` or int, default: the active figure
(12)            The figure, or index of the figure, that is saved to the file.
(8)         """
(8)         if not isinstance(figure, Figure):
(12)            if figure is None:
(16)                manager = Gcf.get_active()
(12)            else:
(16)                manager = Gcf.get_fig_manager(figure)
(12)            if manager is None:
(16)                raise ValueError(f"No figure {figure}")
(12)            figure = manager.canvas.figure
(8)         width, height = figure.get_size_inches()
(8)         if self._n_figures == 0:
(12)            self._write_header(width, height)
(8)         else:
(12)            self._file.write(
(16)                (
(20)                    r'\newpage'
(20)                    r'\ifdefined\pdfpagewidth\pdfpagewidth'
(20)                    fr'\else\pagewidth\fi={width}in'
(20)                    r'\ifdefined\pdfpageheight\pdfpageheight'
(20)                    fr'\else\pageheight\fi={height}in'
(20)                    '%%\n'
(16)                ).encode("ascii")
(12)            )
(8)         figure.savefig(self._file, format="pgf", backend="pgf", **kwargs)
(8)         self._n_figures += 1
(4)     def get_pagecount(self):
(8)         """Return the current number of pages in the multipage pdf file."""
(8)         return self._n_figures
```

File 94 - backend_ps.py:

```
(0)     """
(0)     A PostScript backend, which can produce both PostScript .ps and .eps.
(0)     """
(0)     import codecs
(0)     import datetime
(0)     from enum import Enum
(0)     import functools
(0)     from io import StringIO
(0)     import itertools
(0)     import logging
(0)     import math
(0)     import os
(0)     import pathlib
(0)     import shutil
(0)     from tempfile import TemporaryDirectory
(0)     import time
(0)     import numpy as np
(0)     import matplotlib as mpl
(0)     from matplotlib import _api, cbook, _path, _text_helpers
(0)     from matplotlib._afm import AFM
(0)     from matplotlib.backend_bases import (
(4)         _Backend, FigureCanvasBase, FigureManagerBase, RendererBase)
(0)     from matplotlib.cbook import is_writable_file_like, file_requires_unicode
(0)     from matplotlib.font_manager import get_font
(0)     from matplotlib.ft2font import LOAD_NO_SCALE, FT2Font
(0)     from matplotlib._ttconv import convert_ttf_to_ps
(0)     from matplotlib._mathtext_data import uni2type1
(0)     from matplotlib.path import Path
```

```
(0)             from matplotlib.texmanager import TexManager
(0)             from matplotlib.transforms import Affine2D
(0)             from matplotlib.backends.backend_mixed import MixedModeRenderer
(0)             from . import _backend_pdf_ps
(0)             _log = logging.getLogger(__name__)
(0)             debugPS = False
(0)             @_api.caching_module_getattr
(0)             class __getattr__:
(4)                 psDefs = _api.deprecated("3.8", obj_type="")(property(lambda self:
_psDefs))
(0)             papersize = {'letter': (8.5, 11),
(13)                         'legal': (8.5, 14),
(13)                         'ledger': (11, 17),
(13)                         'a0': (33.11, 46.81),
(13)                         'a1': (23.39, 33.11),
(13)                         'a2': (16.54, 23.39),
(13)                         'a3': (11.69, 16.54),
(13)                         'a4': (8.27, 11.69),
(13)                         'a5': (5.83, 8.27),
(13)                         'a6': (4.13, 5.83),
(13)                         'a7': (2.91, 4.13),
(13)                         'a8': (2.05, 2.91),
(13)                         'a9': (1.46, 2.05),
(13)                         'a10': (1.02, 1.46),
(13)                         'b0': (40.55, 57.32),
(13)                         'b1': (28.66, 40.55),
(13)                         'b2': (20.27, 28.66),
(13)                         'b3': (14.33, 20.27),
(13)                         'b4': (10.11, 14.33),
(13)                         'b5': (7.16, 10.11),
(13)                         'b6': (5.04, 7.16),
(13)                         'b7': (3.58, 5.04),
(13)                         'b8': (2.51, 3.58),
(13)                         'b9': (1.76, 2.51),
(13)                         'b10': (1.26, 1.76)}
(0)             def _get_papertype(w, h):
(4)                 for key, (pw, ph) in sorted(papersize.items(), reverse=True):
(8)                     if key.startswith('l'):
(12)                        continue
(8)                     if w < pw and h < ph:
(12)                        return key
(4)                 return 'a0'
(0)             def _nums_to_str(*args, sep=" "):
(4)                 return sep.join(f"{arg:1.3f}".rstrip("0").rstrip(".") for arg in args)
(0)             def _move_path_to_path_or_stream(src, dst):
(4)                 """
(4)                 Move the contents of file at *src* to path-or-filelike *dst*.
(4)                 If *dst* is a path, the metadata of *src* are *not* copied.
(4)                 """
(4)                 if is_writable_file_like(dst):
(8)                     fh = (open(src, encoding='latin-1')
(14)                        if file_requires_unicode(dst)
(14)                        else open(src, 'rb'))
(8)                     with fh:
(12)                        shutil.copyfileobj(fh, dst)
(4)                 else:
(8)                     shutil.move(src, dst, copy_function=shutil.copyfile)
(0)             def _font_to_ps_type3(font_path, chars):
(4)                 """
(4)                 Subset *chars* from the font at *font_path* into a Type 3 font.
(4)                 Parameters
(4)                 ----------
(4)                 font_path : path-like
(8)                     Path to the font to be subsetted.
(4)                 chars : str
(8)                     The characters to include in the subsetted font.
(4)                 Returns
(4)                 -------
(4)                 str
```

```
 97: (8)                    The string representation of a Type 3 font, which can be included
 98: (8)                    verbatim into a PostScript file.
 99: (4)                """
100: (4)                font = get_font(font_path, hinting_factor=1)
101: (4)                glyph_ids = [font.get_char_index(c) for c in chars]
102: (4)                preamble = """\
103: (0)            %!PS-Adobe-3.0 Resource-Font
104: (0)            %%Creator: Converted from TrueType to Type 3 by Matplotlib.
105: (0)            10 dict begin
106: (0)            /FontName /{font_name} def
107: (0)            /PaintType 0 def
108: (0)            /FontMatrix [{inv_units_per_em} 0 0 {inv_units_per_em} 0 0] def
109: (0)            /FontBBox [{bbox}] def
110: (0)            /FontType 3 def
111: (0)            /Encoding [{encoding}] def
112: (0)            /CharStrings {num_glyphs} dict dup begin
113: (0)            /.notdef 0 def
114: (0)            """.format(font_name=font.postscript_name,
115: (11)                       inv_units_per_em=1 / font.units_per_EM,
116: (11)                       bbox=" ".join(map(str, font.bbox)),
117: (11)                       encoding=" ".join(f"/{font.get_glyph_name(glyph_id)}"
118: (29)                                         for glyph_id in glyph_ids),
119: (11)                       num_glyphs=len(glyph_ids) + 1)
120: (4)                postamble = """
121: (0)            end readonly def
122: (0)            /BuildGlyph {
123: (1)             exch begin
124: (1)             CharStrings exch
125: (1)             2 copy known not {pop /.notdef} if
126: (1)             true 3 1 roll get exec
127: (1)             end
128: (0)            } _d
129: (0)            /BuildChar {
130: (1)             1 index /Encoding get exch get
131: (1)             1 index /BuildGlyph get exec
132: (0)            } _d
133: (0)            FontName currentdict end definefont pop
134: (0)            """
135: (4)                entries = []
136: (4)                for glyph_id in glyph_ids:
137: (8)                    g = font.load_glyph(glyph_id, LOAD_NO_SCALE)
138: (8)                    v, c = font.get_path()
139: (8)                    entries.append(
140: (12)                       "/%(name)s{%(bbox)s sc\n" % {
141: (16)                           "name": font.get_glyph_name(glyph_id),
142: (16)                           "bbox": " ".join(map(str, [g.horiAdvance, 0, *g.bbox])),
143: (12)                       }
144: (12)                       + _path.convert_to_string(
145: (16)                           Path(v * 64, c), None, None, False, None, 0,
146: (16)                           [b"m", b"l", b"", b"c", b""], True).decode("ascii")
147: (12)                       + "ce} _d"
148: (8)                    )
149: (4)                return preamble + "\n".join(entries) + postamble
150: (0)            def _font_to_ps_type42(font_path, chars, fh):
151: (4)                """
152: (4)                Subset *chars* from the font at *font_path* into a Type 42 font at *fh*.
153: (4)                Parameters
154: (4)                ----------
155: (4)                font_path : path-like
156: (8)                    Path to the font to be subsetted.
157: (4)                chars : str
158: (8)                    The characters to include in the subsetted font.
159: (4)                fh : file-like
160: (8)                    Where to write the font.
161: (4)                """
162: (4)                subset_str = ''.join(chr(c) for c in chars)
163: (4)                _log.debug("SUBSET %s characters: %s", font_path, subset_str)
164: (4)                try:
165: (8)                    fontdata = _backend_pdf_ps.get_glyphs_subset(font_path, subset_str)
```

```
166: (8)                    _log.debug("SUBSET %s %d -> %d", font_path,
os.stat(font_path).st_size,
167: (19)                               fontdata.getbuffer().nbytes)
168: (8)                    font = FT2Font(fontdata)
169: (8)                    glyph_ids = [font.get_char_index(c) for c in chars]
170: (8)                    with TemporaryDirectory() as tmpdir:
171: (12)                       tmpfile = os.path.join(tmpdir, "tmp.ttf")
172: (12)                       with open(tmpfile, 'wb') as tmp:
173: (16)                           tmp.write(fontdata.getvalue())
174: (12)                       convert_ttf_to_ps(os.fsencode(tmpfile), fh, 42, glyph_ids)
175: (4)                except RuntimeError:
176: (8)                    _log.warning(
177: (12)                       "The PostScript backend does not currently "
178: (12)                       "support the selected font.")
179: (8)                    raise
180: (0)            def _log_if_debug_on(meth):
181: (4)                """
182: (4)                Wrap `RendererPS` method *meth* to emit a PS comment with the method name,
183: (4)                if the global flag `debugPS` is set.
184: (4)                """
185: (4)                @functools.wraps(meth)
186: (4)                def wrapper(self, *args, **kwargs):
187: (8)                    if debugPS:
188: (12)                       self._pswriter.write(f"% {meth.__name__}\n")
189: (8)                    return meth(self, *args, **kwargs)
190: (4)                return wrapper
191: (0)            class RendererPS(_backend_pdf_ps.RendererPDFPSBase):
192: (4)                """
193: (4)                The renderer handles all the drawing primitives using a graphics
194: (4)                context instance that controls the colors/styles.
195: (4)                """
196: (4)                _afm_font_dir = cbook._get_data_path("fonts/afm")
197: (4)                _use_afm_rc_name = "ps.useafm"
198: (4)                def __init__(self, width, height, pswriter, imagedpi=72):
199: (8)                    super().__init__(width, height)
200: (8)                    self._pswriter = pswriter
201: (8)                    if mpl.rcParams['text.usetex']:
202: (12)                       self.textcnt = 0
203: (12)                       self.psfrag = []
204: (8)                    self.imagedpi = imagedpi
205: (8)                    self.color = None
206: (8)                    self.linewidth = None
207: (8)                    self.linejoin = None
208: (8)                    self.linecap = None
209: (8)                    self.linedash = None
210: (8)                    self.fontname = None
211: (8)                    self.fontsize = None
212: (8)                    self._hatches = {}
213: (8)                    self.image_magnification = imagedpi / 72
214: (8)                    self._clip_paths = {}
215: (8)                    self._path_collection_id = 0
216: (8)                    self._character_tracker = _backend_pdf_ps.CharacterTracker()
217: (8)                    self._logwarn_once = functools.cache(_log.warning)
218: (4)                def _is_transparent(self, rgb_or_rgba):
219: (8)                    if rgb_or_rgba is None:
220: (12)                       return True  # Consistent with rgbFace semantics.
221: (8)                    elif len(rgb_or_rgba) == 4:
222: (12)                       if rgb_or_rgba[3] == 0:
223: (16)                           return True
224: (12)                       if rgb_or_rgba[3] != 1:
225: (16)                           self._logwarn_once(
226: (20)                               "The PostScript backend does not support transparency; "
227: (20)                               "partially transparent artists will be rendered opaque.")
228: (12)                       return False
229: (8)                    else:  # len() == 3.
230: (12)                       return False
231: (4)                def set_color(self, r, g, b, store=True):
232: (8)                    if (r, g, b) != self.color:
233: (12)                       self._pswriter.write(f"{_nums_to_str(r)} setgray\n"
```

```
234: (33)                                         if r == g == b else
235: (33)                                         f"{_nums_to_str(r, g, b)} setrgbcolor\n")
236: (12)                        if store:
237: (16)                            self.color = (r, g, b)
238: (4)                 def set_linewidth(self, linewidth, store=True):
239: (8)                     linewidth = float(linewidth)
240: (8)                     if linewidth != self.linewidth:
241: (12)                        self._pswriter.write(f"{_nums_to_str(linewidth)} setlinewidth\n")
242: (12)                        if store:
243: (16)                            self.linewidth = linewidth
244: (4)                 @staticmethod
245: (4)                 def _linejoin_cmd(linejoin):
246: (8)                     linejoin = {'miter': 0, 'round': 1, 'bevel': 2, 0: 0, 1: 1, 2: 2}[
247: (12)                        linejoin]
248: (8)                     return f"{linejoin:d} setlinejoin\n"
249: (4)                 def set_linejoin(self, linejoin, store=True):
250: (8)                     if linejoin != self.linejoin:
251: (12)                        self._pswriter.write(self._linejoin_cmd(linejoin))
252: (12)                        if store:
253: (16)                            self.linejoin = linejoin
254: (4)                 @staticmethod
255: (4)                 def _linecap_cmd(linecap):
256: (8)                     linecap = {'butt': 0, 'round': 1, 'projecting': 2, 0: 0, 1: 1, 2: 2}[
257: (12)                        linecap]
258: (8)                     return f"{linecap:d} setlinecap\n"
259: (4)                 def set_linecap(self, linecap, store=True):
260: (8)                     if linecap != self.linecap:
261: (12)                        self._pswriter.write(self._linecap_cmd(linecap))
262: (12)                        if store:
263: (16)                            self.linecap = linecap
264: (4)                 def set_linedash(self, offset, seq, store=True):
265: (8)                     if self.linedash is not None:
266: (12)                        oldo, oldseq = self.linedash
267: (12)                        if np.array_equal(seq, oldseq) and oldo == offset:
268: (16)                            return
269: (8)                     self._pswriter.write(f"[{_nums_to_str(*seq)}] {_nums_to_str(offset)}
setdash\n"
270: (29)                                         if seq is not None and len(seq) else
271: (29)                                         "[] 0 setdash\n")
272: (8)                     if store:
273: (12)                        self.linedash = (offset, seq)
274: (4)                 def set_font(self, fontname, fontsize, store=True):
275: (8)                     if (fontname, fontsize) != (self.fontname, self.fontsize):
276: (12)                        self._pswriter.write(f"/{fontname} {fontsize:1.3f} selectfont\n")
277: (12)                        if store:
278: (16)                            self.fontname = fontname
279: (16)                            self.fontsize = fontsize
280: (4)                 def create_hatch(self, hatch):
281: (8)                     sidelen = 72
282: (8)                     if hatch in self._hatches:
283: (12)                        return self._hatches[hatch]
284: (8)                     name = 'H%d' % len(self._hatches)
285: (8)                     linewidth = mpl.rcParams['hatch.linewidth']
286: (8)                     pageheight = self.height * 72
287: (8)                     self._pswriter.write(f"""\
288: (2)               << /PatternType 1
289: (5)                  /PaintType 2
290: (5)                  /TilingType 2
291: (5)                  /BBox[0 0 {sidelen:d} {sidelen:d}]
292: (5)                  /XStep {sidelen:d}
293: (5)                  /YStep {sidelen:d}
294: (5)                  /PaintProc {{
295: (8)                     pop
296: (8)                     {linewidth:g} setlinewidth
297: (0)             {self._convert_path(Path.hatch(hatch), Affine2D().scale(sidelen),
simplify=False)}
298: (8)                     gsave
299: (8)                     fill
300: (8)                     grestore
```

```
(8)                 stroke
(5)             }} bind
(3)           >>
(3)           matrix
(3)           0 {pageheight:g} translate
(3)           makepattern
(3)           /{name} exch def
(0)         """)
(8)                 self._hatches[hatch] = name
(8)                 return name
(4)             def get_image_magnification(self):
(8)                 """
(8)                 Get the factor by which to magnify images passed to draw_image.
(8)                 Allows a backend to have images at a different resolution to other
(8)                 artists.
(8)                 """
(8)                 return self.image_magnification
(4)             def _convert_path(self, path, transform, clip=False, simplify=None):
(8)                 if clip:
(12)                    clip = (0.0, 0.0, self.width * 72.0, self.height * 72.0)
(8)                 else:
(12)                    clip = None
(8)                 return _path.convert_to_string(
(12)                    path, transform, clip, simplify, None,
(12)                    6, [b"m", b"l", b"", b"c", b"cl"], True).decode("ascii")
(4)             def _get_clip_cmd(self, gc):
(8)                 clip = []
(8)                 rect = gc.get_clip_rectangle()
(8)                 if rect is not None:
(12)                    clip.append(f"{_nums_to_str(*rect.p0, *rect.size)} rectclip\n")
(8)                 path, trf = gc.get_clip_path()
(8)                 if path is not None:
(12)                    key = (path, id(trf))
(12)                    custom_clip_cmd = self._clip_paths.get(key)
(12)                    if custom_clip_cmd is None:
(16)                        custom_clip_cmd = "c%d" % len(self._clip_paths)
(16)                        self._pswriter.write(f"""\
(0)         /{custom_clip_cmd} {{
(0)         {self._convert_path(path, trf, simplify=False)}
(0)         clip
(0)         newpath
(0)         }} bind def
(0)         """)
(16)                        self._clip_paths[key] = custom_clip_cmd
(12)                    clip.append(f"{custom_clip_cmd}\n")
(8)                 return "".join(clip)
(4)             @_log_if_debug_on
(4)             def draw_image(self, gc, x, y, im, transform=None):
(8)                 h, w = im.shape[:2]
(8)                 imagecmd = "false 3 colorimage"
(8)                 data = im[::-1, :, :3]  # Vertically flipped rgb values.
(8)                 hexdata = data.tobytes().hex("\n", -64)  # Linewrap to 128 chars.
(8)                 if transform is None:
(12)                    matrix = "1 0 0 1 0 0"
(12)                    xscale = w / self.image_magnification
(12)                    yscale = h / self.image_magnification
(8)                 else:
(12)                    matrix = " ".join(map(str, transform.frozen().to_values()))
(12)                    xscale = 1.0
(12)                    yscale = 1.0
(8)                 self._pswriter.write(f"""\
(0)         gsave
(0)         {self._get_clip_cmd(gc)}
(0)         {x:g} {y:g} translate
(0)         [{matrix}] concat
(0)         {xscale:g} {yscale:g} scale
(0)         /DataString {w:d} string def
(0)         {w:d} {h:d} 8 [ {w:d} 0 0 -{h:d} 0 {h:d} ]
(0)         {{
```

```
370: (0)            currentfile DataString readhexstring pop
371: (0)            }} bind {imagecmd}
372: (0)            {hexdata}
373: (0)            grestore
374: (0)            """)
375: (4)                @_log_if_debug_on
376: (4)                def draw_path(self, gc, path, transform, rgbFace=None):
377: (8)                    clip = rgbFace is None and gc.get_hatch_path() is None
378: (8)                    simplify = path.should_simplify and clip
379: (8)                    ps = self._convert_path(path, transform, clip=clip, simplify=simplify)
380: (8)                    self._draw_ps(ps, gc, rgbFace)
381: (4)                @_log_if_debug_on
382: (4)                def draw_markers(
383: (12)                        self, gc, marker_path, marker_trans, path, trans, rgbFace=None):
384: (8)                    ps_color = (
385: (12)                        None
386: (12)                        if self._is_transparent(rgbFace)
387: (12)                        else f'{_nums_to_str(rgbFace[0])} setgray'
388: (12)                        if rgbFace[0] == rgbFace[1] == rgbFace[2]
389: (12)                        else f'{_nums_to_str(*rgbFace[:3])} setrgbcolor')
390: (8)                    ps_cmd = ['/o {', 'gsave', 'newpath', 'translate']
391: (8)                    lw = gc.get_linewidth()
392: (8)                    alpha = (gc.get_alpha()
393: (17)                             if gc.get_forced_alpha() or len(gc.get_rgb()) == 3
394: (17)                             else gc.get_rgb()[3])
395: (8)                    stroke = lw > 0 and alpha > 0
396: (8)                    if stroke:
397: (12)                        ps_cmd.append('%.1f setlinewidth' % lw)
398: (12)                        ps_cmd.append(self._linejoin_cmd(gc.get_joinstyle()))
399: (12)                        ps_cmd.append(self._linecap_cmd(gc.get_capstyle()))
400: (8)                    ps_cmd.append(self._convert_path(marker_path, marker_trans,
401: (41)                                                     simplify=False))
402: (8)                    if rgbFace:
403: (12)                        if stroke:
404: (16)                            ps_cmd.append('gsave')
405: (12)                        if ps_color:
406: (16)                            ps_cmd.extend([ps_color, 'fill'])
407: (12)                        if stroke:
408: (16)                            ps_cmd.append('grestore')
409: (8)                    if stroke:
410: (12)                        ps_cmd.append('stroke')
411: (8)                    ps_cmd.extend(['grestore', '} bind def'])
412: (8)                    for vertices, code in path.iter_segments(
413: (16)                            trans,
414: (16)                            clip=(0, 0, self.width*72, self.height*72),
415: (16)                            simplify=False):
416: (12)                        if len(vertices):
417: (16)                            x, y = vertices[-2:]
418: (16)                            ps_cmd.append(f"{x:g} {y:g} o")
419: (8)                    ps = '\n'.join(ps_cmd)
420: (8)                    self._draw_ps(ps, gc, rgbFace, fill=False, stroke=False)
421: (4)                @_log_if_debug_on
422: (4)                def draw_path_collection(self, gc, master_transform, paths,
all_transforms,
423: (29)                                        offsets, offset_trans, facecolors, edgecolors,
424: (29)                                        linewidths, linestyles, antialiaseds, urls,
425: (29)                                        offset_position):
426: (8)                    len_path = len(paths[0].vertices) if len(paths) > 0 else 0
427: (8)                    uses_per_path = self._iter_collection_uses_per_path(
428: (12)                        paths, all_transforms, offsets, facecolors, edgecolors)
429: (8)                    should_do_optimization = \
430: (12)                        len_path + 3 * uses_per_path + 3 < (len_path + 2) * uses_per_path
431: (8)                    if not should_do_optimization:
432: (12)                        return RendererBase.draw_path_collection(
433: (16)                            self, gc, master_transform, paths, all_transforms,
434: (16)                            offsets, offset_trans, facecolors, edgecolors,
435: (16)                            linewidths, linestyles, antialiaseds, urls,
436: (16)                            offset_position)
437: (8)                    path_codes = []
```

```
438: (8)                    for i, (path, transform) in enumerate(self._iter_collection_raw_paths(
439: (16)                           master_transform, paths, all_transforms)):
440: (12)                       name = 'p%d_%d' % (self._path_collection_id, i)
441: (12)                       path_bytes = self._convert_path(path, transform, simplify=False)
442: (12)                       self._pswriter.write(f"""\
443: (0)            /{name} {{
444: (0)            newpath
445: (0)            translate
446: (0)            {path_bytes}
447: (0)            }} bind def
448: (0)            """)
449: (12)                       path_codes.append(name)
450: (8)                    for xo, yo, path_id, gc0, rgbFace in self._iter_collection(
451: (16)                           gc, path_codes, offsets, offset_trans,
452: (16)                           facecolors, edgecolors, linewidths, linestyles,
453: (16)                           antialiaseds, urls, offset_position):
454: (12)                       ps = f"{xo:g} {yo:g} {path_id}"
455: (12)                       self._draw_ps(ps, gc0, rgbFace)
456: (8)                    self._path_collection_id += 1
457: (4)                @_log_if_debug_on
458: (4)                def draw_tex(self, gc, x, y, s, prop, angle, *, mtext=None):
459: (8)                    if self._is_transparent(gc.get_rgb()):
460: (12)                       return  # Special handling for fully transparent.
461: (8)                    if not hasattr(self, "psfrag"):
462: (12)                       self._logwarn_once(
463: (16)                           "The PS backend determines usetex status solely based on "
464: (16)                           "rcParams['text.usetex'] and does not support having "
465: (16)                           "usetex=True only for some elements; this element will thus "
466: (16)                           "be rendered as if usetex=False.")
467: (12)                       self.draw_text(gc, x, y, s, prop, angle, False, mtext)
468: (12)                       return
469: (8)                    w, h, bl = self.get_text_width_height_descent(s, prop, ismath="TeX")
470: (8)                    fontsize = prop.get_size_in_points()
471: (8)                    thetext = 'psmarker%d' % self.textcnt
472: (8)                    color = _nums_to_str(*gc.get_rgb()[:3], sep=',')
473: (8)                    fontcmd = {'sans-serif': r'{\sffamily %s}',
474: (19)                               'monospace': r'{\ttfamily %s}'}.get(
475: (23)                                   mpl.rcParams['font.family'][0], r'{\rmfamily %s}')
476: (8)                    s = fontcmd % s
477: (8)                    tex = r'\color[rgb]{%s} %s' % (color, s)
478: (8)                    rangle = np.radians(angle + 90)
479: (8)                    pos = _nums_to_str(x - bl * np.cos(rangle), y - bl * np.sin(rangle))
480: (8)                    self.psfrag.append(
481: (12)                       r'\psfrag{%s}[bl][bl][1][%f]{\fontsize{%f}{%f}%s}' % (
482: (16)                           thetext, angle, fontsize, fontsize*1.25, tex))
483: (8)                    self._pswriter.write(f"""\
484: (0)            gsave
485: (0)            {pos} moveto
486: (0)            ({thetext})
487: (0)            show
488: (0)            grestore
489: (0)            """)
490: (8)                    self.textcnt += 1
491: (4)                @_log_if_debug_on
492: (4)                def draw_text(self, gc, x, y, s, prop, angle, ismath=False, mtext=None):
493: (8)                    if self._is_transparent(gc.get_rgb()):
494: (12)                       return  # Special handling for fully transparent.
495: (8)                    if ismath == 'TeX':
496: (12)                       return self.draw_tex(gc, x, y, s, prop, angle)
497: (8)                    if ismath:
498: (12)                       return self.draw_mathtext(gc, x, y, s, prop, angle)
499: (8)                    stream = []  # list of (ps_name, x, char_name)
500: (8)                    if mpl.rcParams['ps.useafm']:
501: (12)                       font = self._get_font_afm(prop)
502: (12)                       ps_name = (font.postscript_name.encode("ascii", "replace")
503: (24)                                  .decode("ascii"))
504: (12)                       scale = 0.001 * prop.get_size_in_points()
505: (12)                       thisx = 0
506: (12)                       last_name = None  # kerns returns 0 for None.
```

```
507: (12)                        for c in s:
508: (16)                            name = uni2type1.get(ord(c), f"uni{ord(c):04X}")
509: (16)                            try:
510: (20)                                width = font.get_width_from_char_name(name)
511: (16)                            except KeyError:
512: (20)                                name = 'question'
513: (20)                                width = font.get_width_char('?')
514: (16)                            kern = font.get_kern_dist_from_name(last_name, name)
515: (16)                            last_name = name
516: (16)                            thisx += kern * scale
517: (16)                            stream.append((ps_name, thisx, name))
518: (16)                            thisx += width * scale
519: (8)                    else:
520: (12)                        font = self._get_font_ttf(prop)
521: (12)                        self._character_tracker.track(font, s)
522: (12)                        for item in _text_helpers.layout(s, font):
523: (16)                            ps_name = (item.ft_object.postscript_name
524: (27)                                       .encode("ascii", "replace").decode("ascii"))
525: (16)                            glyph_name = item.ft_object.get_glyph_name(item.glyph_idx)
526: (16)                            stream.append((ps_name, item.x, glyph_name))
527: (8)                    self.set_color(*gc.get_rgb())
528: (8)                    for ps_name, group in itertools. \
529: (16)                            groupby(stream, lambda entry: entry[0]):
530: (12)                        self.set_font(ps_name, prop.get_size_in_points(), False)
531: (12)                        thetext = "\n".join(f"{x:g} 0 m /{name:s} glyphshow"
532: (32)                                            for _, x, name in group)
533: (12)                        self._pswriter.write(f"""\
534: (0)            gsave
535: (0)            {self._get_clip_cmd(gc)}
536: (0)            {x:g} {y:g} translate
537: (0)            {angle:g} rotate
538: (0)            {thetext}
539: (0)            grestore
540: (0)            """)
541: (4)                @_log_if_debug_on
542: (4)                def draw_mathtext(self, gc, x, y, s, prop, angle):
543: (8)                    """Draw the math text using matplotlib.mathtext."""
544: (8)                    width, height, descent, glyphs, rects = \
545: (12)                       self._text2path.mathtext_parser.parse(s, 72, prop)
546: (8)                    self.set_color(*gc.get_rgb())
547: (8)                    self._pswriter.write(
548: (12)                       f"gsave\n"
549: (12)                       f"{x:g} {y:g} translate\n"
550: (12)                       f"{angle:g} rotate\n")
551: (8)                    lastfont = None
552: (8)                    for font, fontsize, num, ox, oy in glyphs:
553: (12)                       self._character_tracker.track_glyph(font, num)
554: (12)                       if (font.postscript_name, fontsize) != lastfont:
555: (16)                           lastfont = font.postscript_name, fontsize
556: (16)                           self._pswriter.write(
557: (20)                               f"/{font.postscript_name} {fontsize} selectfont\n")
558: (12)                       glyph_name = (
559: (16)                           font.get_name_char(chr(num)) if isinstance(font, AFM) else
560: (16)                           font.get_glyph_name(font.get_char_index(num)))
561: (12)                       self._pswriter.write(
562: (16)                           f"{ox:g} {oy:g} moveto\n"
563: (16)                           f"/{glyph_name} glyphshow\n")
564: (8)                    for ox, oy, w, h in rects:
565: (12)                       self._pswriter.write(f"{ox} {oy} {w} {h} rectfill\n")
566: (8)                    self._pswriter.write("grestore\n")
567: (4)                @_log_if_debug_on
568: (4)                def draw_gouraud_triangles(self, gc, points, colors, trans):
569: (8)                    assert len(points) == len(colors)
570: (8)                    if len(points) == 0:
571: (12)                       return
572: (8)                    assert points.ndim == 3
573: (8)                    assert points.shape[1] == 3
574: (8)                    assert points.shape[2] == 2
575: (8)                    assert colors.ndim == 3
```

```
576: (8)                    assert colors.shape[1] == 3
577: (8)                    assert colors.shape[2] == 4
578: (8)                    shape = points.shape
579: (8)                    flat_points = points.reshape((shape[0] * shape[1], 2))
580: (8)                    flat_points = trans.transform(flat_points)
581: (8)                    flat_colors = colors.reshape((shape[0] * shape[1], 4))
582: (8)                    points_min = np.min(flat_points, axis=0) - (1 << 12)
583: (8)                    points_max = np.max(flat_points, axis=0) + (1 << 12)
584: (8)                    factor = np.ceil((2 ** 32 - 1) / (points_max - points_min))
585: (8)                    xmin, ymin = points_min
586: (8)                    xmax, ymax = points_max
587: (8)                    data = np.empty(
588: (12)                       shape[0] * shape[1],
589: (12)                       dtype=[('flags', 'u1'), ('points', '2>u4'), ('colors', '3u1')])
590: (8)                    data['flags'] = 0
591: (8)                    data['points'] = (flat_points - points_min) * factor
592: (8)                    data['colors'] = flat_colors[:, :3] * 255.0
593: (8)                    hexdata = data.tobytes().hex("\n", -64)  # Linewrap to 128 chars.
594: (8)                    self._pswriter.write(f"""\
595: (0)            gsave
596: (0)            << /ShadingType 4
597: (3)               /ColorSpace [/DeviceRGB]
598: (3)               /BitsPerCoordinate 32
599: (3)               /BitsPerComponent 8
600: (3)               /BitsPerFlag 8
601: (3)               /AntiAlias true
602: (3)               /Decode [ {xmin:g} {xmax:g} {ymin:g} {ymax:g} 0 1 0 1 0 1 ]
603: (3)               /DataSource <
604: (0)            {hexdata}
605: (0)            >
606: (0)            >>
607: (0)            shfill
608: (0)            grestore
609: (0)            """)
610: (4)                def _draw_ps(self, ps, gc, rgbFace, *, fill=True, stroke=True):
611: (8)                    """
612: (8)                    Emit the PostScript snippet *ps* with all the attributes from *gc*
613: (8)                    applied.  *ps* must consist of PostScript commands to construct a
path.
614: (8)                    The *fill* and/or *stroke* kwargs can be set to False if the *ps*
615: (8)                    string already includes filling and/or stroking, in which case
616: (8)                    `_draw_ps` is just supplying properties and clipping.
617: (8)                    """
618: (8)                    write = self._pswriter.write
619: (8)                    mightstroke = (gc.get_linewidth() > 0
620: (23)                                  and not self._is_transparent(gc.get_rgb()))
621: (8)                    if not mightstroke:
622: (12)                       stroke = False
623: (8)                    if self._is_transparent(rgbFace):
624: (12)                       fill = False
625: (8)                    hatch = gc.get_hatch()
626: (8)                    if mightstroke:
627: (12)                       self.set_linewidth(gc.get_linewidth())
628: (12)                       self.set_linejoin(gc.get_joinstyle())
629: (12)                       self.set_linecap(gc.get_capstyle())
630: (12)                       self.set_linedash(*gc.get_dashes())
631: (8)                    if mightstroke or hatch:
632: (12)                       self.set_color(*gc.get_rgb()[:3])
633: (8)                    write('gsave\n')
634: (8)                    write(self._get_clip_cmd(gc))
635: (8)                    write(ps.strip())
636: (8)                    write("\n")
637: (8)                    if fill:
638: (12)                       if stroke or hatch:
639: (16)                           write("gsave\n")
640: (12)                       self.set_color(*rgbFace[:3], store=False)
641: (12)                       write("fill\n")
642: (12)                       if stroke or hatch:
643: (16)                           write("grestore\n")
```

```
644: (8)                    if hatch:
645: (12)                       hatch_name = self.create_hatch(hatch)
646: (12)                       write("gsave\n")
647: (12)                       write(_nums_to_str(*gc.get_hatch_color()[:3]))
648: (12)                       write(f" {hatch_name} setpattern fill grestore\n")
649: (8)                    if stroke:
650: (12)                       write("stroke\n")
651: (8)                    write("grestore\n")
652: (0)            class _Orientation(Enum):
653: (4)                portrait, landscape = range(2)
654: (4)                def swap_if_landscape(self, shape):
655: (8)                    return shape[::-1] if self.name == "landscape" else shape
656: (0)            class FigureCanvasPS(FigureCanvasBase):
657: (4)                fixed_dpi = 72
658: (4)                filetypes = {'ps': 'Postscript',
659: (17)                            'eps': 'Encapsulated Postscript'}
660: (4)                def get_default_filetype(self):
661: (8)                    return 'ps'
662: (4)                def _print_ps(
663: (12)                       self, fmt, outfile, *,
664: (12)                       metadata=None, papertype=None, orientation='portrait',
665: (12)                       bbox_inches_restore=None, **kwargs):
666: (8)                    dpi = self.figure.dpi
667: (8)                    self.figure.dpi = 72  # Override the dpi kwarg
668: (8)                    dsc_comments = {}
669: (8)                    if isinstance(outfile, (str, os.PathLike)):
670: (12)                       filename = pathlib.Path(outfile).name
671: (12)                       dsc_comments["Title"] = \
672: (16)                           filename.encode("ascii", "replace").decode("ascii")
673: (8)                    dsc_comments["Creator"] = (metadata or {}).get(
674: (12)                       "Creator",
675: (12)                       f"Matplotlib v{mpl.__version__}, https://matplotlib.org/")
676: (8)                    source_date_epoch = os.getenv("SOURCE_DATE_EPOCH")
677: (8)                    dsc_comments["CreationDate"] = (
678: (12)                       datetime.datetime.fromtimestamp(
679: (16)                           int(source_date_epoch),
680: (16)                           datetime.timezone.utc).strftime("%a %b %d %H:%M:%S %Y")
681: (12)                       if source_date_epoch
682: (12)                       else time.ctime())
683: (8)                    dsc_comments = "\n".join(
684: (12)                       f"%%{k}: {v}" for k, v in dsc_comments.items())
685: (8)                    if papertype is None:
686: (12)                       papertype = mpl.rcParams['ps.papersize']
687: (8)                    papertype = papertype.lower()
688: (8)                    _api.check_in_list(['figure', 'auto', *papersize],
papertype=papertype)
689: (8)                    orientation = _api.check_getitem(
690: (12)                       _Orientation, orientation=orientation.lower())
691: (8)                    printer = (self._print_figure_tex
692: (19)                              if mpl.rcParams['text.usetex'] else
693: (19)                              self._print_figure)
694: (8)                    printer(fmt, outfile, dpi=dpi, dsc_comments=dsc_comments,
695: (16)                           orientation=orientation, papertype=papertype,
696: (16)                           bbox_inches_restore=bbox_inches_restore, **kwargs)
697: (4)                def _print_figure(
698: (12)                       self, fmt, outfile, *,
699: (12)                       dpi, dsc_comments, orientation, papertype,
700: (12)                       bbox_inches_restore=None):
701: (8)                    """
702: (8)                    Render the figure to a filesystem path or a file-like object.
703: (8)                    Parameters are as for `.print_figure`, except that *dsc_comments* is a
704: (8)                    string containing Document Structuring Convention comments,
705: (8)                    generated from the *metadata* parameter to `.print_figure`.
706: (8)                    """
707: (8)                    is_eps = fmt == 'eps'
708: (8)                    if not (isinstance(outfile, (str, os.PathLike))
709: (16)                           or is_writable_file_like(outfile)):
710: (12)                       raise ValueError("outfile must be a path or a file-like object")
711: (8)                    width, height = self.figure.get_size_inches()
```

```
(8)                     if papertype == 'auto':
(12)                        papertype = _get_papertype(*orientation.swap_if_landscape((width,
height)))
(8)                     if is_eps or papertype == 'figure':
(12)                        paper_width, paper_height = width, height
(8)                     else:
(12)                        paper_width, paper_height = orientation.swap_if_landscape(
(16)                            papersize[papertype])
(8)                     xo = 72 * 0.5 * (paper_width - width)
(8)                     yo = 72 * 0.5 * (paper_height - height)
(8)                     llx = xo
(8)                     lly = yo
(8)                     urx = llx + self.figure.bbox.width
(8)                     ury = lly + self.figure.bbox.height
(8)                     rotation = 0
(8)                     if orientation is _Orientation.landscape:
(12)                        llx, lly, urx, ury = lly, llx, ury, urx
(12)                        xo, yo = 72 * paper_height - yo, xo
(12)                        rotation = 90
(8)                     bbox = (llx, lly, urx, ury)
(8)                     self._pswriter = StringIO()
(8)                     ps_renderer = RendererPS(width, height, self._pswriter, imagedpi=dpi)
(8)                     renderer = MixedModeRenderer(
(12)                        self.figure, width, height, dpi, ps_renderer,
(12)                        bbox_inches_restore=bbox_inches_restore)
(8)                     self.figure.draw(renderer)
(8)                     def print_figure_impl(fh):
(12)                        if is_eps:
(16)                            print("%!PS-Adobe-3.0 EPSF-3.0", file=fh)
(12)                        else:
(16)                            print("%!PS-Adobe-3.0", file=fh)
(16)                            if papertype != 'figure':
(20)                                print(f"%%DocumentPaperSizes: {papertype}", file=fh)
(16)                            print("%%Pages: 1", file=fh)
(12)                        print(f"%%LanguageLevel: 3\n"
(18)                              f"{dsc_comments}\n"
(18)                              f"%%Orientation: {orientation.name}\n"
(18)                              f"{_get_bbox_header(bbox)}\n"
(18)                              f"%%EndComments\n",
(18)                              end="", file=fh)
(12)                        Ndict = len(_psDefs)
(12)                        print("%%BeginProlog", file=fh)
(12)                        if not mpl.rcParams['ps.useafm']:
(16)                            Ndict += len(ps_renderer._character_tracker.used)
(12)                        print("/mpldict %d dict def" % Ndict, file=fh)
(12)                        print("mpldict begin", file=fh)
(12)                        print("\n".join(_psDefs), file=fh)
(12)                        if not mpl.rcParams['ps.useafm']:
(16)                            for font_path, chars \
(24)                                    in ps_renderer._character_tracker.used.items():
(20)                                if not chars:
(24)                                    continue
(20)                                fonttype = mpl.rcParams['ps.fonttype']
(20)                                if len(chars) > 255:
(24)                                    fonttype = 42
(20)                                fh.flush()
(20)                                if fonttype == 3:
(24)                                    fh.write(_font_to_ps_type3(font_path, chars))
(20)                                else:  # Type 42 only.
(24)                                    _font_to_ps_type42(font_path, chars, fh)
(12)                        print("end", file=fh)
(12)                        print("%%EndProlog", file=fh)
(12)                        if not is_eps:
(16)                            print("%%Page: 1 1", file=fh)
(12)                        print("mpldict begin", file=fh)
(12)                        print("%s translate" % _nums_to_str(xo, yo), file=fh)
(12)                        if rotation:
(16)                            print("%d rotate" % rotation, file=fh)
(12)                        print(f"0 0 {_nums_to_str(width*72, height*72)} rectclip",
```

```
file=fh)
780: (12)                        print(self._pswriter.getvalue(), file=fh)
781: (12)                        print("end", file=fh)
782: (12)                        print("showpage", file=fh)
783: (12)                        if not is_eps:
784: (16)                            print("%%EOF", file=fh)
785: (12)                        fh.flush()
786: (8)                    if mpl.rcParams['ps.usedistiller']:
787: (12)                        with TemporaryDirectory() as tmpdir:
788: (16)                            tmpfile = os.path.join(tmpdir, "tmp.ps")
789: (16)                            with open(tmpfile, 'w', encoding='latin-1') as fh:
790: (20)                                print_figure_impl(fh)
791: (16)                            if mpl.rcParams['ps.usedistiller'] == 'ghostscript':
792: (20)                                _try_distill(gs_distill,
793: (33)                                             tmpfile, is_eps, ptype=papertype, bbox=bbox)
794: (16)                            elif mpl.rcParams['ps.usedistiller'] == 'xpdf':
795: (20)                                _try_distill(xpdf_distill,
796: (33)                                             tmpfile, is_eps, ptype=papertype, bbox=bbox)
797: (16)                            _move_path_to_path_or_stream(tmpfile, outfile)
798: (8)                    else:  # Write directly to outfile.
799: (12)                        with cbook.open_file_cm(outfile, "w", encoding="latin-1") as file:
800: (16)                            if not file_requires_unicode(file):
801: (20)                                file = codecs.getwriter("latin-1")(file)
802: (16)                            print_figure_impl(file)
803: (4)                def _print_figure_tex(
804: (12)                        self, fmt, outfile, *,
805: (12)                        dpi, dsc_comments, orientation, papertype,
806: (12)                        bbox_inches_restore=None):
807: (8)                    """
808: (8)                    If :rc:`text.usetex` is True, a temporary pair of tex/eps files
809: (8)                    are created to allow tex to manage the text layout via the PSFrags
810: (8)                    package. These files are processed to yield the final ps or eps file.
811: (8)                    The rest of the behavior is as for `._print_figure`.
812: (8)                    """
813: (8)                    is_eps = fmt == 'eps'
814: (8)                    width, height = self.figure.get_size_inches()
815: (8)                    xo = 0
816: (8)                    yo = 0
817: (8)                    llx = xo
818: (8)                    lly = yo
819: (8)                    urx = llx + self.figure.bbox.width
820: (8)                    ury = lly + self.figure.bbox.height
821: (8)                    bbox = (llx, lly, urx, ury)
822: (8)                    self._pswriter = StringIO()
823: (8)                    ps_renderer = RendererPS(width, height, self._pswriter, imagedpi=dpi)
824: (8)                    renderer = MixedModeRenderer(self.figure,
825: (37)                                                 width, height, dpi, ps_renderer,
826: (37)                                                 bbox_inches_restore=bbox_inches_restore)
827: (8)                    self.figure.draw(renderer)
828: (8)                    with TemporaryDirectory() as tmpdir:
829: (12)                        tmppath = pathlib.Path(tmpdir, "tmp.ps")
830: (12)                        tmppath.write_text(
831: (16)                            f"""\
832: (0)            %!PS-Adobe-3.0 EPSF-3.0
833: (0)            %%LanguageLevel: 3
834: (0)            {dsc_comments}
835: (0)            {_get_bbox_header(bbox)}
836: (0)            %%EndComments
837: (0)            %%BeginProlog
838: (0)            /mpldict {len(_psDefs)} dict def
839: (0)            mpldict begin
840: (0)            {"".join(_psDefs)}
841: (0)            end
842: (0)            %%EndProlog
843: (0)            mpldict begin
844: (0)            {_nums_to_str(xo, yo)} translate
845: (0)            0 0 {_nums_to_str(width*72, height*72)} rectclip
846: (0)            {self._pswriter.getvalue()}
847: (0)            end
```

```
848: (0)            showpage
849: (0)            """,
850: (16)                           encoding="latin-1")
851: (12)                       if orientation is _Orientation.landscape:  # now, ready to rotate
852: (16)                           width, height = height, width
853: (16)                           bbox = (lly, llx, ury, urx)
854: (12)                       if is_eps or papertype == 'figure':
855: (16)                           paper_width, paper_height = orientation.swap_if_landscape(
856: (20)                               self.figure.get_size_inches())
857: (12)                       else:
858: (16)                           if papertype == 'auto':
859: (20)                               papertype = _get_papertype(width, height)
860: (16)                           paper_width, paper_height = papersize[papertype]
861: (12)                       psfrag_rotated = _convert_psfrags(
862: (16)                           tmppath, ps_renderer.psfrag, paper_width, paper_height,
863: (16)                           orientation.name)
864: (12)                       if (mpl.rcParams['ps.usedistiller'] == 'ghostscript'
865: (20)                               or mpl.rcParams['text.usetex']):
866: (16)                           _try_distill(gs_distill,
867: (29)                                        tmppath, is_eps, ptype=papertype, bbox=bbox,
868: (29)                                        rotated=psfrag_rotated)
869: (12)                       elif mpl.rcParams['ps.usedistiller'] == 'xpdf':
870: (16)                           _try_distill(xpdf_distill,
871: (29)                                        tmppath, is_eps, ptype=papertype, bbox=bbox,
872: (29)                                        rotated=psfrag_rotated)
873: (12)                       _move_path_to_path_or_stream(tmppath, outfile)
874: (4)                print_ps = functools.partialmethod(_print_ps, "ps")
875: (4)                print_eps = functools.partialmethod(_print_ps, "eps")
876: (4)                def draw(self):
877: (8)                    self.figure.draw_without_rendering()
878: (8)                    return super().draw()
879: (0)            def _convert_psfrags(tmppath, psfrags, paper_width, paper_height,
orientation):
880: (4)                """
881: (4)                When we want to use the LaTeX backend with postscript, we write PSFrag
tags
882: (4)                to a temporary postscript file, each one marking a position for LaTeX to
883: (4)                render some text. convert_psfrags generates a LaTeX document containing
the
884: (4)                commands to convert those tags to text. LaTeX/dvips produces the
postscript
885: (4)                file that includes the actual text.
886: (4)                """
887: (4)                with mpl.rc_context({
888: (12)                       "text.latex.preamble":
889: (12)                       mpl.rcParams["text.latex.preamble"] +
890: (12)                       mpl.texmanager._usepackage_if_not_loaded("color") +
891: (12)                       mpl.texmanager._usepackage_if_not_loaded("graphicx") +
892: (12)                       mpl.texmanager._usepackage_if_not_loaded("psfrag") +
893: (12)                       r"\geometry{papersize={%(width)sin,%(height)sin},margin=0in}"
894: (12)                       % {"width": paper_width, "height": paper_height}
895: (4)                ‍‍‍
```

```
913: (8)                    shutil.move(psfile, tmppath)
914: (4)                with open(tmppath) as fh:
915: (8)                    psfrag_rotated = "Landscape" in fh.read(1000)
916: (4)                return psfrag_rotated
917: (0)            def _try_distill(func, tmppath, *args, **kwargs):
918: (4)                try:
919: (8)                    func(str(tmppath), *args, **kwargs)
920: (4)                except mpl.ExecutableNotFoundError as exc:
921: (8)                    _log.warning("%s.  Distillation step skipped.", exc)
922: (0)            def gs_distill(tmpfile, eps=False, ptype='letter', bbox=None, rotated=False):
923: (4)                """
924: (4)                Use ghostscript's pswrite or epswrite device to distill a file.
925: (4)                This yields smaller files without illegal encapsulated postscript
926: (4)                operators. The output is low-level, converting text to outlines.
927: (4)                """
928: (4)                if eps:
929: (8)                    paper_option = ["-dEPSCrop"]
930: (4)                elif ptype == "figure":
931: (8)                    paper_option = [f"-dDEVICEWIDTHPOINTS={bbox[2]}",
932: (24)                                    f"-dDEVICEHEIGHTPOINTS={bbox[3]}"]
933: (4)                else:
934: (8)                    paper_option = [f"-sPAPERSIZE={ptype}"]
935: (4)                psfile = tmpfile + '.ps'
936: (4)                dpi = mpl.rcParams['ps.distiller.res']
937: (4)                cbook._check_and_log_subprocess(
938: (8)                    [mpl._get_executable_info("gs").executable,
939: (9)                     "-dBATCH", "-dNOPAUSE", "-r%d" % dpi, "-sDEVICE=ps2write",
940: (9)                     *paper_option, f"-sOutputFile={psfile}", tmpfile],
941: (8)                    _log)
942: (4)                os.remove(tmpfile)
943: (4)                shutil.move(psfile, tmpfile)
944: (4)                if eps:
945: (8)                    pstoeps(tmpfile, bbox, rotated=rotated)
946: (0)            def xpdf_distill(tmpfile, eps=False, ptype='letter', bbox=None,
rotated=False):
947: (4)                """
948: (4)                Use ghostscript's ps2pdf and xpdf's/poppler's pdftops to distill a file.
949: (4)                This yields smaller files without illegal encapsulated postscript
950: (4)                operators. This distiller is preferred, generating high-level postscript
951: (4)                output that treats text as text.
952: (4)                """
953: (4)                mpl._get_executable_info("gs")  # Effectively checks for ps2pdf.
954: (4)                mpl._get_executable_info("pdftops")
955: (4)                if eps:
956: (8)                    paper_option = ["-dEPSCrop"]
957: (4)                elif ptype == "figure":
958: (8)                    paper_option = [f"-dDEVICEWIDTHPOINTS#{bbox[2]}",
959: (24)                                    f"-dDEVICEHEIGHTPOINTS#{bbox[3]}"]
960: (4)                else:
961: (8)                    paper_option = [f"-sPAPERSIZE#{ptype}"]
962: (4)                with TemporaryDirectory() as tmpdir:
963: (8)                    tmppdf = pathlib.Path(tmpdir, "tmp.pdf")
964: (8)                    tmpps = pathlib.Path(tmpdir, "tmp.ps")
965: (8)                    cbook._check_and_log_subprocess(
966: (12)                       ["ps2pdf",
967: (13)                        "-dAutoFilterColorImages#false",
968: (13)                        "-dAutoFilterGrayImages#false",
969: (13)                        "-sAutoRotatePages#None",
970: (13)                        "-sGrayImageFilter#FlateEncode",
971: (13)                        "-sColorImageFilter#FlateEncode",
972: (13)                        *paper_option,
973: (13)                        tmpfile, tmppdf], _log)
974: (8)                    cbook._check_and_log_subprocess(
975: (12)                       ["pdftops", "-paper", "match", "-level3", tmppdf, tmpps], _log)
976: (8)                    shutil.move(tmpps, tmpfile)
977: (4)                if eps:
978: (8)                    pstoeps(tmpfile)
979: (0)            @_api.deprecated("3.9")
980: (0)            def get_bbox_header(lbrt, rotated=False):
```

```
981: (4)                """
982: (4)                Return a postscript header string for the given bbox lbrt=(l, b, r, t).
983: (4)                Optionally, return rotate command.
984: (4)                """
985: (4)                return _get_bbox_header(lbrt), (_get_rotate_command(lbrt) if rotated else
"")
986: (0)            def _get_bbox_header(lbrt):
987: (4)                """Return a PostScript header string for bounding box *lbrt*=(l, b, r,
t)."""
988: (4)                l, b, r, t = lbrt
989: (4)                return (f"%%BoundingBox: {int(l)} {int(b)} {math.ceil(r)}
{math.ceil(t)}\n"
990: (12)                       f"%%HiResBoundingBox: {l:.6f} {b:.6f} {r:.6f} {t:.6f}")
991: (0)            def _get_rotate_command(lbrt):
992: (4)                """Return a PostScript 90° rotation command for bounding box *lbrt*=(l, b,
r, t)."""
993: (4)                l, b, r, t = lbrt
994: (4)                return f"{l+r:.2f} {0:.2f} translate\n90 rotate"
995: (0)            def pstoeps(tmpfile, bbox=None, rotated=False):
996: (4)                """
997: (4)                Convert the postscript to encapsulated postscript.  The bbox of
998: (4)                the eps file will be replaced with the given *bbox* argument. If
999: (4)                None, original bbox will be used.
1000: (4)               """
1001: (4)               epsfile = tmpfile + '.eps'
1002: (4)               with open(epsfile, 'wb') as epsh, open(tmpfile, 'rb') as tmph:
1003: (8)                   write = epsh.write
1004: (8)                   for line in tmph:
1005: (12)                      if line.startswith(b'%!PS'):
1006: (16)                          write(b"%!PS-Adobe-3.0 EPSF-3.0\n")
1007: (16)                          if bbox:
1008: (20)                              write(_get_bbox_header(bbox).encode('ascii') + b'\n')
1009: (12)                      elif line.startswith(b'%%EndComments'):
1010: (16)                          write(line)
1011: (16)                          write(b'%%BeginProlog\n'
1012: (22)                                b'save\n'
1013: (22)                                b'countdictstack\n'
1014: (22)                                b'mark\n'
1015: (22)                                b'newpath\n'
1016: (22)                                b'/showpage {} def\n'
1017: (22)                                b'/setpagedevice {pop} def\n'
1018: (22)                                b'%%EndProlog\n'
1019: (22)                                b'%%Page 1 1\n')
1020: (16)                          if rotated:  # The output eps file need to be rotated.
1021: (20)                              write(_get_rotate_command(bbox).encode('ascii') + b'\n')
1022: (16)                          break
1023: (12)                      elif bbox and line.startswith((b'%%Bound', b'%%HiResBound',
1024: (43)                                                     b'%%DocumentMedia', b'%%Pages')):
1025: (16)                          pass
1026: (12)                      else:
1027: (16)                          write(line)
1028: (8)                   for line in tmph:
1029: (12)                      if line.startswith(b'%%EOF'):
1030: (16)                          write(b'cleartomark\n'
1031: (22)                                b'countdictstack\n'
1032: (22)                                b'exch sub { end } repeat\n'
1033: (22)                                b'restore\n'
1034: (22)                                b'showpage\n'
1035: (22)                                b'%%EOF\n')
1036: (12)                      elif line.startswith(b'%%PageBoundingBox'):
1037: (16)                          pass
1038: (12)                      else:
1039: (16)                          write(line)
1040: (4)               os.remove(tmpfile)
1041: (4)               shutil.move(epsfile, tmpfile)
1042: (0)           FigureManagerPS = FigureManagerBase
1043: (0)           _psDefs = [
1044: (4)               "/_d { bind def } bind def",
1045: (4)               "/m { moveto } _d",
```

```
1046: (4)               "/l { lineto } _d",
1047: (4)               "/r { rlineto } _d",
1048: (4)               "/c { curveto } _d",
1049: (4)               "/cl { closepath } _d",
1050: (4)               "/ce { closepath eofill } _d",
1051: (4)               "/sc { setcachedevice } _d",
1052: (0)           ]
1053: (0)           @_Backend.export
1054: (0)           class _BackendPS(_Backend):
1055: (4)               backend_version = 'Level II'
1056: (4)               FigureCanvas = FigureCanvasPS
```

File 95 - backend_qt.py:

```
1: (0)              import functools
2: (0)              import os
3: (0)              import sys
4: (0)              import traceback
5: (0)              import matplotlib as mpl
6: (0)              from matplotlib import _api, backend_tools, cbook
7: (0)              from matplotlib._pylab_helpers import Gcf
8: (0)              from matplotlib.backend_bases import (
9: (4)                  _Backend, FigureCanvasBase, FigureManagerBase, NavigationToolbar2,
10: (4)                 TimerBase, cursors, ToolContainerBase, MouseButton,
11: (4)                 CloseEvent, KeyEvent, LocationEvent, MouseEvent, ResizeEvent,
12: (4)                 _allow_interrupt)
13: (0)             import matplotlib.backends.qt_editor.figureoptions as figureoptions
14: (0)             from . import qt_compat
15: (0)             from .qt_compat import (
16: (4)                 QtCore, QtGui, QtWidgets, __version__, QT_API, _to_int, _isdeleted)
17: (0)             SPECIAL_KEYS = {
18: (4)                 _to_int(getattr(QtCore.Qt.Key, k)): v for k, v in [
19: (8)                     ("Key_Escape", "escape"),
20: (8)                     ("Key_Tab", "tab"),
21: (8)                     ("Key_Backspace", "backspace"),
22: (8)                     ("Key_Return", "enter"),
23: (8)                     ("Key_Enter", "enter"),
24: (8)                     ("Key_Insert", "insert"),
25: (8)                     ("Key_Delete", "delete"),
26: (8)                     ("Key_Pause", "pause"),
27: (8)                     ("Key_SysReq", "sysreq"),
28: (8)                     ("Key_Clear", "clear"),
29: (8)                     ("Key_Home", "home"),
30: (8)                     ("Key_End", "end"),
31: (8)                     ("Key_Left", "left"),
32: (8)                     ("Key_Up", "up"),
33: (8)                     ("Key_Right", "right"),
34: (8)                     ("Key_Down", "down"),
35: (8)                     ("Key_PageUp", "pageup"),
36: (8)                     ("Key_PageDown", "pagedown"),
37: (8)                     ("Key_Shift", "shift"),
38: (8)                     ("Key_Control", "control" if sys.platform != "darwin" else "cmd"),
39: (8)                     ("Key_Meta", "meta" if sys.platform != "darwin" else "control"),
40: (8)                     ("Key_Alt", "alt"),
41: (8)                     ("Key_CapsLock", "caps_lock"),
42: (8)                     ("Key_F1", "f1"),
43: (8)                     ("Key_F2", "f2"),
44: (8)                     ("Key_F3", "f3"),
45: (8)                     ("Key_F4", "f4"),
46: (8)                     ("Key_F5", "f5"),
47: (8)                     ("Key_F6", "f6"),
48: (8)                     ("Key_F7", "f7"),
49: (8)                     ("Key_F8", "f8"),
50: (8)                     ("Key_F9", "f9"),
51: (8)                     ("Key_F10", "f10"),
52: (8)                     ("Key_F10", "f11"),
53: (8)                     ("Key_F12", "f12"),
```

```
 54: (8)                    ("Key_Super_L", "super"),
 55: (8)                    ("Key_Super_R", "super"),
 56: (4)                ]
 57: (0)            }
 58: (0)            _MODIFIER_KEYS = [
 59: (4)                (_to_int(getattr(QtCore.Qt.KeyboardModifier, mod)),
 60: (5)                 _to_int(getattr(QtCore.Qt.Key, key)))
 61: (4)                for mod, key in [
 62: (8)                    ("ControlModifier", "Key_Control"),
 63: (8)                    ("AltModifier", "Key_Alt"),
 64: (8)                    ("ShiftModifier", "Key_Shift"),
 65: (8)                    ("MetaModifier", "Key_Meta"),
 66: (4)                ]
 67: (0)            ]
 68: (0)            cursord = {
 69: (4)                k: getattr(QtCore.Qt.CursorShape, v) for k, v in [
 70: (8)                    (cursors.MOVE, "SizeAllCursor"),
 71: (8)                    (cursors.HAND, "PointingHandCursor"),
 72: (8)                    (cursors.POINTER, "ArrowCursor"),
 73: (8)                    (cursors.SELECT_REGION, "CrossCursor"),
 74: (8)                    (cursors.WAIT, "WaitCursor"),
 75: (8)                    (cursors.RESIZE_HORIZONTAL, "SizeHorCursor"),
 76: (8)                    (cursors.RESIZE_VERTICAL, "SizeVerCursor"),
 77: (4)                ]
 78: (0)            }
 79: (0)            @functools.lru_cache(1)
 80: (0)            def _create_qApp():
 81: (4)                app = QtWidgets.QApplication.instance()
 82: (4)                if app is None:
 83: (8)                    if not mpl._c_internal_utils.display_is_valid():
 84: (12)                       raise RuntimeError('Invalid DISPLAY variable')
 85: (8)                    if QT_API in {'PyQt6', 'PySide6'}:
 86: (12)                       other_bindings = ('PyQt5', 'PySide2')
 87: (12)                       qt_version = 6
 88: (8)                    elif QT_API in {'PyQt5', 'PySide2'}:
 89: (12)                       other_bindings = ('PyQt6', 'PySide6')
 90: (12)                       qt_version = 5
 91: (8)                    else:
 92: (12)                       raise RuntimeError("Should never be here")
 93: (8)                    for binding in other_bindings:
 94: (12)                       mod = sys.modules.get(f'{binding}.QtWidgets')
 95: (12)                       if mod is not None and mod.QApplication.instance() is not None:
 96: (16)                           other_core = sys.modules.get(f'{binding}.QtCore')
 97: (16)                           _api.warn_external(
 98: (20)                               f'Matplotlib is using {QT_API} which wraps '
 99: (20)                               f'{QtCore.qVersion()} however an instantiated '
100: (20)                               f'QApplication from {binding} which wraps '
101: (20)                               f'{other_core.qVersion()} exists.  Mixing Qt major '
102: (20)                               'versions may not work as expected.'
103: (16)                           )
104: (16)                           break
105: (8)                    if qt_version == 5:
106: (12)                       try:
107: (16)
QtWidgets.QApplication.setAttribute(QtCore.Qt.AA_EnableHighDpiScaling)
108: (12)                       except AttributeError:  # Only for Qt>=5.6, <6.
109: (16)                           pass
110: (8)                    try:
111: (12)                       QtWidgets.QApplication.setHighDpiScaleFactorRoundingPolicy(
112: (16)                           QtCore.Qt.HighDpiScaleFactorRoundingPolicy.PassThrough)
113: (8)                    except AttributeError:  # Only for Qt>=5.14.
114: (12)                       pass
115: (8)                    app = QtWidgets.QApplication(["matplotlib"])
116: (8)                    if sys.platform == "darwin":
117: (12)                       image = str(cbook._get_data_path('images/matplotlib.svg'))
118: (12)                       icon = QtGui.QIcon(image)
119: (12)                       app.setWindowIcon(icon)
120: (8)                    app.setQuitOnLastWindowClosed(True)
121: (8)                    cbook._setup_new_guiapp()
```

```
122: (8)                    if qt_version == 5:
123: (12)                       app.setAttribute(QtCore.Qt.AA_UseHighDpiPixmaps)
124: (4)                return app
125: (0)            def _allow_interrupt_qt(qapp_or_eventloop):
126: (4)                """A context manager that allows terminating a plot by sending a
SIGINT."""
127: (4)                def prepare_notifier(rsock):
128: (8)                    sn = QtCore.QSocketNotifier(rsock.fileno(),
QtCore.QSocketNotifier.Type.Read)
129: (8)                    @sn.activated.connect
130: (8)                    def _may_clear_sock():
131: (12)                       try:
132: (16)                           rsock.recv(1)
133: (12)                       except BlockingIOError:
134: (16)                           pass
135: (8)                    return sn  # Actually keep the notifier alive.
136: (4)                def handle_sigint():
137: (8)                    if hasattr(qapp_or_eventloop, 'closeAllWindows'):
138: (12)                       qapp_or_eventloop.closeAllWindows()
139: (8)                    qapp_or_eventloop.quit()
140: (4)                return _allow_interrupt(prepare_notifier, handle_sigint)
141: (0)            class TimerQT(TimerBase):
142: (4)                """Subclass of `.TimerBase` using QTimer events."""
143: (4)                def __init__(self, *args, **kwargs):
144: (8)                    self._timer = QtCore.QTimer()
145: (8)                    self._timer.timeout.connect(self._on_timer)
146: (8)                    super().__init__(*args, **kwargs)
147: (4)                def __del__(self):
148: (8)                    if not _isdeleted(self._timer):
149: (12)                       self._timer_stop()
150: (4)                def _timer_set_single_shot(self):
151: (8)                    self._timer.setSingleShot(self._single)
152: (4)                def _timer_set_interval(self):
153: (8)                    self._timer.setInterval(self._interval)
154: (4)                def _timer_start(self):
155: (8)                    self._timer.start()
156: (4)                def _timer_stop(self):
157: (8)                    self._timer.stop()
158: (0)            class FigureCanvasQT(FigureCanvasBase, QtWidgets.QWidget):
159: (4)                required_interactive_framework = "qt"
160: (4)                _timer_cls = TimerQT
161: (4)                manager_class = _api.classproperty(lambda cls: FigureManagerQT)
162: (4)                buttond = {
163: (8)                    getattr(QtCore.Qt.MouseButton, k): v for k, v in [
164: (12)                       ("LeftButton", MouseButton.LEFT),
165: (12)                       ("RightButton", MouseButton.RIGHT),
166: (12)                       ("MiddleButton", MouseButton.MIDDLE),
167: (12)                       ("XButton1", MouseButton.BACK),
168: (12)                       ("XButton2", MouseButton.FORWARD),
169: (8)                    ]
170: (4)                }
171: (4)                def __init__(self, figure=None):
172: (8)                    _create_qApp()
173: (8)                    super().__init__(figure=figure)
174: (8)                    self._draw_pending = False
175: (8)                    self._is_drawing = False
176: (8)                    self._draw_rect_callback = lambda painter: None
177: (8)                    self._in_resize_event = False
178: (8)                    self.setAttribute(QtCore.Qt.WidgetAttribute.WA_OpaquePaintEvent)
179: (8)                    self.setMouseTracking(True)
180: (8)                    self.resize(*self.get_width_height())
181: (8)                    palette = QtGui.QPalette(QtGui.QColor("white"))
182: (8)                    self.setPalette(palette)
183: (4)                def _update_pixel_ratio(self):
184: (8)                    if self._set_device_pixel_ratio(
185: (16)                           self.devicePixelRatioF() or 1):  # rarely, devicePixelRatioF=0
186: (12)                       event = QtGui.QResizeEvent(self.size(), self.size())
187: (12)                       self.resizeEvent(event)
188: (4)                def _update_screen(self, screen):
```

```
189: (8)                    self._update_pixel_ratio()
190: (8)                    if screen is not None:
191: (12)
screen.physicalDotsPerInchChanged.connect(self._update_pixel_ratio)
192: (12)                        screen.logicalDotsPerInchChanged.connect(self._update_pixel_ratio)
193: (4)                def showEvent(self, event):
194: (8)                    window = self.window().windowHandle()
195: (8)                    window.screenChanged.connect(self._update_screen)
196: (8)                    self._update_screen(window.screen())
197: (4)                def set_cursor(self, cursor):
198: (8)                    self.setCursor(_api.check_getitem(cursord, cursor=cursor))
199: (4)                def mouseEventCoords(self, pos=None):
200: (8)                    """
201: (8)                    Calculate mouse coordinates in physical pixels.
202: (8)                    Qt uses logical pixels, but the figure is scaled to physical
203: (8)                    pixels for rendering.  Transform to physical pixels so that
204: (8)                    all of the down-stream transforms work as expected.
205: (8)                    Also, the origin is different and needs to be corrected.
206: (8)                    """
207: (8)                    if pos is None:
208: (12)                       pos = self.mapFromGlobal(QtGui.QCursor.pos())
209: (8)                    elif hasattr(pos, "position"):  # qt6 QtGui.QEvent
210: (12)                       pos = pos.position()
211: (8)                    elif hasattr(pos, "pos"):  # qt5 QtCore.QEvent
212: (12)                       pos = pos.pos()
213: (8)                    x = pos.x()
214: (8)                    y = self.figure.bbox.height / self.device_pixel_ratio - pos.y()
215: (8)                    return x * self.device_pixel_ratio, y * self.device_pixel_ratio
216: (4)                def enterEvent(self, event):
217: (8)                    mods = QtWidgets.QApplication.instance().queryKeyboardModifiers()
218: (8)                    if self.figure is None:
219: (12)                       return
220: (8)                    LocationEvent("figure_enter_event", self,
221: (22)                                  *self.mouseEventCoords(event),
222: (22)                                  modifiers=self._mpl_modifiers(mods),
223: (22)                                  guiEvent=event)._process()
224: (4)                def leaveEvent(self, event):
225: (8)                    QtWidgets.QApplication.restoreOverrideCursor()
226: (8)                    if self.figure is None:
227: (12)                       return
228: (8)                    LocationEvent("figure_leave_event", self,
229: (22)                                  *self.mouseEventCoords(),
230: (22)                                  modifiers=self._mpl_modifiers(),
231: (22)                                  guiEvent=event)._process()
232: (4)                def mousePressEvent(self, event):
233: (8)                    button = self.buttond.get(event.button())
234: (8)                    if button is not None and self.figure is not None:
235: (12)                       MouseEvent("button_press_event", self,
236: (23)                                   *self.mouseEventCoords(event), button,
237: (23)                                   modifiers=self._mpl_modifiers(),
238: (23)                                   guiEvent=event)._process()
239: (4)                def mouseDoubleClickEvent(self, event):
240: (8)                    button = self.buttond.get(event.button())
241: (8)                    if button is not None and self.figure is not None:
242: (12)                       MouseEvent("button_press_event", self,
243: (23)                                   *self.mouseEventCoords(event), button, dblclick=True,
244: (23)                                   modifiers=self._mpl_modifiers(),
245: (23)                                   guiEvent=event)._process()
246: (4)                def mouseMoveEvent(self, event):
247: (8)                    if self.figure is None:
248: (12)                       return
249: (8)                    MouseEvent("motion_notify_event", self,
250: (19)                              *self.mouseEventCoords(event),
251: (19)                              modifiers=self._mpl_modifiers(),
252: (19)                              guiEvent=event)._process()
253: (4)                def mouseReleaseEvent(self, event):
254: (8)                    button = self.buttond.get(event.button())
255: (8)                    if button is not None and self.figure is not None:
256: (12)                       MouseEvent("button_release_event", self,
```

```
(23)                                *self.mouseEventCoords(event), button,
(23)                                modifiers=self._mpl_modifiers(),
(23)                                guiEvent=event)._process()
(4)                 def wheelEvent(self, event):
(8)                     if (event.pixelDelta().isNull()
(16)                            or QtWidgets.QApplication.instance().platformName() == "xcb"):
(12)                        steps = event.angleDelta().y() / 120
(8)                     else:
(12)                        steps = event.pixelDelta().y()
(8)                     if steps and self.figure is not None:
(12)                        MouseEvent("scroll_event", self,
(23)                                    *self.mouseEventCoords(event), step=steps,
(23)                                    modifiers=self._mpl_modifiers(),
(23)                                    guiEvent=event)._process()
(4)                 def keyPressEvent(self, event):
(8)                     key = self._get_key(event)
(8)                     if key is not None and self.figure is not None:
(12)                        KeyEvent("key_press_event", self,
(21)                                 key, *self.mouseEventCoords(),
(21)                                 guiEvent=event)._process()
(4)                 def keyReleaseEvent(self, event):
(8)                     key = self._get_key(event)
(8)                     if key is not None and self.figure is not None:
(12)                        KeyEvent("key_release_event", self,
(21)                                 key, *self.mouseEventCoords(),
(21)                                 guiEvent=event)._process()
(4)                 def resizeEvent(self, event):
(8)                     if self._in_resize_event:  # Prevent PyQt6 recursion
(12)                        return
(8)                     if self.figure is None:
(12)                        return
(8)                     self._in_resize_event = True
(8)                     try:
(12)                        w = event.size().width() * self.device_pixel_ratio
(12)                        h = event.size().height() * self.device_pixel_ratio
(12)                        dpival = self.figure.dpi
(12)                        winch = w / dpival
(12)                        hinch = h / dpival
(12)                        self.figure.set_size_inches(winch, hinch, forward=False)
(12)                        QtWidgets.QWidget.resizeEvent(self, event)
(12)                        ResizeEvent("resize_event", self)._process()
(12)                        self.draw_idle()
(8)                     finally:
(12)                        self._in_resize_event = False
(4)                 def sizeHint(self):
(8)                     w, h = self.get_width_height()
(8)                     return QtCore.QSize(w, h)
(4)                 def minumumSizeHint(self):
(8)                     return QtCore.QSize(10, 10)
(4)                 @staticmethod
(4)                 def _mpl_modifiers(modifiers=None, *, exclude=None):
(8)                     if modifiers is None:
(12)                        modifiers = QtWidgets.QApplication.instance().keyboardModifiers()
(8)                     modifiers = _to_int(modifiers)
(8)                     return [SPECIAL_KEYS[key].replace('control', 'ctrl')
(16)                            for mask, key in _MODIFIER_KEYS
(16)                            if exclude != key and modifiers & mask]
(4)                 def _get_key(self, event):
(8)                     event_key = event.key()
(8)                     mods = self._mpl_modifiers(exclude=event_key)
(8)                     try:
(12)                        key = SPECIAL_KEYS[event_key]
(8)                     except KeyError:
(12)                        if event_key > sys.maxunicode:
(16)                            return None
(12)                        key = chr(event_key)
(12)                        if 'shift' in mods:
(16)                            mods.remove('shift')
(12)                        else:
```

```
(16)                            key = key.lower()
(8)                         return '+'.join(mods + [key])
(4)                     def flush_events(self):
(8)                         QtWidgets.QApplication.instance().processEvents()
(4)                     def start_event_loop(self, timeout=0):
(8)                         if hasattr(self, "_event_loop") and self._event_loop.isRunning():
(12)                            raise RuntimeError("Event loop already running")
(8)                         self._event_loop = event_loop = QtCore.QEventLoop()
(8)                         if timeout > 0:
(12)                            _ = QtCore.QTimer.singleShot(int(timeout * 1000), event_loop.quit)
(8)                         with _allow_interrupt_qt(event_loop):
(12)                            qt_compat._exec(event_loop)
(4)                     def stop_event_loop(self, event=None):
(8)                         if hasattr(self, "_event_loop"):
(12)                            self._event_loop.quit()
(4)                     def draw(self):
(8)                         """Render the figure, and queue a request for a Qt draw."""
(8)                         if self._is_drawing:
(12)                            return
(8)                         with cbook._setattr_cm(self, _is_drawing=True):
(12)                            super().draw()
(8)                         self.update()
(4)                     def draw_idle(self):
(8)                         """Queue redraw of the Agg buffer and request Qt paintEvent."""
(8)                         if not (getattr(self, '_draw_pending', False) or
(16)                                getattr(self, '_is_drawing', False)):
(12)                            self._draw_pending = True
(12)                            QtCore.QTimer.singleShot(0, self._draw_idle)
(4)                     def blit(self, bbox=None):
(8)                         if bbox is None and self.figure:
(12)                            bbox = self.figure.bbox  # Blit the entire canvas if bbox is None.
(8)                         l, b, w, h = [int(pt / self.device_pixel_ratio) for pt in bbox.bounds]
(8)                         t = b + h
(8)                         self.repaint(l, self.rect().height() - t, w, h)
(4)                     def _draw_idle(self):
(8)                         with self._idle_draw_cntx():
(12)                            if not self._draw_pending:
(16)                                return
(12)                            self._draw_pending = False
(12)                            if self.height() <= 0 or self.width() <= 0:
(16)                                return
(12)                            try:
(16)                                self.draw()
(12)                            except Exception:
(16)                                traceback.print_exc()
(4)                     def drawRectangle(self, rect):
(8)                         if rect is not None:
(12)                            x0, y0, w, h = [int(pt / self.device_pixel_ratio) for pt in rect]
(12)                            x1 = x0 + w
(12)                            y1 = y0 + h
(12)                            def _draw_rect_callback(painter):
(16)                                pen = QtGui.QPen(
(20)                                    QtGui.QColor("black"),
(20)                                    1 / self.device_pixel_ratio
(16)                                )
(16)                                pen.setDashPattern([3, 3])
(16)                                for color, offset in [
(24)                                        (QtGui.QColor("black"), 0),
(24)                                        (QtGui.QColor("white"), 3),
(16)                                ]:
(20)                                    pen.setDashOffset(offset)
(20)                                    pen.setColor(color)
(20)                                    painter.setPen(pen)
(20)                                    painter.drawLine(x0, y0, x0, y1)
(20)                                    painter.drawLine(x0, y0, x1, y0)
(20)                                    painter.drawLine(x0, y1, x1, y1)
(20)                                    painter.drawLine(x1, y0, x1, y1)
(8)                         else:
(12)                            def _draw_rect_callback(painter):
```

```
(16)                            return
(8)                     self._draw_rect_callback = _draw_rect_callback
(8)                     self.update()
(0)             class MainWindow(QtWidgets.QMainWindow):
(4)                 closing = QtCore.Signal()
(4)                 def closeEvent(self, event):
(8)                     self.closing.emit()
(8)                     super().closeEvent(event)
(0)             class FigureManagerQT(FigureManagerBase):
(4)                 """
(4)                 Attributes
(4)                 ----------
(4)                 canvas : `FigureCanvas`
(8)                     The FigureCanvas instance
(4)                 num : int or str
(8)                     The Figure number
(4)                 toolbar : qt.QToolBar
(8)                     The qt.QToolBar
(4)                 window : qt.QMainWindow
(8)                     The qt.QMainWindow
(4)                 """
(4)                 def __init__(self, canvas, num):
(8)                     self.window = MainWindow()
(8)                     super().__init__(canvas, num)
(8)                     self.window.closing.connect(self._widgetclosed)
(8)                     if sys.platform != "darwin":
(12)                        image = str(cbook._get_data_path('images/matplotlib.svg'))
(12)                        icon = QtGui.QIcon(image)
(12)                        self.window.setWindowIcon(icon)
(8)                     self.window._destroying = False
(8)                     if self.toolbar:
(12)                        self.window.addToolBar(self.toolbar)
(12)                        tbs_height = self.toolbar.sizeHint().height()
(8)                     else:
(12)                        tbs_height = 0
(8)                     cs = canvas.sizeHint()
(8)                     cs_height = cs.height()
(8)                     height = cs_height + tbs_height
(8)                     self.window.resize(cs.width(), height)
(8)                     self.window.setCentralWidget(self.canvas)
(8)                     if mpl.is_interactive():
(12)                        self.window.show()
(12)                        self.canvas.draw_idle()
(8)                     self.canvas.setFocusPolicy(QtCore.Qt.FocusPolicy.StrongFocus)
(8)                     self.canvas.setFocus()
(8)                     self.window.raise_()
(4)                 def full_screen_toggle(self):
(8)                     if self.window.isFullScreen():
(12)                        self.window.showNormal()
(8)                     else:
(12)                        self.window.showFullScreen()
(4)                 def _widgetclosed(self):
(8)                     CloseEvent("close_event", self.canvas)._process()
(8)                     if self.window._destroying:
(12)                        return
(8)                     self.window._destroying = True
(8)                     try:
(12)                        Gcf.destroy(self)
(8)                     except AttributeError:
(12)                        pass
(4)                 def resize(self, width, height):
(8)                     width = int(width / self.canvas.device_pixel_ratio)
(8)                     height = int(height / self.canvas.device_pixel_ratio)
(8)                     extra_width = self.window.width() - self.canvas.width()
(8)                     extra_height = self.window.height() - self.canvas.height()
(8)                     self.canvas.resize(width, height)
(8)                     self.window.resize(width + extra_width, height + extra_height)
(4)                 @classmethod
(4)                 def start_main_loop(cls):
```

```
464: (8)                    qapp = QtWidgets.QApplication.instance()
465: (8)                    if qapp:
466: (12)                       with _allow_interrupt_qt(qapp):
467: (16)                           qt_compat._exec(qapp)
468: (4)                def show(self):
469: (8)                    self.window._destroying = False
470: (8)                    self.window.show()
471: (8)                    if mpl.rcParams['figure.raise_window']:
472: (12)                       self.window.activateWindow()
473: (12)                       self.window.raise_()
474: (4)                def destroy(self, *args):
475: (8)                    if QtWidgets.QApplication.instance() is None:
476: (12)                       return
477: (8)                    if self.window._destroying:
478: (12)                       return
479: (8)                    self.window._destroying = True
480: (8)                    if self.toolbar:
481: (12)                       self.toolbar.destroy()
482: (8)                    self.window.close()
483: (4)                def get_window_title(self):
484: (8)                    return self.window.windowTitle()
485: (4)                def set_window_title(self, title):
486: (8)                    self.window.setWindowTitle(title)
487: (0)            class NavigationToolbar2QT(NavigationToolbar2, QtWidgets.QToolBar):
488: (4)                _message = QtCore.Signal(str)  # Remove once deprecation below elapses.
489: (4)                message = _api.deprecate_privatize_attribute("3.8")
490: (4)                toolitems = [*NavigationToolbar2.toolitems]
491: (4)                toolitems.insert(
492: (8)                    [name for name, *_ in toolitems].index("Subplots") + 1,
493: (8)                    ("Customize", "Edit axis, curve and image parameters",
494: (9)                     "qt4_editor_options", "edit_parameters"))
495: (4)                def __init__(self, canvas, parent=None, coordinates=True):
496: (8)                    """coordinates: should we show the coordinates on the right?"""
497: (8)                    QtWidgets.QToolBar.__init__(self, parent)
498: (8)                    self.setAllowedAreas(QtCore.Qt.ToolBarArea(
499: (12)                       _to_int(QtCore.Qt.ToolBarArea.TopToolBarArea) |
500: (12)                       _to_int(QtCore.Qt.ToolBarArea.BottomToolBarArea)))
501: (8)                    self.coordinates = coordinates
502: (8)                    self._actions = {}  # mapping of toolitem method names to QActions.
503: (8)                    self._subplot_dialog = None
504: (8)                    for text, tooltip_text, image_file, callback in self.toolitems:
505: (12)                       if text is None:
506: (16)                           self.addSeparator()
507: (12)                       else:
508: (16)                           slot = getattr(self, callback)
509: (16)                           slot = functools.wraps(slot)(functools.partial(slot))
510: (16)                           slot = QtCore.Slot()(slot)
511: (16)                           a = self.addAction(self._icon(image_file + '.png'),
512: (35)                                              text, slot)
513: (16)                           self._actions[callback] = a
514: (16)                           if callback in ['zoom', 'pan']:
515: (20)                               a.setCheckable(True)
516: (16)                           if tooltip_text is not None:
517: (20)                               a.setToolTip(tooltip_text)
518: (8)                    if self.coordinates:
519: (12)                       self.locLabel = QtWidgets.QLabel("", self)
520: (12)                       self.locLabel.setAlignment(QtCore.Qt.AlignmentFlag(
521: (16)                           _to_int(QtCore.Qt.AlignmentFlag.AlignRight) |
522: (16)                           _to_int(QtCore.Qt.AlignmentFlag.AlignVCenter)))
523: (12)                       self.locLabel.setSizePolicy(QtWidgets.QSizePolicy(
524: (16)                           QtWidgets.QSizePolicy.Policy.Expanding,
525: (16)                           QtWidgets.QSizePolicy.Policy.Ignored,
526: (12)                       ))
527: (12)                       labelAction = self.addWidget(self.locLabel)
528: (12)                       labelAction.setVisible(True)
529: (8)                    NavigationToolbar2.__init__(self, canvas)
530: (4)                def _icon(self, name):
531: (8)                    """
532: (8)                    Construct a `.QIcon` from an image file *name*, including the
```

```
extension
533: (8)                    and relative to Matplotlib's "images" data directory.
534: (8)                    """
535: (8)                    path_regular = cbook._get_data_path('images', name)
536: (8)                    path_large = path_regular.with_name(
537: (12)                       path_regular.name.replace('.png', '_large.png'))
538: (8)                    filename = str(path_large if path_large.exists() else path_regular)
539: (8)                    pm = QtGui.QPixmap(filename)
540: (8)                    pm.setDevicePixelRatio(
541: (12)                       self.devicePixelRatioF() or 1)  # rarely, devicePixelRatioF=0
542: (8)                    if self.palette().color(self.backgroundRole()).value() < 128:
543: (12)                       icon_color = self.palette().color(self.foregroundRole())
544: (12)                       mask = pm.createMaskFromColor(
545: (16)                           QtGui.QColor('black'),
546: (16)                           QtCore.Qt.MaskMode.MaskOutColor)
547: (12)                       pm.fill(icon_color)
548: (12)                       pm.setMask(mask)
549: (8)                    return QtGui.QIcon(pm)
550: (4)                def edit_parameters(self):
551: (8)                    axes = self.canvas.figure.get_axes()
552: (8)                    if not axes:
553: (12)                       QtWidgets.QMessageBox.warning(
554: (16)                           self.canvas.parent(), "Error", "There are no Axes to edit.")
555: (12)                       return
556: (8)                    elif len(axes) == 1:
557: (12)                       ax, = axes
558: (8)                    else:
559: (12)                       titles = [
560: (16)                           ax.get_label() or
561: (16)                           ax.get_title() or
562: (16)                           ax.get_title("left") or
563: (16)                           ax.get_title("right") or
564: (16)                           " - ".join(filter(None, [ax.get_xlabel(), ax.get_ylabel()]))
or
565: (16)                           f"<anonymous {type(ax).__name__}>"
566: (16)                           for ax in axes]
567: (12)                       duplicate_titles = [
568: (16)                           title for title in titles if titles.count(title) > 1]
569: (12)                       for i, ax in enumerate(axes):
570: (16)                           if titles[i] in duplicate_titles:
571: (20)                               titles[i] += f" (id: {id(ax):#x})"  # Deduplicate titles.
572: (12)                       item, ok = QtWidgets.QInputDialog.getItem(
573: (16)                           self.canvas.parent(),
574: (16)                           'Customize', 'Select Axes:', titles, 0, False)
575: (12)                       if not ok:
576: (16)                           return
577: (12)                       ax = axes[titles.index(item)]
578: (8)                    figureoptions.figure_edit(ax, self)
579: (4)                def _update_buttons_checked(self):
580: (8)                    if 'pan' in self._actions:
581: (12)                       self._actions['pan'].setChecked(self.mode.name == 'PAN')
582: (8)                    if 'zoom' in self._actions:
583: (12)                       self._actions['zoom'].setChecked(self.mode.name == 'ZOOM')
584: (4)                def pan(self, *args):
585: (8)                    super().pan(*args)
586: (8)                    self._update_buttons_checked()
587: (4)                def zoom(self, *args):
588: (8)                    super().zoom(*args)
589: (8)                    self._update_buttons_checked()
590: (4)                def set_message(self, s):
591: (8)                    self._message.emit(s)
592: (8)                    if self.coordinates:
593: (12)                       self.locLabel.setText(s)
594: (4)                def draw_rubberband(self, event, x0, y0, x1, y1):
595: (8)                    height = self.canvas.figure.bbox.height
596: (8)                    y1 = height - y1
597: (8)                    y0 = height - y0
598: (8)                    rect = [int(val) for val in (x0, y0, x1 - x0, y1 - y0)]
599: (8)                    self.canvas.drawRectangle(rect)
```

```
(4)     def remove_rubberband(self):
(8)         self.canvas.drawRectangle(None)
(4)     def configure_subplots(self):
(8)         if self._subplot_dialog is None:
(12)            self._subplot_dialog = SubplotToolQt(
(16)                self.canvas.figure, self.canvas.parent())
(12)            self.canvas.mpl_connect(
(16)                "close_event", lambda e: self._subplot_dialog.reject())
(8)         self._subplot_dialog.update_from_current_subplotpars()
(8)         self._subplot_dialog.setModal(True)
(8)         self._subplot_dialog.show()
(8)         return self._subplot_dialog
(4)     def save_figure(self, *args):
(8)         filetypes = self.canvas.get_supported_filetypes_grouped()
(8)         sorted_filetypes = sorted(filetypes.items())
(8)         default_filetype = self.canvas.get_default_filetype()
(8)         startpath = os.path.expanduser(mpl.rcParams['savefig.directory'])
(8)         start = os.path.join(startpath, self.canvas.get_default_filename())
(8)         filters = []
(8)         selectedFilter = None
(8)         for name, exts in sorted_filetypes:
(12)            exts_list = " ".join(['*.%s' % ext for ext in exts])
(12)            filter = f'{name} ({exts_list})'
(12)            if default_filetype in exts:
(16)                selectedFilter = filter
(12)            filters.append(filter)
(8)         filters = ';;'.join(filters)
(8)         fname, filter = QtWidgets.QFileDialog.getSaveFileName(
(12)            self.canvas.parent(), "Choose a filename to save to", start,
(12)            filters, selectedFilter)
(8)         if fname:
(12)            if startpath != "":
(16)                mpl.rcParams['savefig.directory'] = os.path.dirname(fname)
(12)            try:
(16)                self.canvas.figure.savefig(fname)
(12)            except Exception as e:
(16)                QtWidgets.QMessageBox.critical(
(20)                    self, "Error saving file", str(e),
(20)                    QtWidgets.QMessageBox.StandardButton.Ok,
(20)                    QtWidgets.QMessageBox.StandardButton.NoButton)
(4)     def set_history_buttons(self):
(8)         can_backward = self._nav_stack._pos > 0
(8)         can_forward = self._nav_stack._pos < len(self._nav_stack) - 1
(8)         if 'back' in self._actions:
(12)            self._actions['back'].setEnabled(can_backward)
(8)         if 'forward' in self._actions:
(12)            self._actions['forward'].setEnabled(can_forward)
(0) class SubplotToolQt(QtWidgets.QDialog):
(4)     def __init__(self, targetfig, parent):
(8)         super().__init__()
(8)         self.setWindowIcon(QtGui.QIcon(
(12)            str(cbook._get_data_path("images/matplotlib.png"))))
(8)         self.setObjectName("SubplotTool")
(8)         self._spinboxes = {}
(8)         main_layout = QtWidgets.QHBoxLayout()
(8)         self.setLayout(main_layout)
(8)         for group, spinboxes, buttons in [
(16)            ("Borders",
(17)             ["top", "bottom", "left", "right"],
(17)             [("Export values", self._export_values)]),
(16)            ("Spacings",
(17)             ["hspace", "wspace"],
(17)             [("Tight layout", self._tight_layout),
(18)              ("Reset", self._reset),
(18)              ("Close", self.close)])]:
(12)            layout = QtWidgets.QVBoxLayout()
(12)            main_layout.addLayout(layout)
(12)            box = QtWidgets.QGroupBox(group)
(12)            layout.addWidget(box)
```

```
669: (12)                        inner = QtWidgets.QFormLayout(box)
670: (12)                        for name in spinboxes:
671: (16)                            self._spinboxes[name] = spinbox = QtWidgets.QDoubleSpinBox()
672: (16)                            spinbox.setRange(0, 1)
673: (16)                            spinbox.setDecimals(3)
674: (16)                            spinbox.setSingleStep(0.005)
675: (16)                            spinbox.setKeyboardTracking(False)
676: (16)                            spinbox.valueChanged.connect(self._on_value_changed)
677: (16)                            inner.addRow(name, spinbox)
678: (12)                        layout.addStretch(1)
679: (12)                        for name, method in buttons:
680: (16)                            button = QtWidgets.QPushButton(name)
681: (16)                            button.setAutoDefault(False)
682: (16)                            button.clicked.connect(method)
683: (16)                            layout.addWidget(button)
684: (16)                            if name == "Close":
685: (20)                                button.setFocus()
686: (8)                    self._figure = targetfig
687: (8)                    self._defaults = {}
688: (8)                    self._export_values_dialog = None
689: (8)                    self.update_from_current_subplotpars()
690: (4)                def update_from_current_subplotpars(self):
691: (8)                    self._defaults = {spinbox: getattr(self._figure.subplotpars, name)
692: (26)                                      for name, spinbox in self._spinboxes.items()}
693: (8)                    self._reset()  # Set spinbox current values without triggering
signals.
694: (4)                def _export_values(self):
695: (8)                    self._export_values_dialog = QtWidgets.QDialog()
696: (8)                    layout = QtWidgets.QVBoxLayout()
697: (8)                    self._export_values_dialog.setLayout(layout)
698: (8)                    text = QtWidgets.QPlainTextEdit()
699: (8)                    text.setReadOnly(True)
700: (8)                    layout.addWidget(text)
701: (8)                    text.setPlainText(
702: (12)                       ",\n".join(f"{attr}={spinbox.value():.3}"
703: (23)                                  for attr, spinbox in self._spinboxes.items()))
704: (8)                    size = text.maximumSize()
705: (8)                    size.setHeight(
706: (12)                       QtGui.QFontMetrics(text.document().defaultFont())
707: (12)                       .size(0, text.toPlainText()).height() + 20)
708: (8)                    text.setMaximumSize(size)
709: (8)                    self._export_values_dialog.show()
710: (4)                def _on_value_changed(self):
711: (8)                    spinboxes = self._spinboxes
712: (8)                    for lower, higher in [("bottom", "top"), ("left", "right")]:
713: (12)                       spinboxes[higher].setMinimum(spinboxes[lower].value() + .001)
714: (12)                       spinboxes[lower].setMaximum(spinboxes[higher].value() - .001)
715: (8)                    self._figure.subplots_adjust(
716: (12)                       **{attr: spinbox.value() for attr, spinbox in spinboxes.items()})
717: (8)                    self._figure.canvas.draw_idle()
718: (4)                def _tight_layout(self):
719: (8)                    self._figure.tight_layout()
720: (8)                    for attr, spinbox in self._spinboxes.items():
721: (12)                       spinbox.blockSignals(True)
722: (12)                       spinbox.setValue(getattr(self._figure.subplotpars, attr))
723: (12)                       spinbox.blockSignals(False)
724: (8)                    self._figure.canvas.draw_idle()
725: (4)                def _reset(self):
726: (8)                    for spinbox, value in self._defaults.items():
727: (12)                       spinbox.setRange(0, 1)
728: (12)                       spinbox.blockSignals(True)
729: (12)                       spinbox.setValue(value)
730: (12)                       spinbox.blockSignals(False)
731: (8)                    self._on_value_changed()
732: (0)            class ToolbarQt(ToolContainerBase, QtWidgets.QToolBar):
733: (4)                def __init__(self, toolmanager, parent=None):
734: (8)                    ToolContainerBase.__init__(self, toolmanager)
735: (8)                    QtWidgets.QToolBar.__init__(self, parent)
736: (8)                    self.setAllowedAreas(QtCore.Qt.ToolBarArea(
```

```
(12)                        _to_int(QtCore.Qt.ToolBarArea.TopToolBarArea) |
(12)                        _to_int(QtCore.Qt.ToolBarArea.BottomToolBarArea)))
(8)                     message_label = QtWidgets.QLabel("")
(8)                     message_label.setAlignment(QtCore.Qt.AlignmentFlag(
(12)                        _to_int(QtCore.Qt.AlignmentFlag.AlignRight) |
(12)                        _to_int(QtCore.Qt.AlignmentFlag.AlignVCenter)))
(8)                     message_label.setSizePolicy(QtWidgets.QSizePolicy(
(12)                        QtWidgets.QSizePolicy.Policy.Expanding,
(12)                        QtWidgets.QSizePolicy.Policy.Ignored,
(8)                     ))
(8)                     self._message_action = self.addWidget(message_label)
(8)                     self._toolitems = {}
(8)                     self._groups = {}
(4)                 def add_toolitem(
(12)                        self, name, group, position, image_file, description, toggle):
(8)                     button = QtWidgets.QToolButton(self)
(8)                     if image_file:
(12)                        button.setIcon(NavigationToolbar2QT._icon(self, image_file))
(8)                     button.setText(name)
(8)                     if description:
(12)                        button.setToolTip(description)
(8)                     def handler():
(12)                        self.trigger_tool(name)
(8)                     if toggle:
(12)                        button.setCheckable(True)
(12)                        button.toggled.connect(handler)
(8)                     else:
(12)                        button.clicked.connect(handler)
(8)                     self._toolitems.setdefault(name, [])
(8)                     self._add_to_group(group, name, button, position)
(8)                     self._toolitems[name].append((button, handler))
(4)                 def _add_to_group(self, group, name, button, position):
(8)                     gr = self._groups.get(group, [])
(8)                     if not gr:
(12)                        sep = self.insertSeparator(self._message_action)
(12)                        gr.append(sep)
(8)                     before = gr[position]
(8)                     widget = self.insertWidget(before, button)
(8)                     gr.insert(position, widget)
(8)                     self._groups[group] = gr
(4)                 def toggle_toolitem(self, name, toggled):
(8)                     if name not in self._toolitems:
(12)                        return
(8)                     for button, handler in self._toolitems[name]:
(12)                        button.toggled.disconnect(handler)
(12)                        button.setChecked(toggled)
(12)                        button.toggled.connect(handler)
(4)                 def remove_toolitem(self, name):
(8)                     for button, handler in self._toolitems.pop(name, []):
(12)                        button.setParent(None)
(4)                 def set_message(self, s):
(8)                     self.widgetForAction(self._message_action).setText(s)
(0)             @backend_tools._register_tool_class(FigureCanvasQT)
(0)             class ConfigureSubplotsQt(backend_tools.ConfigureSubplotsBase):
(4)                 def __init__(self, *args, **kwargs):
(8)                     super().__init__(*args, **kwargs)
(8)                     self._subplot_dialog = None
(4)                 def trigger(self, *args):
(8)                     NavigationToolbar2QT.configure_subplots(self)
(0)             @backend_tools._register_tool_class(FigureCanvasQT)
(0)             class SaveFigureQt(backend_tools.SaveFigureBase):
(4)                 def trigger(self, *args):
(8)                     NavigationToolbar2QT.save_figure(
(12)                        self._make_classic_style_pseudo_toolbar())
(0)             @backend_tools._register_tool_class(FigureCanvasQT)
(0)             class RubberbandQt(backend_tools.RubberbandBase):
(4)                 def draw_rubberband(self, x0, y0, x1, y1):
(8)                     NavigationToolbar2QT.draw_rubberband(
(12)                        self._make_classic_style_pseudo_toolbar(), None, x0, y0, x1, y1)
```

```
806: (4)                def remove_rubberband(self):
807: (8)                    NavigationToolbar2QT.remove_rubberband(
808: (12)                       self._make_classic_style_pseudo_toolbar())
809: (0)            @backend_tools._register_tool_class(FigureCanvasQT)
810: (0)            class HelpQt(backend_tools.ToolHelpBase):
811: (4)                def trigger(self, *args):
812: (8)                    QtWidgets.QMessageBox.information(None, "Help", self._get_help_html())
813: (0)            @backend_tools._register_tool_class(FigureCanvasQT)
814: (0)            class ToolCopyToClipboardQT(backend_tools.ToolCopyToClipboardBase):
815: (4)                def trigger(self, *args, **kwargs):
816: (8)                    pixmap = self.canvas.grab()
817: (8)                    QtWidgets.QApplication.instance().clipboard().setPixmap(pixmap)
818: (0)            FigureManagerQT._toolbar2_class = NavigationToolbar2QT
819: (0)            FigureManagerQT._toolmanager_toolbar_class = ToolbarQt
820: (0)            @_Backend.export
821: (0)            class _BackendQT(_Backend):
822: (4)                backend_version = __version__
823: (4)                FigureCanvas = FigureCanvasQT
824: (4)                FigureManager = FigureManagerQT
825: (4)                mainloop = FigureManagerQT.start_main_loop
```

File 96 - backend_qtagg.py:

```
1: (0)              """
2: (0)              Render to qt from agg.
3: (0)              """
4: (0)              import ctypes
5: (0)              from matplotlib.transforms import Bbox
6: (0)              from .qt_compat import QT_API, QtCore, QtGui
7: (0)              from .backend_agg import FigureCanvasAgg
8: (0)              from .backend_qt import _BackendQT, FigureCanvasQT
9: (0)              from .backend_qt import (  # noqa: F401 # pylint: disable=W0611
10: (4)                 FigureManagerQT, NavigationToolbar2QT)
11: (0)             class FigureCanvasQTAgg(FigureCanvasAgg, FigureCanvasQT):
12: (4)                 def paintEvent(self, event):
13: (8)                     """
14: (8)                     Copy the image from the Agg canvas to the qt.drawable.
15: (8)                     In Qt, all drawing should be done inside of here when a widget is
16: (8)                     shown onscreen.
17: (8)                     """
18: (8)                     self._draw_idle()  # Only does something if a draw is pending.
19: (8)                     if not hasattr(self, 'renderer'):
20: (12)                        return
21: (8)                     painter = QtGui.QPainter(self)
22: (8)                     try:
23: (12)                        rect = event.rect()
24: (12)                        width = rect.width() * self.device_pixel_ratio
25: (12)                        height = rect.height() * self.device_pixel_ratio
26: (12)                        left, top = self.mouseEventCoords(rect.topLeft())
27: (12)                        bottom = top - height
28: (12)                        right = left + width
29: (12)                        bbox = Bbox([[left, bottom], [right, top]])
30: (12)                        buf = memoryview(self.copy_from_bbox(bbox))
31: (12)                        if QT_API == "PyQt6":
32: (16)                            from PyQt6 import sip
33: (16)                            ptr = int(sip.voidptr(buf))
34: (12)                        else:
35: (16)                            ptr = buf
36: (12)                        painter.eraseRect(rect)  # clear the widget canvas
37: (12)                        qimage = QtGui.QImage(ptr, buf.shape[1], buf.shape[0],
38: (34)                                              QtGui.QImage.Format.Format_RGBA8888)
39: (12)                        qimage.setDevicePixelRatio(self.device_pixel_ratio)
40: (12)                        origin = QtCore.QPoint(rect.left(), rect.top())
41: (12)                        painter.drawImage(origin, qimage)
42: (12)                        if QT_API == "PySide2" and QtCore.__version_info__ < (5, 12):
43: (16)                            ctypes.c_long.from_address(id(buf)).value = 1
44: (12)                        self._draw_rect_callback(painter)
```

```
45: (8)                     finally:
46: (12)                        painter.end()
47: (4)                 def print_figure(self, *args, **kwargs):
48: (8)                     super().print_figure(*args, **kwargs)
49: (8)                     self._draw_pending = True
50: (0)             @_BackendQT.export
51: (0)             class _BackendQTAgg(_BackendQT):
52: (4)                 FigureCanvas = FigureCanvasQTAgg
```

File 97 - backend_qtcairo.py:

```
1: (0)              import ctypes
2: (0)              from .backend_cairo import cairo, FigureCanvasCairo
3: (0)              from .backend_qt import _BackendQT, FigureCanvasQT
4: (0)              from .qt_compat import QT_API, QtCore, QtGui
5: (0)              class FigureCanvasQTCairo(FigureCanvasCairo, FigureCanvasQT):
6: (4)                  def draw(self):
7: (8)                      if hasattr(self._renderer.gc, "ctx"):
8: (12)                         self._renderer.dpi = self.figure.dpi
9: (12)                         self.figure.draw(self._renderer)
10: (8)                     super().draw()
11: (4)                 def paintEvent(self, event):
12: (8)                     width = int(self.device_pixel_ratio * self.width())
13: (8)                     height = int(self.device_pixel_ratio * self.height())
14: (8)                     if (width, height) != self._renderer.get_canvas_width_height():
15: (12)                        surface = cairo.ImageSurface(cairo.FORMAT_ARGB32, width, height)
16: (12)                        self._renderer.set_context(cairo.Context(surface))
17: (12)                        self._renderer.dpi = self.figure.dpi
18: (12)                        self.figure.draw(self._renderer)
19: (8)                     buf = self._renderer.gc.ctx.get_target().get_data()
20: (8)                     if QT_API == "PyQt6":
21: (12)                        from PyQt6 import sip
22: (12)                        ptr = int(sip.voidptr(buf))
23: (8)                     else:
24: (12)                        ptr = buf
25: (8)                     qimage = QtGui.QImage(
26: (12)                        ptr, width, height,
27: (12)                        QtGui.QImage.Format.Format_ARGB32_Premultiplied)
28: (8)                     if QT_API == "PySide2" and QtCore.__version_info__ < (5, 12):
29: (12)                        ctypes.c_long.from_address(id(buf)).value = 1
30: (8)                     qimage.setDevicePixelRatio(self.device_pixel_ratio)
31: (8)                     painter = QtGui.QPainter(self)
32: (8)                     painter.eraseRect(event.rect())
33: (8)                     painter.drawImage(0, 0, qimage)
34: (8)                     self._draw_rect_callback(painter)
35: (8)                     painter.end()
36: (0)             @_BackendQT.export
37: (0)             class _BackendQTCairo(_BackendQT):
38: (4)                 FigureCanvas = FigureCanvasQTCairo
```

File 98 - backend_qt5.py:

```
1: (0)              from .. import backends
2: (0)              backends._QT_FORCE_QT5_BINDING = True
3: (0)              from .backend_qt import (  # noqa
4: (4)                  SPECIAL_KEYS,
5: (4)                  cursord, _create_qApp, _BackendQT, TimerQT, MainWindow, FigureCanvasQT,
6: (4)                  FigureManagerQT, ToolbarQt, NavigationToolbar2QT, SubplotToolQt,
7: (4)                  SaveFigureQt, ConfigureSubplotsQt, RubberbandQt,
8: (4)                  HelpQt, ToolCopyToClipboardQT,
9: (4)                  FigureCanvasBase,  FigureManagerBase, MouseButton, NavigationToolbar2,
10: (4)                 TimerBase, ToolContainerBase, figureoptions, Gcf
11: (0)             )
12: (0)             from . import backend_qt as _backend_qt  # noqa
13: (0)             @_BackendQT.export
```

```
14: (0)             class _BackendQT5(_BackendQT):
15: (4)                 pass
16: (0)             def __getattr__(name):
17: (4)                 if name == 'qApp':
18: (8)                     return _backend_qt.qApp
19: (4)                 raise AttributeError(f"module {__name__!r} has no attribute {name!r}")
```

File 99 - backend_qt5agg.py:

```
1: (0)              """
2: (0)              Render to qt from agg
3: (0)              """
4: (0)              from .. import backends
5: (0)              backends._QT_FORCE_QT5_BINDING = True
6: (0)              from .backend_qtagg import (    # noqa: F401, E402 # pylint: disable=W0611
7: (4)                  _BackendQTAgg, FigureCanvasQTAgg, FigureManagerQT, NavigationToolbar2QT,
8: (4)                  FigureCanvasAgg, FigureCanvasQT)
9: (0)              @_BackendQTAgg.export
10: (0)             class _BackendQT5Agg(_BackendQTAgg):
11: (4)                 pass
```

File 100 - backend_qt5cairo.py:

```
1: (0)              from .. import backends
2: (0)              backends._QT_FORCE_QT5_BINDING = True
3: (0)              from .backend_qtcairo import (  # noqa: F401, E402 # pylint: disable=W0611
4: (4)                  _BackendQTCairo, FigureCanvasQTCairo, FigureCanvasCairo, FigureCanvasQT
5: (0)              )
6: (0)              @_BackendQTCairo.export
7: (0)              class _BackendQT5Cairo(_BackendQTCairo):
8: (4)                  pass
```

File 101 - backend_svg.py:

```
1: (0)              import base64
2: (0)              import codecs
3: (0)              import datetime
4: (0)              import gzip
5: (0)              import hashlib
6: (0)              from io import BytesIO
7: (0)              import itertools
8: (0)              import logging
9: (0)              import os
10: (0)             import re
11: (0)             import uuid
12: (0)             import numpy as np
13: (0)             from PIL import Image
14: (0)             import matplotlib as mpl
15: (0)             from matplotlib import cbook, font_manager as fm
16: (0)             from matplotlib.backend_bases import (
17: (5)                  _Backend, FigureCanvasBase, FigureManagerBase, RendererBase)
18: (0)             from matplotlib.backends.backend_mixed import MixedModeRenderer
19: (0)             from matplotlib.colors import rgb2hex
20: (0)             from matplotlib.dates import UTC
21: (0)             from matplotlib.path import Path
22: (0)             from matplotlib import _path
23: (0)             from matplotlib.transforms import Affine2D, Affine2DBase
24: (0)             _log = logging.getLogger(__name__)
25: (0)             def _escape_cdata(s):
26: (4)                 s = s.replace("&", "&amp;")
27: (4)                 s = s.replace("<", "&lt;")
28: (4)                 s = s.replace(">", "&gt;")
29: (4)                 return s
```

```
30: (0)             _escape_xml_comment = re.compile(r'-(?=-)')
31: (0)             def _escape_comment(s):
32: (4)                 s = _escape_cdata(s)
33: (4)                 return _escape_xml_comment.sub('- ', s)
34: (0)             def _escape_attrib(s):
35: (4)                 s = s.replace("&", "&amp;")
36: (4)                 s = s.replace("'", "&apos;")
37: (4)                 s = s.replace('"', "&quot;")
38: (4)                 s = s.replace("<", "&lt;")
39: (4)                 s = s.replace(">", "&gt;")
40: (4)                 return s
41: (0)             def _quote_escape_attrib(s):
42: (4)                 return ('"' + _escape_cdata(s) + '"' if '"' not in s else
43: (12)                        "'" + _escape_cdata(s) + "'" if "'" not in s else
44: (12)                        '"' + _escape_attrib(s) + '"')
45: (0)             def _short_float_fmt(x):
46: (4)                 """
47: (4)                 Create a short string representation of a float, which is %f
48: (4)                 formatting with trailing zeros and the decimal point removed.
49: (4)                 """
50: (4)                 return f'{x:f}'.rstrip('0').rstrip('.')
51: (0)             class XMLWriter:
52: (4)                 """
53: (4)                 Parameters
54: (4)                 ----------
55: (4)                 file : writable text file-like object
56: (4)                 """
57: (4)                 def __init__(self, file):
58: (8)                     self.__write = file.write
59: (8)                     if hasattr(file, "flush"):
60: (12)                        self.flush = file.flush
61: (8)                     self.__open = 0  # true if start tag is open
62: (8)                     self.__tags = []
63: (8)                     self.__data = []
64: (8)                     self.__indentation = " " * 64
65: (4)                 def __flush(self, indent=True):
66: (8)                     if self.__open:
67: (12)                        if indent:
68: (16)                            self.__write(">\n")
69: (12)                        else:
70: (16)                            self.__write(">")
71: (12)                        self.__open = 0
72: (8)                     if self.__data:
73: (12)                        data = ''.join(self.__data)
74: (12)                        self.__write(_escape_cdata(data))
75: (12)                        self.__data = []
76: (4)                 def start(self, tag, attrib={}, **extra):
77: (8)                     """
78: (8)                     Open a new element.  Attributes can be given as keyword
79: (8)                     arguments, or as a string/string dictionary. The method returns
80: (8)                     an opaque identifier that can be passed to the :meth:`close`
81: (8)                     method, to close all open elements up to and including this one.
82: (8)                     Parameters
83: (8)                     ----------
84: (8)                     tag
85: (12)                        Element tag.
86: (8)                     attrib
87: (12)                        Attribute dictionary.  Alternatively, attributes can be given as
88: (12)                        keyword arguments.
89: (8)                     Returns
90: (8)                     -------
91: (8)                     An element identifier.
92: (8)                     """
93: (8)                     self.__flush()
94: (8)                     tag = _escape_cdata(tag)
95: (8)                     self.__data = []
96: (8)                     self.__tags.append(tag)
97: (8)                     self.__write(self.__indentation[:len(self.__tags) - 1])
98: (8)                     self.__write(f"<{tag}")
```

```
 99: (8)                    for k, v in {**attrib, **extra}.items():
100: (12)                       if v:
101: (16)                           k = _escape_cdata(k)
102: (16)                           v = _quote_escape_attrib(v)
103: (16)                           self.__write(f' {k}={v}')
104: (8)                   self.__open = 1
105: (8)                   return len(self.__tags) - 1
106: (4)               def comment(self, comment):
107: (8)                   """
108: (8)                   Add a comment to the output stream.
109: (8)                   Parameters
110: (8)                   ----------
111: (8)                   comment : str
112: (12)                      Comment text.
113: (8)                   """
114: (8)                   self.__flush()
115: (8)                   self.__write(self.__indentation[:len(self.__tags)])
116: (8)                   self.__write(f"<!-- {_escape_comment(comment)} -->\n")
117: (4)               def data(self, text):
118: (8)                   """
119: (8)                   Add character data to the output stream.
120: (8)                   Parameters
121: (8)                   ----------
122: (8)                   text : str
123: (12)                      Character data.
124: (8)                   """
125: (8)                   self.__data.append(text)
126: (4)               def end(self, tag=None, indent=True):
127: (8)                   """
128: (8)                   Close the current element (opened by the most recent call to
129: (8)                   :meth:`start`).
130: (8)                   Parameters
131: (8)                   ----------
132: (8)                   tag
133: (12)                      Element tag.  If given, the tag must match the start tag.  If
134: (12)                      omitted, the current element is closed.
135: (8)                   indent : bool, default: True
136: (8)                   """
137: (8)                   if tag:
138: (12)                      assert self.__tags, f"unbalanced end({tag})"
139: (12)                      assert _escape_cdata(tag) == self.__tags[-1], \
140: (16)                          f"expected end({self.__tags[-1]}), got {tag}"
141: (8)                   else:
142: (12)                      assert self.__tags, "unbalanced end()"
143: (8)                   tag = self.__tags.pop()
144: (8)                   if self.__data:
145: (12)                      self.__flush(indent)
146: (8)                   elif self.__open:
147: (12)                      self.__open = 0
148: (12)                      self.__write("/>\n")
149: (12)                      return
150: (8)                   if indent:
151: (12)                      self.__write(self.__indentation[:len(self.__tags)])
152: (8)                   self.__write(f"</{tag}>\n")
153: (4)               def close(self, id):
154: (8)                   """
155: (8)                   Close open elements, up to (and including) the element identified
156: (8)                   by the given identifier.
157: (8)                   Parameters
158: (8)                   ----------
159: (8)                   id
160: (12)                      Element identifier, as returned by the :meth:`start` method.
161: (8)                   """
162: (8)                   while len(self.__tags) > id:
163: (12)                      self.end()
164: (4)               def element(self, tag, text=None, attrib={}, **extra):
165: (8)                   """
166: (8)                   Add an entire element.  This is the same as calling :meth:`start`,
167: (8)                   :meth:`data`, and :meth:`end` in sequence. The *text* argument can be
```

```
168: (8)                    omitted.
169: (8)                    """
170: (8)                    self.start(tag, attrib, **extra)
171: (8)                    if text:
172: (12)                       self.data(text)
173: (8)                    self.end(indent=False)
174: (4)                def flush(self):
175: (8)                    """Flush the output stream."""
176: (8)                    pass  # replaced by the constructor
177: (0)            def _generate_transform(transform_list):
178: (4)                parts = []
179: (4)                for type, value in transform_list:
180: (8)                    if (type == 'scale' and (value == (1,) or value == (1, 1))
181: (16)                           or type == 'translate' and value == (0, 0)
182: (16)                           or type == 'rotate' and value == (0,)):
183: (12)                       continue
184: (8)                    if type == 'matrix' and isinstance(value, Affine2DBase):
185: (12)                       value = value.to_values()
186: (8)                    parts.append('{}({})'.format(
187: (12)                       type, ' '.join(_short_float_fmt(x) for x in value)))
188: (4)                return ' '.join(parts)
189: (0)            def _generate_css(attrib):
190: (4)                return "; ".join(f"{k}: {v}" for k, v in attrib.items())
191: (0)            _capstyle_d = {'projecting': 'square', 'butt': 'butt', 'round': 'round'}
192: (0)            def _check_is_str(info, key):
193: (4)                if not isinstance(info, str):
194: (8)                    raise TypeError(f'Invalid type for {key} metadata. Expected str, not '
195: (24)                                    f'{type(info)}.')
196: (0)            def _check_is_iterable_of_str(infos, key):
197: (4)                if np.iterable(infos):
198: (8)                    for info in infos:
199: (12)                       if not isinstance(info, str):
200: (16)                           raise TypeError(f'Invalid type for {key} metadata. Expected '
201: (32)                                           f'iterable of str, not {type(info)}.')
202: (4)                else:
203: (8)                    raise TypeError(f'Invalid type for {key} metadata. Expected str or '
204: (24)                                   f'iterable of str, not {type(infos)}.')
205: (0)            class RendererSVG(RendererBase):
206: (4)                def __init__(self, width, height, svgwriter, basename=None, image_dpi=72,
207: (17)                            *, metadata=None):
208: (8)                    self.width = width
209: (8)                    self.height = height
210: (8)                    self.writer = XMLWriter(svgwriter)
211: (8)                    self.image_dpi = image_dpi  # actual dpi at which we rasterize stuff
212: (8)                    if basename is None:
213: (12)                       basename = getattr(svgwriter, "name", "")
214: (12)                       if not isinstance(basename, str):
215: (16)                           basename = ""
216: (8)                    self.basename = basename
217: (8)                    self._groupd = {}
218: (8)                    self._image_counter = itertools.count()
219: (8)                    self._clipd = {}
220: (8)                    self._markers = {}
221: (8)                    self._path_collection_id = 0
222: (8)                    self._hatchd = {}
223: (8)                    self._has_gouraud = False
224: (8)                    self._n_gradients = 0
225: (8)                    super().__init__()
226: (8)                    self._glyph_map = dict()
227: (8)                    str_height = _short_float_fmt(height)
228: (8)                    str_width = _short_float_fmt(width)
229: (8)                    svgwriter.write(svgProlog)
230: (8)                    self._start_id = self.writer.start(
231: (12)                       'svg',
232: (12)                       width=f'{str_width}pt',
233: (12)                       height=f'{str_height}pt',
234: (12)                       viewBox=f'0 0 {str_width} {str_height}',
235: (12)                       xmlns="http://www.w3.org/2000/svg",
236: (12)                       version="1.1",
```

```
(12)                        attrib={'xmlns:xlink': "http://www.w3.org/1999/xlink"})
(8)                     self._write_metadata(metadata)
(8)                     self._write_default_style()
(4)                 def finalize(self):
(8)                     self._write_clips()
(8)                     self._write_hatches()
(8)                     self.writer.close(self._start_id)
(8)                     self.writer.flush()
(4)                 def _write_metadata(self, metadata):
(8)                     if metadata is None:
(12)                        metadata = {}
(8)                     metadata = {
(12)                        'Format': 'image/svg+xml',
(12)                        'Type': 'http://purl.org/dc/dcmitype/StillImage',
(12)                        'Creator':
(16)                            f'Matplotlib v{mpl.__version__}, https://matplotlib.org/',
(12)                        **metadata
(8)                     }
(8)                     writer = self.writer
(8)                     if 'Title' in metadata:
(12)                        title = metadata['Title']
(12)                        _check_is_str(title, 'Title')
(12)                        writer.element('title', text=title)
(8)                     date = metadata.get('Date', None)
(8)                     if date is not None:
(12)                        if isinstance(date, str):
(16)                            dates = [date]
(12)                        elif isinstance(date, (datetime.datetime, datetime.date)):
(16)                            dates = [date.isoformat()]
(12)                        elif np.iterable(date):
(16)                            dates = []
(16)                            for d in date:
(20)                                if isinstance(d, str):
(24)                                    dates.append(d)
(20)                                elif isinstance(d, (datetime.datetime, datetime.date)):
(24)                                    dates.append(d.isoformat())
(20)                                else:
(24)                                    raise TypeError(
(28)                                        f'Invalid type for Date metadata. '
(28)                                        f'Expected iterable of str, date, or datetime, '
(28)                                        f'not {type(d)}.')
(12)                        else:
(16)                            raise TypeError(f'Invalid type for Date metadata. '
(32)                                            f'Expected str, date, datetime, or iterable '
(32)                                            f'of the same, not {type(date)}.')
(12)                        metadata['Date'] = '/'.join(dates)
(8)                     elif 'Date' not in metadata:
(12)                        date = os.getenv("SOURCE_DATE_EPOCH")
(12)                        if date:
(16)                            date = datetime.datetime.fromtimestamp(int(date),
datetime.timezone.utc)
(16)                            metadata['Date'] = date.replace(tzinfo=UTC).isoformat()
(12)                        else:
(16)                            metadata['Date'] = datetime.datetime.today().isoformat()
(8)                     mid = None
(8)                     def ensure_metadata(mid):
(12)                        if mid is not None:
(16)                            return mid
(12)                        mid = writer.start('metadata')
(12)                        writer.start('rdf:RDF', attrib={
(16)                            'xmlns:dc': "http://purl.org/dc/elements/1.1/",
(16)                            'xmlns:cc': "http://creativecommons.org/ns#",
(16)                            'xmlns:rdf': "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
(12)                        })
(12)                        writer.start('cc:Work')
(12)                        return mid
(8)                     uri = metadata.pop('Type', None)
(8)                     if uri is not None:
(12)                        mid = ensure_metadata(mid)
```

```
305: (12)                        writer.element('dc:type', attrib={'rdf:resource': uri})
306: (8)                    for key in ['Title', 'Coverage', 'Date', 'Description', 'Format',
307: (20)                               'Identifier', 'Language', 'Relation', 'Source']:
308: (12)                        info = metadata.pop(key, None)
309: (12)                        if info is not None:
310: (16)                            mid = ensure_metadata(mid)
311: (16)                            _check_is_str(info, key)
312: (16)                            writer.element(f'dc:{key.lower()}', text=info)
313: (8)                    for key in ['Creator', 'Contributor', 'Publisher', 'Rights']:
314: (12)                        agents = metadata.pop(key, None)
315: (12)                        if agents is None:
316: (16)                            continue
317: (12)                        if isinstance(agents, str):
318: (16)                            agents = [agents]
319: (12)                        _check_is_iterable_of_str(agents, key)
320: (12)                        mid = ensure_metadata(mid)
321: (12)                        writer.start(f'dc:{key.lower()}')
322: (12)                        for agent in agents:
323: (16)                            writer.start('cc:Agent')
324: (16)                            writer.element('dc:title', text=agent)
325: (16)                            writer.end('cc:Agent')
326: (12)                        writer.end(f'dc:{key.lower()}')
327: (8)                    keywords = metadata.pop('Keywords', None)
328: (8)                    if keywords is not None:
329: (12)                        if isinstance(keywords, str):
330: (16)                            keywords = [keywords]
331: (12)                        _check_is_iterable_of_str(keywords, 'Keywords')
332: (12)                        mid = ensure_metadata(mid)
333: (12)                        writer.start('dc:subject')
334: (12)                        writer.start('rdf:Bag')
335: (12)                        for keyword in keywords:
336: (16)                            writer.element('rdf:li', text=keyword)
337: (12)                        writer.end('rdf:Bag')
338: (12)                        writer.end('dc:subject')
339: (8)                    if mid is not None:
340: (12)                        writer.close(mid)
341: (8)                    if metadata:
342: (12)                        raise ValueError('Unknown metadata key(s) passed to SVG writer: '
+
343: (29)                                            ','.join(metadata))
344: (4)                def _write_default_style(self):
345: (8)                    writer = self.writer
346: (8)                    default_style = _generate_css({
347: (12)                        'stroke-linejoin': 'round',
348: (12)                        'stroke-linecap': 'butt'})
349: (8)                    writer.start('defs')
350: (8)                    writer.element('style', type='text/css', text='*{%s}' % default_style)
351: (8)                    writer.end('defs')
352: (4)                def _make_id(self, type, content):
353: (8)                    salt = mpl.rcParams['svg.hashsalt']
354: (8)                    if salt is None:
355: (12)                       salt = str(uuid.uuid4())
356: (8)                    m = hashlib.sha256()
357: (8)                    m.update(salt.encode('utf8'))
358: (8)                    m.update(str(content).encode('utf8'))
359: (8)                    return f'{type}{m.hexdigest()[:10]}'
360: (4)                def _make_flip_transform(self, transform):
361: (8)                    return transform + Affine2D().scale(1, -1).translate(0, self.height)
362: (4)                def _get_hatch(self, gc, rgbFace):
363: (8)                    """
364: (8)                    Create a new hatch pattern
365: (8)                    """
366: (8)                    if rgbFace is not None:
367: (12)                       rgbFace = tuple(rgbFace)
368: (8)                    edge = gc.get_hatch_color()
369: (8)                    if edge is not None:
370: (12)                       edge = tuple(edge)
371: (8)                    dictkey = (gc.get_hatch(), rgbFace, edge)
372: (8)                    oid = self._hatchd.get(dictkey)
```

```
(8)                     if oid is None:
(12)                        oid = self._make_id('h', dictkey)
(12)                        self._hatchd[dictkey] = ((gc.get_hatch_path(), rgbFace, edge),
oid)
(8)                     else:
(12)                        _, oid = oid
(8)                     return oid
(4)                 def _write_hatches(self):
(8)                     if not len(self._hatchd):
(12)                        return
(8)                     HATCH_SIZE = 72
(8)                     writer = self.writer
(8)                     writer.start('defs')
(8)                     for (path, face, stroke), oid in self._hatchd.values():
(12)                        writer.start(
(16)                            'pattern',
(16)                            id=oid,
(16)                            patternUnits="userSpaceOnUse",
(16)                            x="0", y="0", width=str(HATCH_SIZE),
(16)                            height=str(HATCH_SIZE))
(12)                        path_data = self._convert_path(
(16)                            path,
(16)                            Affine2D()
(16)                            .scale(HATCH_SIZE).scale(1.0, -1.0).translate(0, HATCH_SIZE),
(16)                            simplify=False)
(12)                        if face is None:
(16)                            fill = 'none'
(12)                        else:
(16)                            fill = rgb2hex(face)
(12)                        writer.element(
(16)                            'rect',
(16)                            x="0", y="0", width=str(HATCH_SIZE+1),
(16)                            height=str(HATCH_SIZE+1),
(16)                            fill=fill)
(12)                        hatch_style = {
(20)                                'fill': rgb2hex(stroke),
(20)                                'stroke': rgb2hex(stroke),
(20)                                'stroke-width': str(mpl.rcParams['hatch.linewidth']),
(20)                                'stroke-linecap': 'butt',
(20)                                'stroke-linejoin': 'miter'
(20)                                }
(12)                        if stroke[3] < 1:
(16)                            hatch_style['stroke-opacity'] = str(stroke[3])
(12)                        writer.element(
(16)                            'path',
(16)                            d=path_data,
(16)                            style=_generate_css(hatch_style)
(16)                            )
(12)                        writer.end('pattern')
(8)                     writer.end('defs')
(4)                 def _get_style_dict(self, gc, rgbFace):
(8)                     """Generate a style string from the GraphicsContext and rgbFace."""
(8)                     attrib = {}
(8)                     forced_alpha = gc.get_forced_alpha()
(8)                     if gc.get_hatch() is not None:
(12)                        attrib['fill'] = f"url(#{self._get_hatch(gc, rgbFace)})"
(12)                        if (rgbFace is not None and len(rgbFace) == 4 and rgbFace[3] !=
1.0
(20)                                and not forced_alpha):
(16)                            attrib['fill-opacity'] = _short_float_fmt(rgbFace[3])
(8)                     else:
(12)                        if rgbFace is None:
(16)                            attrib['fill'] = 'none'
(12)                        else:
(16)                            if tuple(rgbFace[:3]) != (0, 0, 0):
(20)                                attrib['fill'] = rgb2hex(rgbFace)
(16)                            if (len(rgbFace) == 4 and rgbFace[3] != 1.0
(24)                                    and not forced_alpha):
(20)                                attrib['fill-opacity'] = _short_float_fmt(rgbFace[3])
```

```
(8)                     if forced_alpha and gc.get_alpha() != 1.0:
(12)                        attrib['opacity'] = _short_float_fmt(gc.get_alpha())
(8)                     offset, seq = gc.get_dashes()
(8)                     if seq is not None:
(12)                        attrib['stroke-dasharray'] = ','.join(
(16)                            _short_float_fmt(val) for val in seq)
(12)                        attrib['stroke-dashoffset'] = _short_float_fmt(float(offset))
(8)                     linewidth = gc.get_linewidth()
(8)                     if linewidth:
(12)                        rgb = gc.get_rgb()
(12)                        attrib['stroke'] = rgb2hex(rgb)
(12)                        if not forced_alpha and rgb[3] != 1.0:
(16)                            attrib['stroke-opacity'] = _short_float_fmt(rgb[3])
(12)                        if linewidth != 1.0:
(16)                            attrib['stroke-width'] = _short_float_fmt(linewidth)
(12)                        if gc.get_joinstyle() != 'round':
(16)                            attrib['stroke-linejoin'] = gc.get_joinstyle()
(12)                        if gc.get_capstyle() != 'butt':
(16)                            attrib['stroke-linecap'] = _capstyle_d[gc.get_capstyle()]
(8)                     return attrib
(4)                 def _get_style(self, gc, rgbFace):
(8)                     return _generate_css(self._get_style_dict(gc, rgbFace))
(4)                 def _get_clip_attrs(self, gc):
(8)                     cliprect = gc.get_clip_rectangle()
(8)                     clippath, clippath_trans = gc.get_clip_path()
(8)                     if clippath is not None:
(12)                        clippath_trans = self._make_flip_transform(clippath_trans)
(12)                        dictkey = (id(clippath), str(clippath_trans))
(8)                     elif cliprect is not None:
(12)                        x, y, w, h = cliprect.bounds
(12)                        y = self.height-(y+h)
(12)                        dictkey = (x, y, w, h)
(8)                     else:
(12)                        return {}
(8)                     clip = self._clipd.get(dictkey)
(8)                     if clip is None:
(12)                        oid = self._make_id('p', dictkey)
(12)                        if clippath is not None:
(16)                            self._clipd[dictkey] = ((clippath, clippath_trans), oid)
(12)                        else:
(16)                            self._clipd[dictkey] = (dictkey, oid)
(8)                     else:
(12)                        clip, oid = clip
(8)                     return {'clip-path': f'url(#{oid})'}
(4)                 def _write_clips(self):
(8)                     if not len(self._clipd):
(12)                        return
(8)                     writer = self.writer
(8)                     writer.start('defs')
(8)                     for clip, oid in self._clipd.values():
(12)                        writer.start('clipPath', id=oid)
(12)                        if len(clip) == 2:
(16)                            clippath, clippath_trans = clip
(16)                            path_data = self._convert_path(
(20)                                clippath, clippath_trans, simplify=False)
(16)                            writer.element('path', d=path_data)
(12)                        else:
(16)                            x, y, w, h = clip
(16)                            writer.element(
(20)                                'rect',
(20)                                x=_short_float_fmt(x),
(20)                                y=_short_float_fmt(y),
(20)                                width=_short_float_fmt(w),
(20)                                height=_short_float_fmt(h))
(12)                        writer.end('clipPath')
(8)                     writer.end('defs')
(4)                 def open_group(self, s, gid=None):
(8)                     if gid:
(12)                        self.writer.start('g', id=gid)
```

```
(8)                     else:
(12)                        self._groupd[s] = self._groupd.get(s, 0) + 1
(12)                        self.writer.start('g', id=f"{s}_{self._groupd[s]:d}")
(4)                 def close_group(self, s):
(8)                     self.writer.end('g')
(4)                 def option_image_nocomposite(self):
(8)                     return not mpl.rcParams['image.composite_image']
(4)                 def _convert_path(self, path, transform=None, clip=None, simplify=None,
(22)                                  sketch=None):
(8)                     if clip:
(12)                        clip = (0.0, 0.0, self.width, self.height)
(8)                     else:
(12)                        clip = None
(8)                     return _path.convert_to_string(
(12)                        path, transform, clip, simplify, sketch, 6,
(12)                        [b'M', b'L', b'Q', b'C', b'z'], False).decode('ascii')
(4)                 def draw_path(self, gc, path, transform, rgbFace=None):
(8)                     trans_and_flip = self._make_flip_transform(transform)
(8)                     clip = (rgbFace is None and gc.get_hatch_path() is None)
(8)                     simplify = path.should_simplify and clip
(8)                     path_data = self._convert_path(
(12)                        path, trans_and_flip, clip=clip, simplify=simplify,
(12)                        sketch=gc.get_sketch_params())
(8)                     if gc.get_url() is not None:
(12)                        self.writer.start('a', {'xlink:href': gc.get_url()})
(8)                     self.writer.element('path', d=path_data, **self._get_clip_attrs(gc),
(28)                                        style=self._get_style(gc, rgbFace))
(8)                     if gc.get_url() is not None:
(12)                        self.writer.end('a')
(4)                 def draw_markers(
(12)                        self, gc, marker_path, marker_trans, path, trans, rgbFace=None):
(8)                     if not len(path.vertices):
(12)                        return
(8)                     writer = self.writer
(8)                     path_data = self._convert_path(
(12)                        marker_path,
(12)                        marker_trans + Affine2D().scale(1.0, -1.0),
(12)                        simplify=False)
(8)                     style = self._get_style_dict(gc, rgbFace)
(8)                     dictkey = (path_data, _generate_css(style))
(8)                     oid = self._markers.get(dictkey)
(8)                     style = _generate_css({k: v for k, v in style.items()
(30)                                           if k.startswith('stroke')})
(8)                     if oid is None:
(12)                        oid = self._make_id('m', dictkey)
(12)                        writer.start('defs')
(12)                        writer.element('path', id=oid, d=path_data, style=style)
(12)                        writer.end('defs')
(12)                        self._markers[dictkey] = oid
(8)                     writer.start('g', **self._get_clip_attrs(gc))
(8)                     trans_and_flip = self._make_flip_transform(trans)
(8)                     attrib = {'xlink:href': f'#{oid}'}
(8)                     clip = (0, 0, self.width*72, self.height*72)
(8)                     for vertices, code in path.iter_segments(
(16)                            trans_and_flip, clip=clip, simplify=False):
(12)                        if len(vertices):
(16)                            x, y = vertices[-2:]
(16)                            attrib['x'] = _short_float_fmt(x)
(16)                            attrib['y'] = _short_float_fmt(y)
(16)                            attrib['style'] = self._get_style(gc, rgbFace)
(16)                            writer.element('use', attrib=attrib)
(8)                     writer.end('g')
(4)                 def draw_path_collection(self, gc, master_transform, paths,
all_transforms,
(29)                                         offsets, offset_trans, facecolors, edgecolors,
(29)                                         linewidths, linestyles, antialiaseds, urls,
(29)                                         offset_position):
(8)                     len_path = len(paths[0].vertices) if len(paths) > 0 else 0
(8)                     uses_per_path = self._iter_collection_uses_per_path(
```

```
(12)                        paths, all_transforms, offsets, facecolors, edgecolors)
(8)                     should_do_optimization = \
(12)                        len_path + 9 * uses_per_path + 3 < (len_path + 5) * uses_per_path
(8)                     if not should_do_optimization:
(12)                        return super().draw_path_collection(
(16)                            gc, master_transform, paths, all_transforms,
(16)                            offsets, offset_trans, facecolors, edgecolors,
(16)                            linewidths, linestyles, antialiaseds, urls,
(16)                            offset_position)
(8)                     writer = self.writer
(8)                     path_codes = []
(8)                     writer.start('defs')
(8)                     for i, (path, transform) in enumerate(self._iter_collection_raw_paths(
(16)                            master_transform, paths, all_transforms)):
(12)                        transform = Affine2D(transform.get_matrix()).scale(1.0, -1.0)
(12)                        d = self._convert_path(path, transform, simplify=False)
(12)                        oid = 'C{:x}_{:x}_{}'.format(
(16)                            self._path_collection_id, i, self._make_id('', d))
(12)                        writer.element('path', id=oid, d=d)
(12)                        path_codes.append(oid)
(8)                     writer.end('defs')
(8)                     for xo, yo, path_id, gc0, rgbFace in self._iter_collection(
(16)                            gc, path_codes, offsets, offset_trans,
(16)                            facecolors, edgecolors, linewidths, linestyles,
(16)                            antialiaseds, urls, offset_position):
(12)                        url = gc0.get_url()
(12)                        if url is not None:
(16)                            writer.start('a', attrib={'xlink:href': url})
(12)                        clip_attrs = self._get_clip_attrs(gc0)
(12)                        if clip_attrs:
(16)                            writer.start('g', **clip_attrs)
(12)                        attrib = {
(16)                            'xlink:href': f'#{path_id}',
(16)                            'x': _short_float_fmt(xo),
(16)                            'y': _short_float_fmt(self.height - yo),
(16)                            'style': self._get_style(gc0, rgbFace)
(16)                            }
(12)                        writer.element('use', attrib=attrib)
(12)                        if clip_attrs:
(16)                            writer.end('g')
(12)                        if url is not None:
(16)                            writer.end('a')
(8)                     self._path_collection_id += 1
(4)                 def _draw_gouraud_triangle(self, transformed_points, colors):
(8)                     avg_color = np.average(colors, axis=0)
(8)                     if avg_color[-1] == 0:
(12)                        return
(8)                     writer = self.writer
(8)                     writer.start('defs')
(8)                     for i in range(3):
(12)                        x1, y1 = transformed_points[i]
(12)                        x2, y2 = transformed_points[(i + 1) % 3]
(12)                        x3, y3 = transformed_points[(i + 2) % 3]
(12)                        rgba_color = colors[i]
(12)                        if x2 == x3:
(16)                            xb = x2
(16)                            yb = y1
(12)                        elif y2 == y3:
(16)                            xb = x1
(16)                            yb = y2
(12)                        else:
(16)                            m1 = (y2 - y3) / (x2 - x3)
(16)                            b1 = y2 - (m1 * x2)
(16)                            m2 = -(1.0 / m1)
(16)                            b2 = y1 - (m2 * x1)
(16)                            xb = (-b1 + b2) / (m1 - m2)
(16)                            yb = m2 * xb + b2
(12)                        writer.start(
(16)                            'linearGradient',
```

```
(16)                            id=f"GR{self._n_gradients:x}_{i:d}",
(16)                            gradientUnits="userSpaceOnUse",
(16)                            x1=_short_float_fmt(x1), y1=_short_float_fmt(y1),
(16)                            x2=_short_float_fmt(xb), y2=_short_float_fmt(yb))
(12)                        writer.element(
(16)                            'stop',
(16)                            offset='1',
(16)                            style=_generate_css({
(20)                                'stop-color': rgb2hex(avg_color),
(20)                                'stop-opacity': _short_float_fmt(rgba_color[-1])}))
(12)                        writer.element(
(16)                            'stop',
(16)                            offset='0',
(16)                            style=_generate_css({'stop-color': rgb2hex(rgba_color),
(36)                                                 'stop-opacity': "0"}))
(12)                        writer.end('linearGradient')
(8)                     writer.end('defs')
(8)                     dpath = (f"M {_short_float_fmt(x1)},{_short_float_fmt(y1)}"
(17)                             f" L {_short_float_fmt(x2)},{_short_float_fmt(y2)}"
(17)                             f" {_short_float_fmt(x3)},{_short_float_fmt(y3)} Z")
(8)                     writer.element(
(12)                        'path',
(12)                        attrib={'d': dpath,
(20)                                'fill': rgb2hex(avg_color),
(20)                                'fill-opacity': '1',
(20)                                'shape-rendering': "crispEdges"})
(8)                     writer.start(
(16)                            'g',
(16)                            attrib={'stroke': "none",
(24)                                    'stroke-width': "0",
(24)                                    'shape-rendering': "crispEdges",
(24)                                    'filter': "url(#colorMat)"})
(8)                     writer.element(
(12)                        'path',
(12)                        attrib={'d': dpath,
(20)                                'fill': f'url(#GR{self._n_gradients:x}_0)',
(20)                                'shape-rendering': "crispEdges"})
(8)                     writer.element(
(12)                        'path',
(12)                        attrib={'d': dpath,
(20)                                'fill': f'url(#GR{self._n_gradients:x}_1)',
(20)                                'filter': 'url(#colorAdd)',
(20)                                'shape-rendering': "crispEdges"})
(8)                     writer.element(
(12)                        'path',
(12)                        attrib={'d': dpath,
(20)                                'fill': f'url(#GR{self._n_gradients:x}_2)',
(20)                                'filter': 'url(#colorAdd)',
(20)                                'shape-rendering': "crispEdges"})
(8)                     writer.end('g')
(8)                     self._n_gradients += 1
(4)                 def draw_gouraud_triangles(self, gc, triangles_array, colors_array,
(31)                                           transform):
(8)                     writer = self.writer
(8)                     writer.start('g', **self._get_clip_attrs(gc))
(8)                     transform = transform.frozen()
(8)                     trans_and_flip = self._make_flip_transform(transform)
(8)                     if not self._has_gouraud:
(12)                        self._has_gouraud = True
(12)                        writer.start(
(16)                            'filter',
(16)                            id='colorAdd')
(12)                        writer.element(
(16)                            'feComposite',
(16)                            attrib={'in': 'SourceGraphic'},
(16)                            in2='BackgroundImage',
(16)                            operator='arithmetic',
(16)                            k2="1", k3="1")
(12)                        writer.end('filter')
```

```
715: (12)                        writer.start(
716: (16)                            'filter',
717: (16)                            id='colorMat')
718: (12)                        writer.element(
719: (16)                            'feColorMatrix',
720: (16)                            attrib={'type': 'matrix'},
721: (16)                            values='1 0 0 0 0 \n0 1 0 0 0 \n0 0 1 0 0 \n1 1 1 1 0 \n0 0 0
0 1 ')
722: (12)                        writer.end('filter')
723: (8)                    for points, colors in zip(triangles_array, colors_array):
724: (12)                        self._draw_gouraud_triangle(trans_and_flip.transform(points),
colors)
725: (8)                    writer.end('g')
726: (4)                def option_scale_image(self):
727: (8)                    return True
728: (4)                def get_image_magnification(self):
729: (8)                    return self.image_dpi / 72.0
730: (4)                def draw_image(self, gc, x, y, im, transform=None):
731: (8)                    h, w = im.shape[:2]
732: (8)                    if w == 0 or h == 0:
733: (12)                       return
734: (8)                    clip_attrs = self._get_clip_attrs(gc)
735: (8)                    if clip_attrs:
736: (12)                       self.writer.start('g', **clip_attrs)
737: (8)                    url = gc.get_url()
738: (8)                    if url is not None:
739: (12)                       self.writer.start('a', attrib={'xlink:href': url})
740: (8)                    attrib = {}
741: (8)                    oid = gc.get_gid()
742: (8)                    if mpl.rcParams['svg.image_inline']:
743: (12)                       buf = BytesIO()
744: (12)                       Image.fromarray(im).save(buf, format="png")
745: (12)                       oid = oid or self._make_id('image', buf.getvalue())
746: (12)                       attrib['xlink:href'] = (
747: (16)                           "\n" +
748: (16)                           base64.b64encode(buf.getvalue()).decode('ascii'))
749: (8)                    else:
750: (12)                       if self.basename is None:
751: (16)                           raise ValueError("Cannot save image data to filesystem when "
752: (33)                                            "writing SVG to an in-memory buffer")
753: (12)                       filename = f'{self.basename}.image{next(self._image_counter)}.png'
754: (12)                       _log.info('Writing image file for inclusion: %s', filename)
755: (12)                       Image.fromarray(im).save(filename)
756: (12)                       oid = oid or 'Im_' + self._make_id('image', filename)
757: (12)                       attrib['xlink:href'] = filename
758: (8)                    attrib['id'] = oid
759: (8)                    if transform is None:
760: (12)                       w = 72.0 * w / self.image_dpi
761: (12)                       h = 72.0 * h / self.image_dpi
762: (12)                       self.writer.element(
763: (16)                           'image',
764: (16)                           transform=_generate_transform([
765: (20)                               ('scale', (1, -1)), ('translate', (0, -h))]),
766: (16)                           x=_short_float_fmt(x),
767: (16)                           y=_short_float_fmt(-(self.height - y - h)),
768: (16)                           width=_short_float_fmt(w), height=_short_float_fmt(h),
769: (16)                           attrib=attrib)
770: (8)                    else:
771: (12)                       alpha = gc.get_alpha()
772: (12)                       if alpha != 1.0:
773: (16)                           attrib['opacity'] = _short_float_fmt(alpha)
774: (12)                       flipped = (
775: (16)                           Affine2D().scale(1.0 / w, 1.0 / h) +
776: (16)                           transform +
777: (16)                           Affine2D()
778: (16)                           .translate(x, y)
779: (16)                           .scale(1.0, -1.0)
780: (16)                           .translate(0.0, self.height))
781: (12)                       attrib['transform'] = _generate_transform(
```

```
782: (16)                            [('matrix', flipped.frozen())])
783: (12)                        attrib['style'] = (
784: (16)                            'image-rendering:crisp-edges;'
785: (16)                            'image-rendering:pixelated')
786: (12)                        self.writer.element(
787: (16)                            'image',
788: (16)                            width=_short_float_fmt(w), height=_short_float_fmt(h),
789: (16)                            attrib=attrib)
790: (8)                    if url is not None:
791: (12)                        self.writer.end('a')
792: (8)                    if clip_attrs:
793: (12)                        self.writer.end('g')
794: (4)                def _update_glyph_map_defs(self, glyph_map_new):
795: (8)                    """
796: (8)                    Emit definitions for not-yet-defined glyphs, and record them as having
797: (8)                    been defined.
798: (8)                    """
799: (8)                    writer = self.writer
800: (8)                    if glyph_map_new:
801: (12)                        writer.start('defs')
802: (12)                        for char_id, (vertices, codes) in glyph_map_new.items():
803: (16)                            char_id = self._adjust_char_id(char_id)
804: (16)                            path_data = self._convert_path(
805: (20)                                Path(vertices * 64, codes), simplify=False)
806: (16)                            writer.element(
807: (20)                                'path', id=char_id, d=path_data,
808: (20)                                transform=_generate_transform([('scale', (1 / 64,))]))
809: (12)                        writer.end('defs')
810: (12)                        self._glyph_map.update(glyph_map_new)
811: (4)                def _adjust_char_id(self, char_id):
812: (8)                    return char_id.replace("%20", "_")
813: (4)                def _draw_text_as_path(self, gc, x, y, s, prop, angle, ismath,
mtext=None):
814: (8)                    writer = self.writer
815: (8)                    writer.comment(s)
816: (8)                    glyph_map = self._glyph_map
817: (8)                    text2path = self._text2path
818: (8)                    color = rgb2hex(gc.get_rgb())
819: (8)                    fontsize = prop.get_size_in_points()
820: (8)                    style = {}
821: (8)                    if color != '#000000':
822: (12)                       style['fill'] = color
823: (8)                    alpha = gc.get_alpha() if gc.get_forced_alpha() else gc.get_rgb()[3]
824: (8)                    if alpha != 1:
825: (12)                       style['opacity'] = _short_float_fmt(alpha)
826: (8)                    font_scale = fontsize / text2path.FONT_SCALE
827: (8)                    attrib = {
828: (12)                       'style': _generate_css(style),
829: (12)                       'transform': _generate_transform([
830: (16)                           ('translate', (x, y)),
831: (16)                           ('rotate', (-angle,)),
832: (16)                           ('scale', (font_scale, -font_scale))]),
833: (8)                    }
834: (8)                    writer.start('g', attrib=attrib)
835: (8)                    if not ismath:
836: (12)                       font = text2path._get_font(prop)
837: (12)                       _glyphs = text2path.get_glyphs_with_font(
838: (16)                           font, s, glyph_map=glyph_map, return_new_glyphs_only=True)
839: (12)                       glyph_info, glyph_map_new, rects = _glyphs
840: (12)                       self._update_glyph_map_defs(glyph_map_new)
841: (12)                       for glyph_id, xposition, yposition, scale in glyph_info:
842: (16)                           attrib = {'xlink:href': f'#{glyph_id}'}
843: (16)                           if xposition != 0.0:
844: (20)                               attrib['x'] = _short_float_fmt(xposition)
845: (16)                           if yposition != 0.0:
846: (20)                               attrib['y'] = _short_float_fmt(yposition)
847: (16)                           writer.element('use', attrib=attrib)
848: (8)                    else:
849: (12)                       if ismath == "TeX":
```

```
850: (16)                            _glyphs = text2path.get_glyphs_tex(
851: (20)                                prop, s, glyph_map=glyph_map, return_new_glyphs_only=True)
852: (12)                        else:
853: (16)                            _glyphs = text2path.get_glyphs_mathtext(
854: (20)                                prop, s, glyph_map=glyph_map, return_new_glyphs_only=True)
855: (12)                        glyph_info, glyph_map_new, rects = _glyphs
856: (12)                        self._update_glyph_map_defs(glyph_map_new)
857: (12)                        for char_id, xposition, yposition, scale in glyph_info:
858: (16)                            char_id = self._adjust_char_id(char_id)
859: (16)                            writer.element(
860: (20)                                'use',
861: (20)                                transform=_generate_transform([
862: (24)                                    ('translate', (xposition, yposition)),
863: (24)                                    ('scale', (scale,)),
864: (24)                                    ]),
865: (20)                                attrib={'xlink:href': f'#{char_id}'})
866: (12)                        for verts, codes in rects:
867: (16)                            path = Path(verts, codes)
868: (16)                            path_data = self._convert_path(path, simplify=False)
869: (16)                            writer.element('path', d=path_data)
870: (8)                    writer.end('g')
871: (4)                def _draw_text_as_text(self, gc, x, y, s, prop, angle, ismath,
mtext=None):
872: (8)                    writer = self.writer
873: (8)                    color = rgb2hex(gc.get_rgb())
874: (8)                    style = {}
875: (8)                    if color != '#000000':
876: (12)                       style['fill'] = color
877: (8)                    alpha = gc.get_alpha() if gc.get_forced_alpha() else gc.get_rgb()[3]
878: (8)                    if alpha != 1:
879: (12)                       style['opacity'] = _short_float_fmt(alpha)
880: (8)                    if not ismath:
881: (12)                       attrib = {}
882: (12)                       font_parts = []
883: (12)                       if prop.get_style() != 'normal':
884: (16)                           font_parts.append(prop.get_style())
885: (12)                       if prop.get_variant() != 'normal':
886: (16)                           font_parts.append(prop.get_variant())
887: (12)                       weight = fm.weight_dict[prop.get_weight()]
888: (12)                       if weight != 400:
889: (16)                           font_parts.append(f'{weight}')
890: (12)                       def _normalize_sans(name):
891: (16)                           return 'sans-serif' if name in ['sans', 'sans serif'] else
name
892: (12)                       def _expand_family_entry(fn):
893: (16)                           fn = _normalize_sans(fn)
894: (16)                           if fn in fm.font_family_aliases:
895: (20)                               for name in fm.FontManager._expand_aliases(fn):
896: (24)                                   yield _normalize_sans(name)
897: (16)                           yield fn
898: (12)                       def _get_all_quoted_names(prop):
899: (16)                           return [name if name in fm.font_family_aliases else repr(name)
900: (24)                                   for entry in prop.get_family()
901: (24)                                   for name in _expand_family_entry(entry)]
902: (12)                       font_parts.extend([
903: (16)                           f'{_short_float_fmt(prop.get_size())}px',
904: (16)                           ", ".join(dict.fromkeys(_get_all_quoted_names(prop)))
905: (12)                       ])
906: (12)                       style['font'] = ' '.join(font_parts)
907: (12)                       if prop.get_stretch() != 'normal':
908: (16)                           style['font-stretch'] = prop.get_stretch()
909: (12)                       attrib['style'] = _generate_css(style)
910: (12)                       if mtext and (angle == 0 or mtext.get_rotation_mode() ==
"anchor"):
911: (16)                           transform = mtext.get_transform()
912: (16)                           ax, ay = transform.transform(mtext.get_unitless_position())
913: (16)                           ay = self.height - ay
914: (16)                           angle_rad = np.deg2rad(angle)
915: (16)                           dir_vert = np.array([np.sin(angle_rad), np.cos(angle_rad)])
```

```
916: (16)                            v_offset = np.dot(dir_vert, [(x - ax), (y - ay)])
917: (16)                            ax = ax + v_offset * dir_vert[0]
918: (16)                            ay = ay + v_offset * dir_vert[1]
919: (16)                            ha_mpl_to_svg = {'left': 'start', 'right': 'end',
920: (33)                                             'center': 'middle'}
921: (16)                            style['text-anchor'] = ha_mpl_to_svg[mtext.get_ha()]
922: (16)                            attrib['x'] = _short_float_fmt(ax)
923: (16)                            attrib['y'] = _short_float_fmt(ay)
924: (16)                            attrib['style'] = _generate_css(style)
925: (16)                            attrib['transform'] = _generate_transform([
926: (20)                                ("rotate", (-angle, ax, ay))])
927: (12)                        else:
928: (16)                            attrib['transform'] = _generate_transform([
929: (20)                                ('translate', (x, y)),
930: (20)                                ('rotate', (-angle,))])
931: (12)                        writer.element('text', s, attrib=attrib)
932: (8)                    else:
933: (12)                        writer.comment(s)
934: (12)                        width, height, descent, glyphs, rects = \
935: (16)                            self._text2path.mathtext_parser.parse(s, 72, prop)
936: (12)                        writer.start('g',
937: (25)                                     style=_generate_css(style),
938: (25)                                     transform=_generate_transform([
939: (29)                                         ('translate', (x, y)),
940: (29)                                         ('rotate', (-angle,))]),
941: (25)                                     )
942: (12)                        writer.start('text')
943: (12)                        spans = {}
944: (12)                        for font, fontsize, thetext, new_x, new_y in glyphs:
945: (16)                            entry = fm.ttfFontProperty(font)
946: (16)                            font_parts = []
947: (16)                            if entry.style != 'normal':
948: (20)                                font_parts.append(entry.style)
949: (16)                            if entry.variant != 'normal':
950: (20)                                font_parts.append(entry.variant)
951: (16)                            if entry.weight != 400:
952: (20)                                font_parts.append(f'{entry.weight}')
953: (16)                            font_parts.extend([
954: (20)                                f'{_short_float_fmt(fontsize)}px',
955: (20)                                f'{entry.name!r}',  # ensure quoting
956: (16)                            ])
957: (16)                            style = {'font': ' '.join(font_parts)}
958: (16)                            if entry.stretch != 'normal':
959: (20)                                style['font-stretch'] = entry.stretch
960: (16)                            style = _generate_css(style)
961: (16)                            if thetext == 32:
962: (20)                                thetext = 0xa0  # non-breaking space
963: (16)                            spans.setdefault(style, []).append((new_x, -new_y, thetext))
964: (12)                        for style, chars in spans.items():
965: (16)                            chars.sort()
966: (16)                            if len({y for x, y, t in chars}) == 1:  # Are all y's the
same?
967: (20)                                ys = str(chars[0][1])
968: (16)                            else:
969: (20)                                ys = ' '.join(str(c[1]) for c in chars)
970: (16)                            attrib = {
971: (20)                                'style': style,
972: (20)                                'x': ' '.join(_short_float_fmt(c[0]) for c in chars),
973: (20)                                'y': ys
974: (20)                                }
975: (16)                            writer.element(
976: (20)                                'tspan',
977: (20)                                ''.join(chr(c[2]) for c in chars),
978: (20)                                attrib=attrib)
979: (12)                        writer.end('text')
980: (12)                        for x, y, width, height in rects:
981: (16)                            writer.element(
982: (20)                                'rect',
983: (20)                                x=_short_float_fmt(x),
```

```
(20)                                y=_short_float_fmt(-y-1),
(20)                                width=_short_float_fmt(width),
(20)                                height=_short_float_fmt(height)
(20)                                )
(12)                        writer.end('g')
(4)                 def draw_text(self, gc, x, y, s, prop, angle, ismath=False, mtext=None):
(8)                     clip_attrs = self._get_clip_attrs(gc)
(8)                     if clip_attrs:
(12)                        self.writer.start('g', **clip_attrs)
(8)                     if gc.get_url() is not None:
(12)                        self.writer.start('a', {'xlink:href': gc.get_url()})
(8)                     if mpl.rcParams['svg.fonttype'] == 'path':
(12)                        self._draw_text_as_path(gc, x, y, s, prop, angle, ismath, mtext)
(8)                     else:
(12)                        self._draw_text_as_text(gc, x, y, s, prop, angle, ismath, mtext)
(8)                     if gc.get_url() is not None:
(12)                        self.writer.end('a')
(8)                     if clip_attrs:
(12)                        self.writer.end('g')
(4)                 def flipy(self):
(8)                     return True
(4)                 def get_canvas_width_height(self):
(8)                     return self.width, self.height
(4)                 def get_text_width_height_descent(self, s, prop, ismath):
(8)                     return self._text2path.get_text_width_height_descent(s, prop, ismath)
(0)             class FigureCanvasSVG(FigureCanvasBase):
(4)                 filetypes = {'svg': 'Scalable Vector Graphics',
(17)                             'svgz': 'Scalable Vector Graphics'}
(4)                 fixed_dpi = 72
(4)                 def print_svg(self, filename, *, bbox_inches_restore=None, metadata=None):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     filename : str or path-like or file-like
(12)                        Output target; if a string, a file will be opened for writing.
(8)                     metadata : dict[str, Any], optional
(12)                        Metadata in the SVG file defined as key-value pairs of strings,
(12)                        datetimes, or lists of strings, e.g., ``{'Creator': 'My software',
(12)                        'Contributor': ['Me', 'My Friend'], 'Title': 'Awesome'}``.
(12)                        The standard keys and their value types are:
(12)                        * *str*: ``'Coverage''``, ``'Description''``, ``'Format''``,
(14)                          ``'Identifier''``, ``'Language''``, ``'Relation''``, ``'Source''``,
(14)                          ``'Title''``, and ``'Type''``.
(12)                        * *str* or *list of str*: ``'Contributor''``, ``'Creator''``,
(14)                          ``'Keywords''``, ``'Publisher''``, and ``'Rights''``.
(12)                        * *str*, *date*, *datetime*, or *tuple* of same: ``'Date''``. If a
(14)                          non-*str*, then it will be formatted as ISO 8601.
(12)                        Values have been predefined for ``'Creator''``, ``'Date''``,
(12)                        ``'Format''``, and ``'Type''``. They can be removed by setting them
(12)                        to `None`.
(12)                        Information is encoded as `Dublin Core Metadata`__.
(12)                        .. _DC: https://www.dublincore.org/specifications/dublin-core/
(12)                        __ DC_
(8)                     """
(8)                     with cbook.open_file_cm(filename, "w", encoding="utf-8") as fh:
(12)                        if not cbook.file_requires_unicode(fh):
(16)                            fh = codecs.getwriter('utf-8')(fh)
(12)                        dpi = self.figure.dpi
(12)                        self.figure.dpi = 72
(12)                        width, height = self.figure.get_size_inches()
(12)                        w, h = width * 72, height * 72
(12)                        renderer = MixedModeRenderer(
(16)                            self.figure, width, height, dpi,
(16)                            RendererSVG(w, h, fh, image_dpi=dpi, metadata=metadata),
(16)                            bbox_inches_restore=bbox_inches_restore)
(12)                        self.figure.draw(renderer)
(12)                        renderer.finalize()
(4)                 def print_svgz(self, filename, **kwargs):
(8)                     with cbook.open_file_cm(filename, "wb") as fh, \
```

```
1053: (16)                          gzip.GzipFile(mode='w', fileobj=fh) as gzipwriter:
1054: (12)                      return self.print_svg(gzipwriter, **kwargs)
1055: (4)               def get_default_filetype(self):
1056: (8)                   return 'svg'
1057: (4)               def draw(self):
1058: (8)                   self.figure.draw_without_rendering()
1059: (8)                   return super().draw()
1060: (0)           FigureManagerSVG = FigureManagerBase
1061: (0)           svgProlog = """\
1062: (0)           <?xml version="1.0" encoding="utf-8" standalone="no"?>
1063: (0)           <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
1064: (2)             "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
1065: (0)           """
1066: (0)           @_Backend.export
1067: (0)           class _BackendSVG(_Backend):
1068: (4)               backend_version = mpl.__version__
1069: (4)               FigureCanvas = FigureCanvasSVG
```

File 102 - backend_template.py:

```
1: (0)              """
2: (0)              A fully functional, do-nothing backend intended as a template for backend
3: (0)              writers.  It is fully functional in that you can select it as a backend e.g.
4: (0)              with ::
5: (4)                  import matplotlib
6: (4)                  matplotlib.use("template")
7: (0)              and your program will (should!) run without error, though no output is
8: (0)              produced.  This provides a starting point for backend writers; you can
9: (0)              selectively implement drawing methods (`~.RendererTemplate.draw_path`,
10: (0)             `~.RendererTemplate.draw_image`, etc.) and slowly see your figure come to life
11: (0)             instead having to have a full-blown implementation before getting any results.
12: (0)             Copy this file to a directory outside the Matplotlib source tree, somewhere
13: (0)             where Python can import it (by adding the directory to your ``sys.path`` or by
14: (0)             packaging it as a normal Python package); if the backend is importable as
15: (0)             ``import my.backend`` you can then select it using ::
16: (4)                 import matplotlib
17: (4)                 matplotlib.use("module://my.backend")
18: (0)             If your backend implements support for saving figures (i.e. has a `print_xyz`
19: (0)             method), you can register it as the default handler for a given file type::
20: (4)                 from matplotlib.backend_bases import register_backend
21: (4)                 register_backend('xyz', 'my_backend', 'XYZ File Format')
22: (4)                 ...
23: (4)                 plt.savefig("figure.xyz")
24: (0)             """
25: (0)             from matplotlib import _api
26: (0)             from matplotlib._pylab_helpers import Gcf
27: (0)             from matplotlib.backend_bases import (
28: (5)                  FigureCanvasBase, FigureManagerBase, GraphicsContextBase, RendererBase)
29: (0)             from matplotlib.figure import Figure
30: (0)             class RendererTemplate(RendererBase):
31: (4)                 """
32: (4)                 The renderer handles drawing/rendering operations.
33: (4)                 This is a minimal do-nothing class that can be used to get started when
34: (4)                 writing a new backend.  Refer to `.backend_bases.RendererBase` for
35: (4)                 documentation of the methods.
36: (4)                 """
37: (4)                 def __init__(self, dpi):
38: (8)                     super().__init__()
39: (8)                     self.dpi = dpi
40: (4)                 def draw_path(self, gc, path, transform, rgbFace=None):
41: (8)                     pass
42: (4)                 def draw_image(self, gc, x, y, im):
43: (8)                     pass
44: (4)                 def draw_text(self, gc, x, y, s, prop, angle, ismath=False, mtext=None):
45: (8)                     pass
46: (4)                 def flipy(self):
47: (8)                     return True
```

```
48: (4)                 def get_canvas_width_height(self):
49: (8)                     return 100, 100
50: (4)                 def get_text_width_height_descent(self, s, prop, ismath):
51: (8)                     return 1, 1, 1
52: (4)                 def new_gc(self):
53: (8)                     return GraphicsContextTemplate()
54: (4)                 def points_to_pixels(self, points):
55: (8)                     return points
56: (0)             class GraphicsContextTemplate(GraphicsContextBase):
57: (4)                 """
58: (4)                 The graphics context provides the color, line styles, etc.  See the cairo
59: (4)                 and postscript backends for examples of mapping the graphics context
60: (4)                 attributes (cap styles, join styles, line widths, colors) to a particular
61: (4)                 backend.  In cairo this is done by wrapping a cairo.Context object and
62: (4)                 forwarding the appropriate calls to it using a dictionary mapping styles
63: (4)                 to gdk constants.  In Postscript, all the work is done by the renderer,
64: (4)                 mapping line styles to postscript calls.
65: (4)                 If it's more appropriate to do the mapping at the renderer level (as in
66: (4)                 the postscript backend), you don't need to override any of the GC methods.
67: (4)                 If it's more appropriate to wrap an instance (as in the cairo backend) and
68: (4)                 do the mapping here, you'll need to override several of the setter
69: (4)                 methods.
70: (4)                 The base GraphicsContext stores colors as an RGB tuple on the unit
71: (4)                 interval, e.g., (0.5, 0.0, 1.0). You may need to map this to colors
72: (4)                 appropriate for your backend.
73: (4)                 """
74: (0)             class FigureManagerTemplate(FigureManagerBase):
75: (4)                 """
76: (4)                 Helper class for pyplot mode, wraps everything up into a neat bundle.
77: (4)                 For non-interactive backends, the base class is sufficient.  For
78: (4)                 interactive backends, see the documentation of the `.FigureManagerBase`
79: (4)                 class for the list of methods that can/should be overridden.
80: (4)                 """
81: (0)             class FigureCanvasTemplate(FigureCanvasBase):
82: (4)                 """
83: (4)                 The canvas the figure renders into.  Calls the draw and print fig
84: (4)                 methods, creates the renderers, etc.
85: (4)                 Note: GUI templates will want to connect events for button presses,
86: (4)                 mouse movements and key presses to functions that call the base
87: (4)                 class methods button_press_event, button_release_event,
88: (4)                 motion_notify_event, key_press_event, and key_release_event.  See the
89: (4)                 implementations of the interactive backends for examples.
90: (4)                 Attributes
91: (4)                 ----------
92: (4)                 figure : `~matplotlib.figure.Figure`
93: (8)                     A high-level Figure instance
94: (4)                 """
95: (4)                 manager_class = FigureManagerTemplate
96: (4)                 def draw(self):
97: (8)                     """
98: (8)                     Draw the figure using the renderer.
99: (8)                     It is important that this method actually walk the artist tree
100: (8)                    even if not output is produced because this will trigger
101: (8)                    deferred work (like computing limits auto-limits and tick
102: (8)                    values) that users may want access to before saving to disk.
103: (8)                    """
104: (8)                    renderer = RendererTemplate(self.figure.dpi)
105: (8)                    self.figure.draw(renderer)
106: (4)                filetypes = {**FigureCanvasBase.filetypes, 'foo': 'My magic Foo format'}
107: (4)                def print_foo(self, filename, **kwargs):
108: (8)                    """
109: (8)                    Write out format foo.
110: (8)                    This method is normally called via `.Figure.savefig` and
111: (8)                    `.FigureCanvasBase.print_figure`, which take care of setting the
figure
112: (8)                    facecolor, edgecolor, and dpi to the desired output values, and will
113: (8)                    restore them to the original values.  Therefore, `print_foo` does not
114: (8)                    need to handle these settings.
115: (8)                    """
```

```
116: (8)                    self.draw()
117: (4)                def get_default_filetype(self):
118: (8)                    return 'foo'
119: (0)            FigureCanvas = FigureCanvasTemplate
120: (0)            FigureManager = FigureManagerTemplate
```

File 103 - _backend_tk.py:

```
1: (0)              import uuid
2: (0)              import weakref
3: (0)              from contextlib import contextmanager
4: (0)              import logging
5: (0)              import math
6: (0)              import os.path
7: (0)              import pathlib
8: (0)              import sys
9: (0)              import tkinter as tk
10: (0)             import tkinter.filedialog
11: (0)             import tkinter.font
12: (0)             import tkinter.messagebox
13: (0)             from tkinter.simpledialog import SimpleDialog
14: (0)             import numpy as np
15: (0)             from PIL import Image, ImageTk
16: (0)             import matplotlib as mpl
17: (0)             from matplotlib import _api, backend_tools, cbook, _c_internal_utils
18: (0)             from matplotlib.backend_bases import (
19: (4)                 _Backend, FigureCanvasBase, FigureManagerBase, NavigationToolbar2,
20: (4)                 TimerBase, ToolContainerBase, cursors, _Mode,
21: (4)                 CloseEvent, KeyEvent, LocationEvent, MouseEvent, ResizeEvent)
22: (0)             from matplotlib._pylab_helpers import Gcf
23: (0)             from . import _tkagg
24: (0)             from ._tkagg import TK_PHOTO_COMPOSITE_OVERLAY, TK_PHOTO_COMPOSITE_SET
25: (0)             _log = logging.getLogger(__name__)
26: (0)             cursord = {
27: (4)                 cursors.MOVE: "fleur",
28: (4)                 cursors.HAND: "hand2",
29: (4)                 cursors.POINTER: "arrow",
30: (4)                 cursors.SELECT_REGION: "crosshair",
31: (4)                 cursors.WAIT: "watch",
32: (4)                 cursors.RESIZE_HORIZONTAL: "sb_h_double_arrow",
33: (4)                 cursors.RESIZE_VERTICAL: "sb_v_double_arrow",
34: (0)             }
35: (0)             @contextmanager
36: (0)             def _restore_foreground_window_at_end():
37: (4)                 foreground = _c_internal_utils.Win32_GetForegroundWindow()
38: (4)                 try:
39: (8)                     yield
40: (4)                 finally:
41: (8)                     if foreground and mpl.rcParams['tk.window_focus']:
42: (12)                        _c_internal_utils.Win32_SetForegroundWindow(foreground)
43: (0)             _blit_args = {}
44: (0)             _blit_tcl_name = "mpl_blit_" + uuid.uuid4().hex
45: (0)             def _blit(argsid):
46: (4)                 """
47: (4)                 Thin wrapper to blit called via tkapp.call.
48: (4)                 *argsid* is a unique string identifier to fetch the correct arguments from
49: (4)                 the ``_blit_args`` dict, since arguments cannot be passed directly.
50: (4)                 """
51: (4)                 photoimage, data, offsets, bbox, comp_rule = _blit_args.pop(argsid)
52: (4)                 if not photoimage.tk.call("info", "commands", photoimage):
53: (8)                     return
54: (4)                 _tkagg.blit(photoimage.tk.interpaddr(), str(photoimage), data, comp_rule,
offsets,
55: (16)                            bbox)
56: (0)             def blit(photoimage, aggimage, offsets, bbox=None):
57: (4)                 """
58: (4)                 Blit *aggimage* to *photoimage*.
```

```
59: (4)                 *offsets* is a tuple describing how to fill the ``offset`` field of the
60: (4)                 ``Tk_PhotoImageBlock`` struct: it should be (0, 1, 2, 3) for RGBA8888
data,
61: (4)                 (2, 1, 0, 3) for little-endian ARBG32 (i.e. GBRA8888) data and (1, 2, 3,
0)
62: (4)                 for big-endian ARGB32 (i.e. ARGB8888) data.
63: (4)                 If *bbox* is passed, it defines the region that gets blitted. That region
64: (4)                 will be composed with the previous data according to the alpha channel.
65: (4)                 Blitting will be clipped to pixels inside the canvas, including silently
66: (4)                 doing nothing if the *bbox* region is entirely outside the canvas.
67: (4)                 Tcl events must be dispatched to trigger a blit from a non-Tcl thread.
68: (4)                 """
69: (4)                 data = np.asarray(aggimage)
70: (4)                 height, width = data.shape[:2]
71: (4)                 if bbox is not None:
72: (8)                     (x1, y1), (x2, y2) = bbox.__array__()
73: (8)                     x1 = max(math.floor(x1), 0)
74: (8)                     x2 = min(math.ceil(x2), width)
75: (8)                     y1 = max(math.floor(y1), 0)
76: (8)                     y2 = min(math.ceil(y2), height)
77: (8)                     if (x1 > x2) or (y1 > y2):
78: (12)                        return
79: (8)                     bboxptr = (x1, x2, y1, y2)
80: (8)                     comp_rule = TK_PHOTO_COMPOSITE_OVERLAY
81: (4)                 else:
82: (8)                     bboxptr = (0, width, 0, height)
83: (8)                     comp_rule = TK_PHOTO_COMPOSITE_SET
84: (4)                 args = photoimage, data, offsets, bboxptr, comp_rule
85: (4)                 argsid = str(id(args))
86: (4)                 _blit_args[argsid] = args
87: (4)                 try:
88: (8)                     photoimage.tk.call(_blit_tcl_name, argsid)
89: (4)                 except tk.TclError as e:
90: (8)                     if "invalid command name" not in str(e):
91: (12)                        raise
92: (8)                     photoimage.tk.createcommand(_blit_tcl_name, _blit)
93: (8)                     photoimage.tk.call(_blit_tcl_name, argsid)
94: (0)             class TimerTk(TimerBase):
95: (4)                 """Subclass of `backend_bases.TimerBase` using Tk timer events."""
96: (4)                 def __init__(self, parent, *args, **kwargs):
97: (8)                     self._timer = None
98: (8)                     super().__init__(*args, **kwargs)
99: (8)                     self.parent = parent
100: (4)                def _timer_start(self):
101: (8)                    self._timer_stop()
102: (8)                    self._timer = self.parent.after(self._interval, self._on_timer)
103: (4)                def _timer_stop(self):
104: (8)                    if self._timer is not None:
105: (12)                       self.parent.after_cancel(self._timer)
106: (8)                    self._timer = None
107: (4)                def _on_timer(self):
108: (8)                    super()._on_timer()
109: (8)                    if not self._single and self._timer:
110: (12)                       if self._interval > 0:
111: (16)                           self._timer = self.parent.after(self._interval,
self._on_timer)
112: (12)                       else:
113: (16)                           self._timer = self.parent.after_idle(
114: (20)                               lambda: self.parent.after(self._interval, self._on_timer)
115: (16)                           )
116: (8)                    else:
117: (12)                       self._timer = None
118: (0)            class FigureCanvasTk(FigureCanvasBase):
119: (4)                required_interactive_framework = "tk"
120: (4)                manager_class = _api.classproperty(lambda cls: FigureManagerTk)
121: (4)                def __init__(self, figure=None, master=None):
122: (8)                    super().__init__(figure)
123: (8)                    self._idle_draw_id = None
124: (8)                    self._event_loop_id = None
```

```
125: (8)                    w, h = self.get_width_height(physical=True)
126: (8)                    self._tkcanvas = tk.Canvas(
127: (12)                       master=master, background="white",
128: (12)                       width=w, height=h, borderwidth=0, highlightthickness=0)
129: (8)                    self._tkphoto = tk.PhotoImage(
130: (12)                       master=self._tkcanvas, width=w, height=h)
131: (8)                    self._tkcanvas_image_region = self._tkcanvas.create_image(
132: (12)                       w//2, h//2, image=self._tkphoto)
133: (8)                    self._tkcanvas.bind("<Configure>", self.resize)
134: (8)                    if sys.platform == 'win32':
135: (12)                       self._tkcanvas.bind("<Map>", self._update_device_pixel_ratio)
136: (8)                    self._tkcanvas.bind("<Key>", self.key_press)
137: (8)                    self._tkcanvas.bind("<Motion>", self.motion_notify_event)
138: (8)                    self._tkcanvas.bind("<Enter>", self.enter_notify_event)
139: (8)                    self._tkcanvas.bind("<Leave>", self.leave_notify_event)
140: (8)                    self._tkcanvas.bind("<KeyRelease>", self.key_release)
141: (8)                    for name in ["<Button-1>", "<Button-2>", "<Button-3>"]:
142: (12)                       self._tkcanvas.bind(name, self.button_press_event)
143: (8)                    for name in [
144: (16)                           "<Double-Button-1>", "<Double-Button-2>", "<Double-Button-
3>"]:
145: (12)                       self._tkcanvas.bind(name, self.button_dblclick_event)
146: (8)                    for name in [
147: (16)                           "<ButtonRelease-1>", "<ButtonRelease-2>", "<ButtonRelease-
3>"]:
148: (12)                       self._tkcanvas.bind(name, self.button_release_event)
149: (8)                    for name in "<Button-4>", "<Button-5>":
150: (12)                       self._tkcanvas.bind(name, self.scroll_event)
151: (8)                    root = self._tkcanvas.winfo_toplevel()
152: (8)                    weakself = weakref.ref(self)
153: (8)                    weakroot = weakref.ref(root)
154: (8)                    def scroll_event_windows(event):
155: (12)                       self = weakself()
156: (12)                       if self is None:
157: (16)                           root = weakroot()
158: (16)                           if root is not None:
159: (20)                               root.unbind("<MouseWheel>", scroll_event_windows_id)
160: (16)                           return
161: (12)                       return self.scroll_event_windows(event)
162: (8)                    scroll_event_windows_id = root.bind("<MouseWheel>",
scroll_event_windows, "+")
163: (8)                    def filter_destroy(event):
164: (12)                       self = weakself()
165: (12)                       if self is None:
166: (16)                           root = weakroot()
167: (16)                           if root is not None:
168: (20)                               root.unbind("<Destroy>", filter_destroy_id)
169: (16)                           return
170: (12)                       if event.widget is self._tkcanvas:
171: (16)                           CloseEvent("close_event", self)._process()
172: (8)                    filter_destroy_id = root.bind("<Destroy>", filter_destroy, "+")
173: (8)                    self._tkcanvas.focus_set()
174: (8)                    self._rubberband_rect_black = None
175: (8)                    self._rubberband_rect_white = None
176: (4)                def _update_device_pixel_ratio(self, event=None):
177: (8)                    ratio = round(self._tkcanvas.tk.call('tk', 'scaling') / (96 / 72), 2)
178: (8)                    if self._set_device_pixel_ratio(ratio):
179: (12)                       w, h = self.get_width_height(physical=True)
180: (12)                       self._tkcanvas.configure(width=w, height=h)
181: (4)                def resize(self, event):
182: (8)                    width, height = event.width, event.height
183: (8)                    dpival = self.figure.dpi
184: (8)                    winch = width / dpival
185: (8)                    hinch = height / dpival
186: (8)                    self.figure.set_size_inches(winch, hinch, forward=False)
187: (8)                    self._tkcanvas.delete(self._tkcanvas_image_region)
188: (8)                    self._tkphoto.configure(width=int(width), height=int(height))
189: (8)                    self._tkcanvas_image_region = self._tkcanvas.create_image(
190: (12)                       int(width / 2), int(height / 2), image=self._tkphoto)
```

```
(8)                 ResizeEvent("resize_event", self)._process()
(8)                 self.draw_idle()
(4)             def draw_idle(self):
(8)                 if self._idle_draw_id:
(12)                    return
(8)                 def idle_draw(*args):
(12)                    try:
(16)                        self.draw()
(12)                    finally:
(16)                        self._idle_draw_id = None
(8)                 self._idle_draw_id = self._tkcanvas.after_idle(idle_draw)
(4)             def get_tk_widget(self):
(8)                 """
(8)                 Return the Tk widget used to implement FigureCanvasTkAgg.
(8)                 Although the initial implementation uses a Tk canvas,  this routine
(8)                 is intended to hide that fact.
(8)                 """
(8)                 return self._tkcanvas
(4)             def _event_mpl_coords(self, event):
(8)                 return (self._tkcanvas.canvasx(event.x),
(16)                        self.figure.bbox.height - self._tkcanvas.canvasy(event.y))
(4)             def motion_notify_event(self, event):
(8)                 MouseEvent("motion_notify_event", self,
(19)                           *self._event_mpl_coords(event),
(19)                           modifiers=self._mpl_modifiers(event),
(19)                           guiEvent=event)._process()
(4)             def enter_notify_event(self, event):
(8)                 LocationEvent("figure_enter_event", self,
(22)                              *self._event_mpl_coords(event),
(22)                              modifiers=self._mpl_modifiers(event),
(22)                              guiEvent=event)._process()
(4)             def leave_notify_event(self, event):
(8)                 LocationEvent("figure_leave_event", self,
(22)                              *self._event_mpl_coords(event),
(22)                              modifiers=self._mpl_modifiers(event),
(22)                              guiEvent=event)._process()
(4)             def button_press_event(self, event, dblclick=False):
(8)                 self._tkcanvas.focus_set()
(8)                 num = getattr(event, 'num', None)
(8)                 if sys.platform == 'darwin':  # 2 and 3 are reversed.
(12)                    num = {2: 3, 3: 2}.get(num, num)
(8)                 MouseEvent("button_press_event", self,
(19)                           *self._event_mpl_coords(event), num, dblclick=dblclick,
(19)                           modifiers=self._mpl_modifiers(event),
(19)                           guiEvent=event)._process()
(4)             def button_dblclick_event(self, event):
(8)                 self.button_press_event(event, dblclick=True)
(4)             def button_release_event(self, event):
(8)                 num = getattr(event, 'num', None)
(8)                 if sys.platform == 'darwin':  # 2 and 3 are reversed.
(12)                    num = {2: 3, 3: 2}.get(num, num)
(8)                 MouseEvent("button_release_event", self,
(19)                           *self._event_mpl_coords(event), num,
(19)                           modifiers=self._mpl_modifiers(event),
(19)                           guiEvent=event)._process()
(4)             def scroll_event(self, event):
(8)                 num = getattr(event, 'num', None)
(8)                 step = 1 if num == 4 else -1 if num == 5 else 0
(8)                 MouseEvent("scroll_event", self,
(19)                           *self._event_mpl_coords(event), step=step,
(19)                           modifiers=self._mpl_modifiers(event),
(19)                           guiEvent=event)._process()
(4)             def scroll_event_windows(self, event):
(8)                 """MouseWheel event processor"""
(8)                 w = event.widget.winfo_containing(event.x_root, event.y_root)
(8)                 if w != self._tkcanvas:
(12)                    return
(8)                 x = self._tkcanvas.canvasx(event.x_root - w.winfo_rootx())
(8)                 y = (self.figure.bbox.height
```

```
260: (13)                         - self._tkcanvas.canvasy(event.y_root - w.winfo_rooty()))
261: (8)                     step = event.delta / 120
262: (8)                     MouseEvent("scroll_event", self,
263: (19)                                x, y, step=step, modifiers=self._mpl_modifiers(event),
264: (19)                                guiEvent=event)._process()
265: (4)                 @staticmethod
266: (4)                 def _mpl_modifiers(event, *, exclude=None):
267: (8)                     modifiers = [
268: (12)                        ("ctrl", 1 << 2, "control"),
269: (12)                        ("alt", 1 << 17, "alt"),
270: (12)                        ("shift", 1 << 0, "shift"),
271: (8)                     ] if sys.platform == "win32" else [
272: (12)                        ("ctrl", 1 << 2, "control"),
273: (12)                        ("alt", 1 << 4, "alt"),
274: (12)                        ("shift", 1 << 0, "shift"),
275: (12)                        ("cmd", 1 << 3, "cmd"),
276: (8)                     ] if sys.platform == "darwin" else [
277: (12)                        ("ctrl", 1 << 2, "control"),
278: (12)                        ("alt", 1 << 3, "alt"),
279: (12)                        ("shift", 1 << 0, "shift"),
280: (12)                        ("super", 1 << 6, "super"),
281: (8)                     ]
282: (8)                     return [name for name, mask, key in modifiers
283: (16)                             if event.state & mask and exclude != key]
284: (4)                 def _get_key(self, event):
285: (8)                     unikey = event.char
286: (8)                     key = cbook._unikey_or_keysym_to_mplkey(unikey, event.keysym)
287: (8)                     if key is not None:
288: (12)                        mods = self._mpl_modifiers(event, exclude=key)
289: (12)                        if "shift" in mods and unikey:
290: (16)                            mods.remove("shift")
291: (12)                        return "+".join([*mods, key])
292: (4)                 def key_press(self, event):
293: (8)                     KeyEvent("key_press_event", self,
294: (17)                             self._get_key(event), *self._event_mpl_coords(event),
295: (17)                             guiEvent=event)._process()
296: (4)                 def key_release(self, event):
297: (8)                     KeyEvent("key_release_event", self,
298: (17)                             self._get_key(event), *self._event_mpl_coords(event),
299: (17)                             guiEvent=event)._process()
300: (4)                 def new_timer(self, *args, **kwargs):
301: (8)                     return TimerTk(self._tkcanvas, *args, **kwargs)
302: (4)                 def flush_events(self):
303: (8)                     self._tkcanvas.update()
304: (4)                 def start_event_loop(self, timeout=0):
305: (8)                     if timeout > 0:
306: (12)                        milliseconds = int(1000 * timeout)
307: (12)                        if milliseconds > 0:
308: (16)                            self._event_loop_id = self._tkcanvas.after(
309: (20)                                milliseconds, self.stop_event_loop)
310: (12)                        else:
311: (16)                            self._event_loop_id = self._tkcanvas.after_idle(
312: (20)                                self.stop_event_loop)
313: (8)                     self._tkcanvas.mainloop()
314: (4)                 def stop_event_loop(self):
315: (8)                     if self._event_loop_id:
316: (12)                        self._tkcanvas.after_cancel(self._event_loop_id)
317: (12)                        self._event_loop_id = None
318: (8)                     self._tkcanvas.quit()
319: (4)                 def set_cursor(self, cursor):
320: (8)                     try:
321: (12)                        self._tkcanvas.configure(cursor=cursord[cursor])
322: (8)                     except tkinter.TclError:
323: (12)                        pass
324: (0)             class FigureManagerTk(FigureManagerBase):
325: (4)                 """
326: (4)                 Attributes
327: (4)                 ----------
328: (4)                 canvas : `FigureCanvas`
```

```
(8)                 The FigureCanvas instance
(4)             num : int or str
(8)                 The Figure number
(4)             toolbar : tk.Toolbar
(8)                 The tk.Toolbar
(4)             window : tk.Window
(8)                 The tk.Window
(4)             """
(4)             _owns_mainloop = False
(4)             def __init__(self, canvas, num, window):
(8)                 self.window = window
(8)                 super().__init__(canvas, num)
(8)                 self.window.withdraw()
(8)                 self.canvas._tkcanvas.pack(side=tk.TOP, fill=tk.BOTH, expand=1)
(8)                 window_frame = int(window.wm_frame(), 16)
(8)                 self._window_dpi = tk.IntVar(master=window, value=96,
(37)                                             name=f'window_dpi{window_frame}')
(8)                 self._window_dpi_cbname = ''
(8)                 if _tkagg.enable_dpi_awareness(window_frame, window.tk.interpaddr()):
(12)                    self._window_dpi_cbname = self._window_dpi.trace_add(
(16)                        'write', self._update_window_dpi)
(8)                 self._shown = False
(4)             @classmethod
(4)             def create_with_canvas(cls, canvas_class, figure, num):
(8)                 with _restore_foreground_window_at_end():
(12)                    if cbook._get_running_interactive_framework() is None:
(16)                        cbook._setup_new_guiapp()
(16)                        _c_internal_utils.Win32_SetProcessDpiAwareness_max()
(12)                    window = tk.Tk(className="matplotlib")
(12)                    window.withdraw()
(12)                    icon_fname = str(cbook._get_data_path(
(16)                        'images/matplotlib.png'))
(12)                    icon_img = ImageTk.PhotoImage(file=icon_fname, master=window)
(12)                    icon_fname_large = str(cbook._get_data_path(
(16)                        'images/matplotlib_large.png'))
(12)                    icon_img_large = ImageTk.PhotoImage(
(16)                        file=icon_fname_large, master=window)
(12)                    window.iconphoto(False, icon_img_large, icon_img)
(12)                    canvas = canvas_class(figure, master=window)
(12)                    manager = cls(canvas, num, window)
(12)                    if mpl.is_interactive():
(16)                        manager.show()
(16)                        canvas.draw_idle()
(12)                    return manager
(4)             @classmethod
(4)             def start_main_loop(cls):
(8)                 managers = Gcf.get_all_fig_managers()
(8)                 if managers:
(12)                    first_manager = managers[0]
(12)                    manager_class = type(first_manager)
(12)                    if manager_class._owns_mainloop:
(16)                        return
(12)                    manager_class._owns_mainloop = True
(12)                    try:
(16)                        first_manager.window.mainloop()
(12)                    finally:
(16)                        manager_class._owns_mainloop = False
(4)             def _update_window_dpi(self, *args):
(8)                 newdpi = self._window_dpi.get()
(8)                 self.window.call('tk', 'scaling', newdpi / 72)
(8)                 if self.toolbar and hasattr(self.toolbar, '_rescale'):
(12)                    self.toolbar._rescale()
(8)                 self.canvas._update_device_pixel_ratio()
(4)             def resize(self, width, height):
(8)                 max_size = 1_400_000  # the measured max on xorg 1.20.8 was 1_409_023
(8)                 if (width > max_size or height > max_size) and sys.platform ==
'linux':
(12)                    raise ValueError(
(16)                        'You have requested to resize the '
```

```
397: (16)                            f'Tk window to ({width}, {height}), one of which '
398: (16)                            f'is bigger than {max_size}.  At larger sizes xorg will '
399: (16)                            'either exit with an error on newer versions (~1.20) or '
400: (16)                            'cause corruption on older version (~1.19).  We '
401: (16)                            'do not expect a window over a million pixel wide or tall '
402: (16)                            'to be intended behavior.')
403: (8)                        self.canvas._tkcanvas.configure(width=width, height=height)
404: (4)                    def show(self):
405: (8)                        with _restore_foreground_window_at_end():
406: (12)                           if not self._shown:
407: (16)                               def destroy(*args):
408: (20)                                   Gcf.destroy(self)
409: (16)                               self.window.protocol("WM_DELETE_WINDOW", destroy)
410: (16)                               self.window.deiconify()
411: (16)                               self.canvas._tkcanvas.focus_set()
412: (12)                           else:
413: (16)                               self.canvas.draw_idle()
414: (12)                           if mpl.rcParams['figure.raise_window']:
415: (16)                               self.canvas.manager.window.attributes('-topmost', 1)
416: (16)                               self.canvas.manager.window.attributes('-topmost', 0)
417: (12)                           self._shown = True
418: (4)                    def destroy(self, *args):
419: (8)                        if self.canvas._idle_draw_id:
420: (12)                           self.canvas._tkcanvas.after_cancel(self.canvas._idle_draw_id)
421: (8)                        if self.canvas._event_loop_id:
422: (12)                           self.canvas._tkcanvas.after_cancel(self.canvas._event_loop_id)
423: (8)                        if self._window_dpi_cbname:
424: (12)                           self._window_dpi.trace_remove('write', self._window_dpi_cbname)
425: (8)                        def delayed_destroy():
426: (12)                           self.window.destroy()
427: (12)                           if self._owns_mainloop and not Gcf.get_num_fig_managers():
428: (16)                               self.window.quit()
429: (8)                        if cbook._get_running_interactive_framework() == "tk":
430: (12)                           self.window.after_idle(self.window.after, 0, delayed_destroy)
431: (8)                        else:
432: (12)                           self.window.update()
433: (12)                           delayed_destroy()
434: (4)                    def get_window_title(self):
435: (8)                        return self.window.wm_title()
436: (4)                    def set_window_title(self, title):
437: (8)                        self.window.wm_title(title)
438: (4)                    def full_screen_toggle(self):
439: (8)                        is_fullscreen = bool(self.window.attributes('-fullscreen'))
440: (8)                        self.window.attributes('-fullscreen', not is_fullscreen)
441: (0)                class NavigationToolbar2Tk(NavigationToolbar2, tk.Frame):
442: (4)                    def __init__(self, canvas, window=None, *, pack_toolbar=True):
443: (8)                        """
444: (8)                        Parameters
445: (8)                        ----------
446: (8)                        canvas : `FigureCanvas`
447: (12)                           The figure canvas on which to operate.
448: (8)                        window : tk.Window
449: (12)                           The tk.Window which owns this toolbar.
450: (8)                        pack_toolbar : bool, default: True
451: (12)                           If True, add the toolbar to the parent's pack manager's packing
452: (12)                           list during initialization with ``side="bottom"`` and
``fill="x"``.
453: (12)                           If you want to use the toolbar with a different layout manager,
use
454: (12)                           ``pack_toolbar=False``.
455: (8)                        """
456: (8)                        if window is None:
457: (12)                           window = canvas.get_tk_widget().master
458: (8)                        tk.Frame.__init__(self, master=window, borderwidth=2,
459: (26)                                         width=int(canvas.figure.bbox.width), height=50)
460: (8)                        self._buttons = {}
461: (8)                        for text, tooltip_text, image_file, callback in self.toolitems:
462: (12)                           if text is None:
463: (16)                               self._Spacer()
```

```
464: (12)                        else:
465: (16)                            self._buttons[text] = button = self._Button(
466: (20)                                text,
467: (20)                                str(cbook._get_data_path(f"images/{image_file}.png")),
468: (20)                                toggle=callback in ["zoom", "pan"],
469: (20)                                command=getattr(self, callback),
470: (16)                            )
471: (16)                            if tooltip_text is not None:
472: (20)                                add_tooltip(button, tooltip_text)
473: (8)                    self._label_font = tkinter.font.Font(root=window, size=10)
474: (8)                    label = tk.Label(master=self, font=self._label_font,
475: (25)                                     text='\N{NO-BREAK SPACE}\n\N{NO-BREAK SPACE}')
476: (8)                    label.pack(side=tk.RIGHT)
477: (8)                    self.message = tk.StringVar(master=self)
478: (8)                    self._message_label = tk.Label(master=self, font=self._label_font,
479: (39)                                                  textvariable=self.message,
480: (39)                                                  justify=tk.RIGHT)
481: (8)                    self._message_label.pack(side=tk.RIGHT)
482: (8)                    NavigationToolbar2.__init__(self, canvas)
483: (8)                    if pack_toolbar:
484: (12)                       self.pack(side=tk.BOTTOM, fill=tk.X)
485: (4)                def _rescale(self):
486: (8)                    """
487: (8)                    Scale all children of the toolbar to current DPI setting.
488: (8)                    Before this is called, the Tk scaling setting will have been updated
to
489: (8)                    match the new DPI. Tk widgets do not update for changes to scaling,
but
490: (8)                    all measurements made after the change will match the new scaling.
Thus
491: (8)                    this function re-applies all the same sizes in points, which Tk will
492: (8)                    scale correctly to pixels.
493: (8)                    """
494: (8)                    for widget in self.winfo_children():
495: (12)                       if isinstance(widget, (tk.Button, tk.Checkbutton)):
496: (16)                           if hasattr(widget, '_image_file'):
497: (20)                               NavigationToolbar2Tk._set_image_for_button(self, widget)
498: (16)                           else:
499: (20)                               pass
500: (12)                       elif isinstance(widget, tk.Frame):
501: (16)                           widget.configure(height='18p')
502: (16)                           widget.pack_configure(padx='3p')
503: (12)                       elif isinstance(widget, tk.Label):
504: (16)                           pass  # Text is handled by the font setting instead.
505: (12)                       else:
506: (16)                           _log.warning('Unknown child class %s', widget.winfo_class)
507: (8)                    self._label_font.configure(size=10)
508: (4)                def _update_buttons_checked(self):
509: (8)                    for text, mode in [('Zoom', _Mode.ZOOM), ('Pan', _Mode.PAN)]:
510: (12)                       if text in self._buttons:
511: (16)                           if self.mode == mode:
512: (20)                               self._buttons[text].select()  # NOT .invoke()
513: (16)                           else:
514: (20)                               self._buttons[text].deselect()
515: (4)                def pan(self, *args):
516: (8)                    super().pan(*args)
517: (8)                    self._update_buttons_checked()
518: (4)                def zoom(self, *args):
519: (8)                    super().zoom(*args)
520: (8)                    self._update_buttons_checked()
521: (4)                def set_message(self, s):
522: (8)                    self.message.set(s)
523: (4)                def draw_rubberband(self, event, x0, y0, x1, y1):
524: (8)                    if self.canvas._rubberband_rect_white:
525: (12)                       self.canvas._tkcanvas.delete(self.canvas._rubberband_rect_white)
526: (8)                    if self.canvas._rubberband_rect_black:
527: (12)                       self.canvas._tkcanvas.delete(self.canvas._rubberband_rect_black)
528: (8)                    height = self.canvas.figure.bbox.height
529: (8)                    y0 = height - y0
```

```
530: (8)                    y1 = height - y1
531: (8)                    self.canvas._rubberband_rect_black = (
532: (12)                       self.canvas._tkcanvas.create_rectangle(
533: (16)                           x0, y0, x1, y1))
534: (8)                    self.canvas._rubberband_rect_white = (
535: (12)                       self.canvas._tkcanvas.create_rectangle(
536: (16)                           x0, y0, x1, y1, outline='white', dash=(3, 3)))
537: (4)                def remove_rubberband(self):
538: (8)                    if self.canvas._rubberband_rect_white:
539: (12)                       self.canvas._tkcanvas.delete(self.canvas._rubberband_rect_white)
540: (12)                       self.canvas._rubberband_rect_white = None
541: (8)                    if self.canvas._rubberband_rect_black:
542: (12)                       self.canvas._tkcanvas.delete(self.canvas._rubberband_rect_black)
543: (12)                       self.canvas._rubberband_rect_black = None
544: (4)                def _set_image_for_button(self, button):
545: (8)                    """
546: (8)                    Set the image for a button based on its pixel size.
547: (8)                    The pixel size is determined by the DPI scaling of the window.
548: (8)                    """
549: (8)                    if button._image_file is None:
550: (12)                       return
551: (8)                    path_regular = cbook._get_data_path('images', button._image_file)
552: (8)                    path_large = path_regular.with_name(
553: (12)                       path_regular.name.replace('.png', '_large.png'))
554: (8)                    size = button.winfo_pixels('18p')
555: (8)                    def _get_color(color_name):
556: (12)                       return button.winfo_rgb(button.cget(color_name))
557: (8)                    def _is_dark(color):
558: (12)                       if isinstance(color, str):
559: (16)                           color = _get_color(color)
560: (12)                       return max(color) < 65535 / 2
561: (8)                    def _recolor_icon(image, color):
562: (12)                       image_data = np.asarray(image).copy()
563: (12)                       black_mask = (image_data[..., :3] == 0).all(axis=-1)
564: (12)                       image_data[black_mask, :3] = color
565: (12)                       return Image.fromarray(image_data, mode="RGBA")
566: (8)                    with Image.open(path_large if (size > 24 and path_large.exists())
567: (24)                                   else path_regular) as im:
568: (12)                       im = im.convert("RGBA")
569: (12)                       image = ImageTk.PhotoImage(im.resize((size, size)), master=self)
570: (12)                       button._ntimage = image
571: (12)                       foreground = (255 / 65535) * np.array(
572: (16)                           button.winfo_rgb(button.cget("foreground")))
573: (12)                       im_alt = _recolor_icon(im, foreground)
574: (12)                       image_alt = ImageTk.PhotoImage(
575: (16)                           im_alt.resize((size, size)), master=self)
576: (12)                       button._ntimage_alt = image_alt
577: (8)                    if _is_dark("background"):
578: (12)                       image_kwargs = {"image": image_alt}
579: (8)                    else:
580: (12)                       image_kwargs = {"image": image}
581: (8)                    if (
582: (12)                       isinstance(button, tk.Checkbutton)
583: (12)                       and button.cget("selectcolor") != ""
584: (8)                    ):
585: (12)                       if self._windowingsystem != "x11":
586: (16)                           selectcolor = "selectcolor"
587: (12)                       else:
588: (16)                           r1, g1, b1 = _get_color("selectcolor")
589: (16)                           r2, g2, b2 = _get_color("activebackground")
590: (16)                           selectcolor = ((r1+r2)/2, (g1+g2)/2, (b1+b2)/2)
591: (12)                       if _is_dark(selectcolor):
592: (16)                           image_kwargs["selectimage"] = image_alt
593: (12)                       else:
594: (16)                           image_kwargs["selectimage"] = image
595: (8)                    button.configure(**image_kwargs, height='18p', width='18p')
596: (4)                def _Button(self, text, image_file, toggle, command):
597: (8)                    if not toggle:
598: (12)                       b = tk.Button(
```

```
599: (16)                            master=self, text=text, command=command,
600: (16)                            relief="flat", overrelief="groove", borderwidth=1,
601: (12)                        )
602: (8)                    else:
603: (12)                        var = tk.IntVar(master=self)
604: (12)                        b = tk.Checkbutton(
605: (16)                            master=self, text=text, command=command, indicatoron=False,
606: (16)                            variable=var, offrelief="flat", overrelief="groove",
607: (16)                            borderwidth=1
608: (12)                        )
609: (12)                        b.var = var
610: (8)                    b._image_file = image_file
611: (8)                    if image_file is not None:
612: (12)                        NavigationToolbar2Tk._set_image_for_button(self, b)
613: (8)                    else:
614: (12)                        b.configure(font=self._label_font)
615: (8)                    b.pack(side=tk.LEFT)
616: (8)                    return b
617: (4)                def _Spacer(self):
618: (8)                    s = tk.Frame(master=self, height='18p', relief=tk.RIDGE,
bg='DarkGray')
619: (8)                    s.pack(side=tk.LEFT, padx='3p')
620: (8)                    return s
621: (4)                def save_figure(self, *args):
622: (8)                    filetypes = self.canvas.get_supported_filetypes_grouped()
623: (8)                    tk_filetypes = [
624: (12)                        (name, " ".join(f"*.{ext}" for ext in exts))
625: (12)                        for name, exts in sorted(filetypes.items())
626: (8)                    ]
627: (8)                    default_extension = self.canvas.get_default_filetype()
628: (8)                    default_filetype = self.canvas.get_supported_filetypes()
[default_extension]
629: (8)                    filetype_variable = tk.StringVar(self.canvas.get_tk_widget(),
default_filetype)
630: (8)                    defaultextension = ''
631: (8)                    initialdir = os.path.expanduser(mpl.rcParams['savefig.directory'])
632: (8)                    initialfile = pathlib.Path(self.canvas.get_default_filename()).stem
633: (8)                    fname = tkinter.filedialog.asksaveasfilename(
634: (12)                        master=self.canvas.get_tk_widget().master,
635: (12)                        title='Save the figure',
636: (12)                        filetypes=tk_filetypes,
637: (12)                        defaultextension=defaultextension,
638: (12)                        initialdir=initialdir,
639: (12)                        initialfile=initialfile,
640: (12)                        typevariable=filetype_variable
641: (12)                        )
642: (8)                    if fname in ["", ()]:
643: (12)                        return
644: (8)                    if initialdir != "":
645: (12)                        mpl.rcParams['savefig.directory'] = (
646: (16)                            os.path.dirname(str(fname)))
647: (8)                    if pathlib.Path(fname).suffix[1:] != "":
648: (12)                        extension = None
649: (8)                    else:
650: (12)                        extension = filetypes[filetype_variable.get()][0]
651: (8)                    try:
652: (12)                        self.canvas.figure.savefig(fname, format=extension)
653: (8)                    except Exception as e:
654: (12)                        tkinter.messagebox.showerror("Error saving file", str(e))
655: (4)                def set_history_buttons(self):
656: (8)                    state_map = {True: tk.NORMAL, False: tk.DISABLED}
657: (8)                    can_back = self._nav_stack._pos > 0
658: (8)                    can_forward = self._nav_stack._pos < len(self._nav_stack) - 1
659: (8)                    if "Back" in self._buttons:
660: (12)                        self._buttons['Back']['state'] = state_map[can_back]
661: (8)                    if "Forward" in self._buttons:
662: (12)                        self._buttons['Forward']['state'] = state_map[can_forward]
663: (0)            def add_tooltip(widget, text):
664: (4)                tipwindow = None
```

```
665: (4)                def showtip(event):
666: (8)                    """Display text in tooltip window."""
667: (8)                    nonlocal tipwindow
668: (8)                    if tipwindow or not text:
669: (12)                       return
670: (8)                    x, y, _, _ = widget.bbox("insert")
671: (8)                    x = x + widget.winfo_rootx() + widget.winfo_width()
672: (8)                    y = y + widget.winfo_rooty()
673: (8)                    tipwindow = tk.Toplevel(widget)
674: (8)                    tipwindow.overrideredirect(1)
675: (8)                    tipwindow.geometry(f"+{x}+{y}")
676: (8)                    try:  # For Mac OS
677: (12)                       tipwindow.tk.call("::tk::unsupported::MacWindowStyle",
678: (30)                                         "style", tipwindow._w,
679: (30)                                         "help", "noActivates")
680: (8)                    except tk.TclError:
681: (12)                       pass
682: (8)                    label = tk.Label(tipwindow, text=text, justify=tk.LEFT,
683: (25)                                    relief=tk.SOLID, borderwidth=1)
684: (8)                    label.pack(ipadx=1)
685: (4)                def hidetip(event):
686: (8)                    nonlocal tipwindow
687: (8)                    if tipwindow:
688: (12)                       tipwindow.destroy()
689: (8)                    tipwindow = None
690: (4)                widget.bind("<Enter>", showtip)
691: (4)                widget.bind("<Leave>", hidetip)
692: (0)            @backend_tools._register_tool_class(FigureCanvasTk)
693: (0)            class RubberbandTk(backend_tools.RubberbandBase):
694: (4)                def draw_rubberband(self, x0, y0, x1, y1):
695: (8)                    NavigationToolbar2Tk.draw_rubberband(
696: (12)                       self._make_classic_style_pseudo_toolbar(), None, x0, y0, x1, y1)
697: (4)                def remove_rubberband(self):
698: (8)                    NavigationToolbar2Tk.remove_rubberband(
699: (12)                       self._make_classic_style_pseudo_toolbar())
700: (0)            class ToolbarTk(ToolContainerBase, tk.Frame):
701: (4)                def __init__(self, toolmanager, window=None):
702: (8)                    ToolContainerBase.__init__(self, toolmanager)
703: (8)                    if window is None:
704: (12)                       window = self.toolmanager.canvas.get_tk_widget().master
705: (8)                    xmin, xmax = self.toolmanager.canvas.figure.bbox.intervalx
706: (8)                    height, width = 50, xmax - xmin
707: (8)                    tk.Frame.__init__(self, master=window,
708: (26)                                      width=int(width), height=int(height),
709: (26)                                      borderwidth=2)
710: (8)                    self._label_font = tkinter.font.Font(size=10)
711: (8)                    label = tk.Label(master=self, font=self._label_font,
712: (25)                                    text='\N{NO-BREAK SPACE}\n\N{NO-BREAK SPACE}')
713: (8)                    label.pack(side=tk.RIGHT)
714: (8)                    self._message = tk.StringVar(master=self)
715: (8)                    self._message_label = tk.Label(master=self, font=self._label_font,
716: (39)                                                  textvariable=self._message)
717: (8)                    self._message_label.pack(side=tk.RIGHT)
718: (8)                    self._toolitems = {}
719: (8)                    self.pack(side=tk.TOP, fill=tk.X)
720: (8)                    self._groups = {}
721: (4)                def _rescale(self):
722: (8)                    return NavigationToolbar2Tk._rescale(self)
723: (4)                def add_toolitem(
724: (12)                       self, name, group, position, image_file, description, toggle):
725: (8)                    frame = self._get_groupframe(group)
726: (8)                    buttons = frame.pack_slaves()
727: (8)                    if position >= len(buttons) or position < 0:
728: (12)                       before = None
729: (8)                    else:
730: (12)                       before = buttons[position]
731: (8)                    button = NavigationToolbar2Tk._Button(frame, name, image_file, toggle,
732: (46)                                                         lambda:
self._button_click(name))
```

```
733: (8)                    button.pack_configure(before=before)
734: (8)                    if description is not None:
735: (12)                       add_tooltip(button, description)
736: (8)                    self._toolitems.setdefault(name, [])
737: (8)                    self._toolitems[name].append(button)
738: (4)                def _get_groupframe(self, group):
739: (8)                    if group not in self._groups:
740: (12)                       if self._groups:
741: (16)                           self._add_separator()
742: (12)                       frame = tk.Frame(master=self, borderwidth=0)
743: (12)                       frame.pack(side=tk.LEFT, fill=tk.Y)
744: (12)                       frame._label_font = self._label_font
745: (12)                       self._groups[group] = frame
746: (8)                    return self._groups[group]
747: (4)                def _add_separator(self):
748: (8)                    return NavigationToolbar2Tk._Spacer(self)
749: (4)                def _button_click(self, name):
750: (8)                    self.trigger_tool(name)
751: (4)                def toggle_toolitem(self, name, toggled):
752: (8)                    if name not in self._toolitems:
753: (12)                       return
754: (8)                    for toolitem in self._toolitems[name]:
755: (12)                       if toggled:
756: (16)                           toolitem.select()
757: (12)                       else:
758: (16)                           toolitem.deselect()
759: (4)                def remove_toolitem(self, name):
760: (8)                    for toolitem in self._toolitems.pop(name, []):
761: (12)                       toolitem.pack_forget()
762: (4)                def set_message(self, s):
763: (8)                    self._message.set(s)
764: (0)            @backend_tools._register_tool_class(FigureCanvasTk)
765: (0)            class SaveFigureTk(backend_tools.SaveFigureBase):
766: (4)                def trigger(self, *args):
767: (8)                    NavigationToolbar2Tk.save_figure(
768: (12)                       self._make_classic_style_pseudo_toolbar())
769: (0)            @backend_tools._register_tool_class(FigureCanvasTk)
770: (0)            class ConfigureSubplotsTk(backend_tools.ConfigureSubplotsBase):
771: (4)                def trigger(self, *args):
772: (8)                    NavigationToolbar2Tk.configure_subplots(self)
773: (0)            @backend_tools._register_tool_class(FigureCanvasTk)
774: (0)            class HelpTk(backend_tools.ToolHelpBase):
775: (4)                def trigger(self, *args):
776: (8)                    dialog = SimpleDialog(
777: (12)                       self.figure.canvas._tkcanvas, self._get_help_text(), ["OK"])
778: (8)                    dialog.done = lambda num: dialog.frame.master.withdraw()
779: (0)            Toolbar = ToolbarTk
780: (0)            FigureManagerTk._toolbar2_class = NavigationToolbar2Tk
781: (0)            FigureManagerTk._toolmanager_toolbar_class = ToolbarTk
782: (0)            @_Backend.export
783: (0)            class _BackendTk(_Backend):
784: (4)                backend_version = tk.TkVersion
785: (4)                FigureCanvas = FigureCanvasTk
786: (4)                FigureManager = FigureManagerTk
787: (4)                mainloop = FigureManagerTk.start_main_loop
```

File 104 - backend_tkagg.py:

```
1: (0)              from . import _backend_tk
2: (0)              from .backend_agg import FigureCanvasAgg
3: (0)              from ._backend_tk import _BackendTk, FigureCanvasTk
4: (0)              from ._backend_tk import (  # noqa: F401 # pylint: disable=W0611
5: (4)                  FigureManagerTk, NavigationToolbar2Tk)
6: (0)              class FigureCanvasTkAgg(FigureCanvasAgg, FigureCanvasTk):
7: (4)                  def draw(self):
8: (8)                      super().draw()
9: (8)                      self.blit()
```

```
10: (4)                 def blit(self, bbox=None):
11: (8)                     _backend_tk.blit(self._tkphoto, self.renderer.buffer_rgba(),
12: (25)                                     (0, 1, 2, 3), bbox=bbox)
13: (0)             @_BackendTk.export
14: (0)             class _BackendTkAgg(_BackendTk):
15: (4)                 FigureCanvas = FigureCanvasTkAgg
```

File 105 - backend_webagg.py:

```
1: (0)              """Displays Agg images in the browser, with interactivity."""
2: (0)              from contextlib import contextmanager
3: (0)              import errno
4: (0)              from io import BytesIO
5: (0)              import json
6: (0)              import mimetypes
7: (0)              from pathlib import Path
8: (0)              import random
9: (0)              import sys
10: (0)             import signal
11: (0)             import threading
12: (0)             try:
13: (4)                 import tornado
14: (0)             except ImportError as err:
15: (4)                 raise RuntimeError("The WebAgg backend requires Tornado.") from err
16: (0)             import tornado.web
17: (0)             import tornado.ioloop
18: (0)             import tornado.websocket
19: (0)             import matplotlib as mpl
20: (0)             from matplotlib.backend_bases import _Backend
21: (0)             from matplotlib._pylab_helpers import Gcf
22: (0)             from . import backend_webagg_core as core
23: (0)             from .backend_webagg_core import (  # noqa: F401 # pylint: disable=W0611
24: (4)                 TimerAsyncio, TimerTornado)
25: (0)             webagg_server_thread = threading.Thread(
26: (4)                 target=lambda: tornado.ioloop.IOLoop.instance().start())
27: (0)             class FigureManagerWebAgg(core.FigureManagerWebAgg):
28: (4)                 _toolbar2_class = core.NavigationToolbar2WebAgg
29: (4)                 @classmethod
30: (4)                 def pyplot_show(cls, *, block=None):
31: (8)                     WebAggApplication.initialize()
32: (8)                     url = "http://{address}:{port}{prefix}".format(
33: (12)                        address=WebAggApplication.address,
34: (12)                        port=WebAggApplication.port,
35: (12)                        prefix=WebAggApplication.url_prefix)
36: (8)                     if mpl.rcParams['webagg.open_in_browser']:
37: (12)                        import webbrowser
38: (12)                        if not webbrowser.open(url):
39: (16)                            print(f"To view figure, visit {url}")
40: (8)                     else:
41: (12)                        print(f"To view figure, visit {url}")
42: (8)                     WebAggApplication.start()
43: (0)             class FigureCanvasWebAgg(core.FigureCanvasWebAggCore):
44: (4)                 manager_class = FigureManagerWebAgg
45: (0)             class WebAggApplication(tornado.web.Application):
46: (4)                 initialized = False
47: (4)                 started = False
48: (4)                 class FavIcon(tornado.web.RequestHandler):
49: (8)                     def get(self):
50: (12)                        self.set_header('Content-Type', 'image/png')
51: (12)                        self.write(Path(mpl.get_data_path(),
52: (28)                                        'images/matplotlib.png').read_bytes())
53: (4)                 class SingleFigurePage(tornado.web.RequestHandler):
54: (8)                     def __init__(self, application, request, *, url_prefix='', **kwargs):
55: (12)                        self.url_prefix = url_prefix
56: (12)                        super().__init__(application, request, **kwargs)
57: (8)                     def get(self, fignum):
58: (12)                        fignum = int(fignum)
```

```
(12)                        manager = Gcf.get_fig_manager(fignum)
(12)                        ws_uri = f'ws://{self.request.host}{self.url_prefix}/'
(12)                        self.render(
(16)                            "single_figure.html",
(16)                            prefix=self.url_prefix,
(16)                            ws_uri=ws_uri,
(16)                            fig_id=fignum,
(16)                            toolitems=core.NavigationToolbar2WebAgg.toolitems,
(16)                            canvas=manager.canvas)
(4)                 class AllFiguresPage(tornado.web.RequestHandler):
(8)                     def __init__(self, application, request, *, url_prefix='', **kwargs):
(12)                        self.url_prefix = url_prefix
(12)                        super().__init__(application, request, **kwargs)
(8)                     def get(self):
(12)                        ws_uri = f'ws://{self.request.host}{self.url_prefix}/'
(12)                        self.render(
(16)                            "all_figures.html",
(16)                            prefix=self.url_prefix,
(16)                            ws_uri=ws_uri,
(16)                            figures=sorted(Gcf.figs.items()),
(16)                            toolitems=core.NavigationToolbar2WebAgg.toolitems)
(4)                 class MplJs(tornado.web.RequestHandler):
(8)                     def get(self):
(12)                        self.set_header('Content-Type', 'application/javascript')
(12)                        js_content = core.FigureManagerWebAgg.get_javascript()
(12)                        self.write(js_content)
(4)                 class Download(tornado.web.RequestHandler):
(8)                     def get(self, fignum, fmt):
(12)                        fignum = int(fignum)
(12)                        manager = Gcf.get_fig_manager(fignum)
(12)                        self.set_header(
(16)                            'Content-Type', mimetypes.types_map.get(fmt, 'binary'))
(12)                        buff = BytesIO()
(12)                        manager.canvas.figure.savefig(buff, format=fmt)
(12)                        self.write(buff.getvalue())
(4)                 class WebSocket(tornado.websocket.WebSocketHandler):
(8)                     supports_binary = True
(8)                     def open(self, fignum):
(12)                        self.fignum = int(fignum)
(12)                        self.manager = Gcf.get_fig_manager(self.fignum)
(12)                        self.manager.add_web_socket(self)
(12)                        if hasattr(self, 'set_nodelay'):
(16)                            self.set_nodelay(True)
(8)                     def on_close(self):
(12)                        self.manager.remove_web_socket(self)
(8)                     def on_message(self, message):
(12)                        message = json.loads(message)
(12)                        if message['type'] == 'supports_binary':
(16)                            self.supports_binary = message['value']
(12)                        else:
(16)                            manager = Gcf.get_fig_manager(self.fignum)
(16)                            if manager is not None:
(20)                                manager.handle_json(message)
(8)                     def send_json(self, content):
(12)                        self.write_message(json.dumps(content))
(8)                     def send_binary(self, blob):
(12)                        if self.supports_binary:
(16)                            self.write_message(blob, binary=True)
(12)                        else:
(16)                            data_uri = "{}".format(
(20)                                blob.encode('base64').replace('\n', ''))
(16)                            self.write_message(data_uri)
(4)                 def __init__(self, url_prefix=''):
(8)                     if url_prefix:
(12)                        assert url_prefix[0] == '/' and url_prefix[-1] != '/', \
(16)                            'url_prefix must start with a "/" and not end with one.'
(8)                     super().__init__(
(12)                        [
(16)                            (url_prefix + r'/_static/(.*)',
```

```
128: (17)                            tornado.web.StaticFileHandler,
129: (17)                            {'path': core.FigureManagerWebAgg.get_static_file_path()}),
130: (16)                           (url_prefix + r'/_images/(.*)',
131: (17)                            tornado.web.StaticFileHandler,
132: (17)                            {'path': Path(mpl.get_data_path(), 'images')}),
133: (16)                           (url_prefix + r'/favicon.ico', self.FavIcon),
134: (16)                           (url_prefix + r'/([0-9]+)', self.SingleFigurePage,
135: (17)                            {'url_prefix': url_prefix}),
136: (16)                           (url_prefix + r'/?', self.AllFiguresPage,
137: (17)                            {'url_prefix': url_prefix}),
138: (16)                           (url_prefix + r'/js/mpl.js', self.MplJs),
139: (16)                           (url_prefix + r'/([0-9]+)/ws', self.WebSocket),
140: (16)                           (url_prefix + r'/([0-9]+)/download.([a-z0-9.]+)',
141: (17)                            self.Download),
142: (12)                       ],
143: (12)                       template_path=core.FigureManagerWebAgg.get_static_file_path())
144: (4)                @classmethod
145: (4)                def initialize(cls, url_prefix='', port=None, address=None):
146: (8)                    if cls.initialized:
147: (12)                       return
148: (8)                    app = cls(url_prefix=url_prefix)
149: (8)                    cls.url_prefix = url_prefix
150: (8)                    def random_ports(port, n):
151: (12)                       """
152: (12)                       Generate a list of n random ports near the given port.
153: (12)                       The first 5 ports will be sequential, and the remaining n-5 will
be
154: (12)                       randomly selected in the range [port-2*n, port+2*n].
155: (12)                       """
156: (12)                       for i in range(min(5, n)):
157: (16)                           yield port + i
158: (12)                       for i in range(n - 5):
159: (16)                           yield port + random.randint(-2 * n, 2 * n)
160: (8)                    if address is None:
161: (12)                       cls.address = mpl.rcParams['webagg.address']
162: (8)                    else:
163: (12)                       cls.address = address
164: (8)                    cls.port = mpl.rcParams['webagg.port']
165: (8)                    for port in random_ports(cls.port,
166: (33)                                            mpl.rcParams['webagg.port_retries']):
167: (12)                       try:
168: (16)                           app.listen(port, cls.address)
169: (12)                       except OSError as e:
170: (16)                           if e.errno != errno.EADDRINUSE:
171: (20)                               raise
172: (12)                       else:
173: (16)                           cls.port = port
174: (16)                           break
175: (8)                    else:
176: (12)                       raise SystemExit(
177: (16)                           "The webagg server could not be started because an available "
178: (16)                           "port could not be found")
179: (8)                    cls.initialized = True
180: (4)                @classmethod
181: (4)                def start(cls):
182: (8)                    import asyncio
183: (8)                    try:
184: (12)                       asyncio.get_running_loop()
185: (8)                    except RuntimeError:
186: (12)                       pass
187: (8)                    else:
188: (12)                       cls.started = True
189: (8)                    if cls.started:
190: (12)                       return
191: (8)                    """
192: (8)                    IOLoop.running() was removed as of Tornado 2.4; see for example
193: (8)                    https://groups.google.com/forum/#!topic/python-tornado/QLMzkpQBGOY
194: (8)                    Thus there is no correct way to check if the loop has already been
195: (8)                    launched. We may end up with two concurrently running loops in that
```

```
196: (8)                    unlucky case with all the expected consequences.
197: (8)                    """
198: (8)                    ioloop = tornado.ioloop.IOLoop.instance()
199: (8)                    def shutdown():
200: (12)                       ioloop.stop()
201: (12)                       print("Server is stopped")
202: (12)                       sys.stdout.flush()
203: (12)                       cls.started = False
204: (8)                    @contextmanager
205: (8)                    def catch_sigint():
206: (12)                       old_handler = signal.signal(
207: (16)                           signal.SIGINT,
208: (16)                           lambda sig, frame: ioloop.add_callback_from_signal(shutdown))
209: (12)                       try:
210: (16)                           yield
211: (12)                       finally:
212: (16)                           signal.signal(signal.SIGINT, old_handler)
213: (8)                    cls.started = True
214: (8)                    print("Press Ctrl+C to stop WebAgg server")
215: (8)                    sys.stdout.flush()
216: (8)                    with catch_sigint():
217: (12)                       ioloop.start()
218: (0)            def ipython_inline_display(figure):
219: (4)                import tornado.template
220: (4)                WebAggApplication.initialize()
221: (4)                import asyncio
222: (4)                try:
223: (8)                    asyncio.get_running_loop()
224: (4)                except RuntimeError:
225: (8)                    if not webagg_server_thread.is_alive():
226: (12)                       webagg_server_thread.start()
227: (4)                fignum = figure.number
228: (4)                tpl = Path(core.FigureManagerWebAgg.get_static_file_path(),
229: (15)                          "ipython_inline_figure.html").read_text()
230: (4)                t = tornado.template.Template(tpl)
231: (4)                return t.generate(
232: (8)                    prefix=WebAggApplication.url_prefix,
233: (8)                    fig_id=fignum,
234: (8)                    toolitems=core.NavigationToolbar2WebAgg.toolitems,
235: (8)                    canvas=figure.canvas,
236: (8)                    port=WebAggApplication.port).decode('utf-8')
237: (0)            @_Backend.export
238: (0)            class _BackendWebAgg(_Backend):
239: (4)                FigureCanvas = FigureCanvasWebAgg
240: (4)                FigureManager = FigureManagerWebAgg
```

File 106 - backend_tkcairo.py:

```
1: (0)              import sys
2: (0)              import numpy as np
3: (0)              from . import _backend_tk
4: (0)              from .backend_cairo import cairo, FigureCanvasCairo
5: (0)              from ._backend_tk import _BackendTk, FigureCanvasTk
6: (0)              class FigureCanvasTkCairo(FigureCanvasCairo, FigureCanvasTk):
7: (4)                  def draw(self):
8: (8)                      width = int(self.figure.bbox.width)
9: (8)                      height = int(self.figure.bbox.height)
10: (8)                     surface = cairo.ImageSurface(cairo.FORMAT_ARGB32, width, height)
11: (8)                     self._renderer.set_context(cairo.Context(surface))
12: (8)                     self._renderer.dpi = self.figure.dpi
13: (8)                     self.figure.draw(self._renderer)
14: (8)                     buf = np.reshape(surface.get_data(), (height, width, 4))
15: (8)                     _backend_tk.blit(
16: (12)                        self._tkphoto, buf,
17: (12)                        (2, 1, 0, 3) if sys.byteorder == "little" else (1, 2, 3, 0))
18: (0)             @_BackendTk.export
19: (0)             class _BackendTkCairo(_BackendTk):
```

```
20: (4)                 FigureCanvas = FigureCanvasTkCairo

----------------------------------------

File 107 - backend_webagg_core.py:

1: (0)              """Displays Agg images in the browser, with interactivity."""
2: (0)              import asyncio
3: (0)              import datetime
4: (0)              from io import BytesIO, StringIO
5: (0)              import json
6: (0)              import logging
7: (0)              import os
8: (0)              from pathlib import Path
9: (0)              import numpy as np
10: (0)             from PIL import Image
11: (0)             from matplotlib import _api, backend_bases, backend_tools
12: (0)             from matplotlib.backends import backend_agg
13: (0)             from matplotlib.backend_bases import (
14: (4)                 _Backend, KeyEvent, LocationEvent, MouseEvent, ResizeEvent)
15: (0)             _log = logging.getLogger(__name__)
16: (0)             _SPECIAL_KEYS_LUT = {'Alt': 'alt',
17: (21)                                 'AltGraph': 'alt',
18: (21)                                 'CapsLock': 'caps_lock',
19: (21)                                 'Control': 'control',
20: (21)                                 'Meta': 'meta',
21: (21)                                 'NumLock': 'num_lock',
22: (21)                                 'ScrollLock': 'scroll_lock',
23: (21)                                 'Shift': 'shift',
24: (21)                                 'Super': 'super',
25: (21)                                 'Enter': 'enter',
26: (21)                                 'Tab': 'tab',
27: (21)                                 'ArrowDown': 'down',
28: (21)                                 'ArrowLeft': 'left',
29: (21)                                 'ArrowRight': 'right',
30: (21)                                 'ArrowUp': 'up',
31: (21)                                 'End': 'end',
32: (21)                                 'Home': 'home',
33: (21)                                 'PageDown': 'pagedown',
34: (21)                                 'PageUp': 'pageup',
35: (21)                                 'Backspace': 'backspace',
36: (21)                                 'Delete': 'delete',
37: (21)                                 'Insert': 'insert',
38: (21)                                 'Escape': 'escape',
39: (21)                                 'Pause': 'pause',
40: (21)                                 'Select': 'select',
41: (21)                                 'Dead': 'dead',
42: (21)                                 'F1': 'f1',
43: (21)                                 'F2': 'f2',
44: (21)                                 'F3': 'f3',
45: (21)                                 'F4': 'f4',
46: (21)                                 'F5': 'f5',
47: (21)                                 'F6': 'f6',
48: (21)                                 'F7': 'f7',
49: (21)                                 'F8': 'f8',
50: (21)                                 'F9': 'f9',
51: (21)                                 'F10': 'f10',
52: (21)                                 'F11': 'f11',
53: (21)                                 'F12': 'f12'}
54: (0)             def _handle_key(key):
55: (4)                 """Handle key values"""
56: (4)                 value = key[key.index('k') + 1:]
57: (4)                 if 'shift+' in key:
58: (8)                     if len(value) == 1:
59: (12)                        key = key.replace('shift+', '')
60: (4)                 if value in _SPECIAL_KEYS_LUT:
61: (8)                     value = _SPECIAL_KEYS_LUT[value]
62: (4)                 key = key[:key.index('k')] + value
63: (4)                 return key
```

```
(0)  class TimerTornado(backend_bases.TimerBase):
(4)      def __init__(self, *args, **kwargs):
(8)          self._timer = None
(8)          super().__init__(*args, **kwargs)
(4)      def _timer_start(self):
(8)          import tornado
(8)          self._timer_stop()
(8)          if self._single:
(12)             ioloop = tornado.ioloop.IOLoop.instance()
(12)             self._timer = ioloop.add_timeout(
(16)                 datetime.timedelta(milliseconds=self.interval),
(16)                 self._on_timer)
(8)          else:
(12)             self._timer = tornado.ioloop.PeriodicCallback(
(16)                 self._on_timer,
(16)                 max(self.interval, 1e-6))
(12)             self._timer.start()
(4)      def _timer_stop(self):
(8)          import tornado
(8)          if self._timer is None:
(12)             return
(8)          elif self._single:
(12)             ioloop = tornado.ioloop.IOLoop.instance()
(12)             ioloop.remove_timeout(self._timer)
(8)          else:
(12)             self._timer.stop()
(8)          self._timer = None
(4)      def _timer_set_interval(self):
(8)          if self._timer is not None:
(12)             self._timer_stop()
(12)             self._timer_start()
(0)  class TimerAsyncio(backend_bases.TimerBase):
(4)      def __init__(self, *args, **kwargs):
(8)          self._task = None
(8)          super().__init__(*args, **kwargs)
(4)      async def _timer_task(self, interval):
(8)          while True:
(12)             try:
(16)                 await asyncio.sleep(interval)
(16)                 self._on_timer()
(16)                 if self._single:
(20)                     break
(12)             except asyncio.CancelledError:
(16)                 break
(4)      def _timer_start(self):
(8)          self._timer_stop()
(8)          self._task = asyncio.ensure_future(
(12)             self._timer_task(max(self.interval / 1_000., 1e-6))
(8)          )
(4)      def _timer_stop(self):
(8)          if self._task is not None:
(12)             self._task.cancel()
(8)          self._task = None
(4)      def _timer_set_interval(self):
(8)          if self._task is not None:
(12)             self._timer_stop()
(12)             self._timer_start()
(0)  class FigureCanvasWebAggCore(backend_agg.FigureCanvasAgg):
(4)      manager_class = _api.classproperty(lambda cls: FigureManagerWebAgg)
(4)      _timer_cls = TimerAsyncio
(4)      supports_blit = False
(4)      def __init__(self, *args, **kwargs):
(8)          super().__init__(*args, **kwargs)
(8)          self._png_is_old = True
(8)          self._force_full = True
(8)          self._last_buff = np.empty((0, 0))
(8)          self._current_image_mode = 'full'
(8)          self._last_mouse_xy = (None, None)
(4)      def show(self):
```

```
(8)                 from matplotlib.pyplot import show
(8)                 show()
(4)             def draw(self):
(8)                 self._png_is_old = True
(8)                 try:
(12)                    super().draw()
(8)                 finally:
(12)                    self.manager.refresh_all()  # Swap the frames.
(4)             def blit(self, bbox=None):
(8)                 self._png_is_old = True
(8)                 self.manager.refresh_all()
(4)             def draw_idle(self):
(8)                 self.send_event("draw")
(4)             def set_cursor(self, cursor):
(8)                 cursor = _api.check_getitem({
(12)                    backend_tools.Cursors.HAND: 'pointer',
(12)                    backend_tools.Cursors.POINTER: 'default',
(12)                    backend_tools.Cursors.SELECT_REGION: 'crosshair',
(12)                    backend_tools.Cursors.MOVE: 'move',
(12)                    backend_tools.Cursors.WAIT: 'wait',
(12)                    backend_tools.Cursors.RESIZE_HORIZONTAL: 'ew-resize',
(12)                    backend_tools.Cursors.RESIZE_VERTICAL: 'ns-resize',
(8)                 }, cursor=cursor)
(8)                 self.send_event('cursor', cursor=cursor)
(4)             def set_image_mode(self, mode):
(8)                 """
(8)                 Set the image mode for any subsequent images which will be sent
(8)                 to the clients. The modes may currently be either 'full' or 'diff'.
(8)                 Note: diff images may not contain transparency, therefore upon
(8)                 draw this mode may be changed if the resulting image has any
(8)                 transparent component.
(8)                 """
(8)                 _api.check_in_list(['full', 'diff'], mode=mode)
(8)                 if self._current_image_mode != mode:
(12)                    self._current_image_mode = mode
(12)                    self.handle_send_image_mode(None)
(4)             def get_diff_image(self):
(8)                 if self._png_is_old:
(12)                    renderer = self.get_renderer()
(12)                    pixels = np.asarray(renderer.buffer_rgba())
(12)                    buff = pixels.view(np.uint32).squeeze(2)
(12)                    if (self._force_full
(20)                            or buff.shape != self._last_buff.shape
(20)                            or (pixels[:, :, 3] != 255).any()):
(16)                        self.set_image_mode('full')
(16)                        output = buff
(12)                    else:
(16)                        self.set_image_mode('diff')
(16)                        diff = buff != self._last_buff
(16)                        output = np.where(diff, buff, 0)
(12)                    self._last_buff = buff.copy()
(12)                    self._force_full = False
(12)                    self._png_is_old = False
(12)                    data = output.view(dtype=np.uint8).reshape((*output.shape, 4))
(12)                    with BytesIO() as png:
(16)                        Image.fromarray(data).save(png, format="png")
(16)                        return png.getvalue()
(4)             def handle_event(self, event):
(8)                 e_type = event['type']
(8)                 handler = getattr(self, f'handle_{e_type}',
(26)                                  self.handle_unknown_event)
(8)                 return handler(event)
(4)             def handle_unknown_event(self, event):
(8)                 _log.warning('Unhandled message type %s. %s', event["type"], event)
(4)             def handle_ack(self, event):
(8)                 pass
(4)             def handle_draw(self, event):
(8)                 self.draw()
(4)             def _handle_mouse(self, event):
```

```
202: (8)                    x = event['x']
203: (8)                    y = event['y']
204: (8)                    y = self.get_renderer().height - y
205: (8)                    self._last_mouse_xy = x, y
206: (8)                    button = event['button'] + 1
207: (8)                    e_type = event['type']
208: (8)                    modifiers = event['modifiers']
209: (8)                    guiEvent = event.get('guiEvent')
210: (8)                    if e_type in ['button_press', 'button_release']:
211: (12)                       MouseEvent(e_type + '_event', self, x, y, button,
212: (23)                                   modifiers=modifiers, guiEvent=guiEvent)._process()
213: (8)                    elif e_type == 'dblclick':
214: (12)                       MouseEvent('button_press_event', self, x, y, button,
dblclick=True,
215: (23)                                   modifiers=modifiers, guiEvent=guiEvent)._process()
216: (8)                    elif e_type == 'scroll':
217: (12)                       MouseEvent('scroll_event', self, x, y, step=event['step'],
218: (23)                                   modifiers=modifiers, guiEvent=guiEvent)._process()
219: (8)                    elif e_type == 'motion_notify':
220: (12)                       MouseEvent(e_type + '_event', self, x, y,
221: (23)                                   modifiers=modifiers, guiEvent=guiEvent)._process()
222: (8)                    elif e_type in ['figure_enter', 'figure_leave']:
223: (12)                       LocationEvent(e_type + '_event', self, x, y,
224: (26)                                     modifiers=modifiers, guiEvent=guiEvent)._process()
225: (4)                handle_button_press = handle_button_release = handle_dblclick = \
226: (8)                    handle_figure_enter = handle_figure_leave = handle_motion_notify = \
227: (8)                    handle_scroll = _handle_mouse
228: (4)                def _handle_key(self, event):
229: (8)                    KeyEvent(event['type'] + '_event', self,
230: (17)                            _handle_key(event['key']), *self._last_mouse_xy,
231: (17)                            guiEvent=event.get('guiEvent'))._process()
232: (4)                handle_key_press = handle_key_release = _handle_key
233: (4)                def handle_toolbar_button(self, event):
234: (8)                    getattr(self.toolbar, event['name'])()
235: (4)                def handle_refresh(self, event):
236: (8)                    figure_label = self.figure.get_label()
237: (8)                    if not figure_label:
238: (12)                       figure_label = f"Figure {self.manager.num}"
239: (8)                    self.send_event('figure_label', label=figure_label)
240: (8)                    self._force_full = True
241: (8)                    if self.toolbar:
242: (12)                       self.toolbar.set_history_buttons()
243: (8)                    self.draw_idle()
244: (4)                def handle_resize(self, event):
245: (8)                    x = int(event.get('width', 800)) * self.device_pixel_ratio
246: (8)                    y = int(event.get('height', 800)) * self.device_pixel_ratio
247: (8)                    fig = self.figure
248: (8)                    fig.set_size_inches(x / fig.dpi, y / fig.dpi, forward=False)
249: (8)                    self._png_is_old = True
250: (8)                    self.manager.resize(*fig.bbox.size, forward=False)
251: (8)                    ResizeEvent('resize_event', self)._process()
252: (8)                    self.draw_idle()
253: (4)                def handle_send_image_mode(self, event):
254: (8)                    self.send_event('image_mode', mode=self._current_image_mode)
255: (4)                def handle_set_device_pixel_ratio(self, event):
256: (8)                    self._handle_set_device_pixel_ratio(event.get('device_pixel_ratio',
1))
257: (4)                def handle_set_dpi_ratio(self, event):
258: (8)                    self._handle_set_device_pixel_ratio(event.get('dpi_ratio', 1))
259: (4)                def _handle_set_device_pixel_ratio(self, device_pixel_ratio):
260: (8)                    if self._set_device_pixel_ratio(device_pixel_ratio):
261: (12)                       self._force_full = True
262: (12)                       self.draw_idle()
263: (4)                def send_event(self, event_type, **kwargs):
264: (8)                    if self.manager:
265: (12)                       self.manager._send_event(event_type, **kwargs)
266: (0)            _ALLOWED_TOOL_ITEMS = {
267: (4)                'home',
268: (4)                'back',
```

```
(4)                 'forward',
(4)                 'pan',
(4)                 'zoom',
(4)                 'download',
(4)                 None,
(0)             }
(0)             class NavigationToolbar2WebAgg(backend_bases.NavigationToolbar2):
(4)                 toolitems = [
(8)                     (text, tooltip_text, image_file, name_of_method)
(8)                     for text, tooltip_text, image_file, name_of_method
(8)                     in (*backend_bases.NavigationToolbar2.toolitems,
(12)                        ('Download', 'Download plot', 'filesave', 'download'))
(8)                     if name_of_method in _ALLOWED_TOOL_ITEMS
(4)                 ]
(4)                 def __init__(self, canvas):
(8)                     self.message = ''
(8)                     super().__init__(canvas)
(4)                 def set_message(self, message):
(8)                     if message != self.message:
(12)                        self.canvas.send_event("message", message=message)
(8)                     self.message = message
(4)                 def draw_rubberband(self, event, x0, y0, x1, y1):
(8)                     self.canvas.send_event("rubberband", x0=x0, y0=y0, x1=x1, y1=y1)
(4)                 def remove_rubberband(self):
(8)                     self.canvas.send_event("rubberband", x0=-1, y0=-1, x1=-1, y1=-1)
(4)                 def save_figure(self, *args):
(8)                     """Save the current figure"""
(8)                     self.canvas.send_event('save')
(4)                 def pan(self):
(8)                     super().pan()
(8)                     self.canvas.send_event('navigate_mode', mode=self.mode.name)
(4)                 def zoom(self):
(8)                     super().zoom()
(8)                     self.canvas.send_event('navigate_mode', mode=self.mode.name)
(4)                 def set_history_buttons(self):
(8)                     can_backward = self._nav_stack._pos > 0
(8)                     can_forward = self._nav_stack._pos < len(self._nav_stack) - 1
(8)                     self.canvas.send_event('history_buttons',
(31)                                           Back=can_backward, Forward=can_forward)
(0)             class FigureManagerWebAgg(backend_bases.FigureManagerBase):
(4)                 _toolbar2_class = None
(4)                 ToolbarCls = NavigationToolbar2WebAgg
(4)                 _window_title = "Matplotlib"
(4)                 def __init__(self, canvas, num):
(8)                     self.web_sockets = set()
(8)                     super().__init__(canvas, num)
(4)                 def show(self):
(8)                     pass
(4)                 def resize(self, w, h, forward=True):
(8)                     self._send_event(
(12)                        'resize',
(12)                        size=(w / self.canvas.device_pixel_ratio,
(18)                              h / self.canvas.device_pixel_ratio),
(12)                        forward=forward)
(4)                 def set_window_title(self, title):
(8)                     self._send_event('figure_label', label=title)
(8)                     self._window_title = title
(4)                 def get_window_title(self):
(8)                     return self._window_title
(4)                 def add_web_socket(self, web_socket):
(8)                     assert hasattr(web_socket, 'send_binary')
(8)                     assert hasattr(web_socket, 'send_json')
(8)                     self.web_sockets.add(web_socket)
(8)                     self.resize(*self.canvas.figure.bbox.size)
(8)                     self._send_event('refresh')
(4)                 def remove_web_socket(self, web_socket):
(8)                     self.web_sockets.remove(web_socket)
(4)                 def handle_json(self, content):
(8)                     self.canvas.handle_event(content)
```

```
338: (4)                def refresh_all(self):
339: (8)                    if self.web_sockets:
340: (12)                       diff = self.canvas.get_diff_image()
341: (12)                       if diff is not None:
342: (16)                           for s in self.web_sockets:
343: (20)                               s.send_binary(diff)
344: (4)                @classmethod
345: (4)                def get_javascript(cls, stream=None):
346: (8)                    if stream is None:
347: (12)                       output = StringIO()
348: (8)                    else:
349: (12)                       output = stream
350: (8)                    output.write((Path(__file__).parent / "web_backend/js/mpl.js")
351: (21)                                 .read_text(encoding="utf-8"))
352: (8)                    toolitems = []
353: (8)                    for name, tooltip, image, method in cls.ToolbarCls.toolitems:
354: (12)                       if name is None:
355: (16)                           toolitems.append(['', '', '', ''])
356: (12)                       else:
357: (16)                           toolitems.append([name, tooltip, image, method])
358: (8)                    output.write(f"mpl.toolbar_items = {json.dumps(toolitems)};\n\n")
359: (8)                    extensions = []
360: (8)                    for filetype, ext in sorted(FigureCanvasWebAggCore.
361: (36)                                               get_supported_filetypes_grouped().
362: (36)                                               items()):
363: (12)                       extensions.append(ext[0])
364: (8)                    output.write(f"mpl.extensions = {json.dumps(extensions)};\n\n")
365: (8)                    output.write("mpl.default_extension = {};".format(
366: (12)                       json.dumps(FigureCanvasWebAggCore.get_default_filetype())))
367: (8)                    if stream is None:
368: (12)                       return output.getvalue()
369: (4)                @classmethod
370: (4)                def get_static_file_path(cls):
371: (8)                    return os.path.join(os.path.dirname(__file__), 'web_backend')
372: (4)                def _send_event(self, event_type, **kwargs):
373: (8)                    payload = {'type': event_type, **kwargs}
374: (8)                    for s in self.web_sockets:
375: (12)                       s.send_json(payload)
376: (0)            @_Backend.export
377: (0)            class _BackendWebAggCoreAgg(_Backend):
378: (4)                FigureCanvas = FigureCanvasWebAggCore
379: (4)                FigureManager = FigureManagerWebAgg
```

File 108 - backend_wx.py:

```
1: (0)              """
2: (0)              A wxPython backend for matplotlib.
3: (0)              Originally contributed by Jeremy O'Donoghue (jeremy@o-donoghue.com) and John
4: (0)              Hunter (jdhunter@ace.bsd.uchicago.edu).
5: (0)              Copyright (C) Jeremy O'Donoghue & John Hunter, 2003-4.
6: (0)              """
7: (0)              import functools
8: (0)              import logging
9: (0)              import math
10: (0)             import pathlib
11: (0)             import sys
12: (0)             import weakref
13: (0)             import numpy as np
14: (0)             import PIL.Image
15: (0)             import matplotlib as mpl
16: (0)             from matplotlib.backend_bases import (
17: (4)                 _Backend, FigureCanvasBase, FigureManagerBase,
18: (4)                 GraphicsContextBase, MouseButton, NavigationToolbar2, RendererBase,
19: (4)                 TimerBase, ToolContainerBase, cursors,
20: (4)                 CloseEvent, KeyEvent, LocationEvent, MouseEvent, ResizeEvent)
21: (0)             from matplotlib import _api, cbook, backend_tools, _c_internal_utils
22: (0)             from matplotlib._pylab_helpers import Gcf
```

```
(0)             from matplotlib.path import Path
(0)             from matplotlib.transforms import Affine2D
(0)             import wx
(0)             import wx.svg
(0)             _log = logging.getLogger(__name__)
(0)             PIXELS_PER_INCH = 75
(0)             @functools.lru_cache(1)
(0)             def _create_wxapp():
(4)                 wxapp = wx.App(False)
(4)                 wxapp.SetExitOnFrameDelete(True)
(4)                 cbook._setup_new_guiapp()
(4)                 _c_internal_utils.Win32_SetProcessDpiAwareness_max()
(4)                 return wxapp
(0)             class TimerWx(TimerBase):
(4)                 """Subclass of `.TimerBase` using wx.Timer events."""
(4)                 def __init__(self, *args, **kwargs):
(8)                     self._timer = wx.Timer()
(8)                     self._timer.Notify = self._on_timer
(8)                     super().__init__(*args, **kwargs)
(4)                 def _timer_start(self):
(8)                     self._timer.Start(self._interval, self._single)
(4)                 def _timer_stop(self):
(8)                     self._timer.Stop()
(4)                 def _timer_set_interval(self):
(8)                     if self._timer.IsRunning():
(12)                        self._timer_start()  # Restart with new interval.
(0)             @_api.deprecated(
(4)                 "2.0", name="wx", obj_type="backend", removal="the future",
(4)                 alternative="wxagg",
(4)                 addendum="See the Matplotlib usage FAQ for more info on backends.")
(0)             class RendererWx(RendererBase):
(4)                 """
(4)                 The renderer handles all the drawing primitives using a graphics
(4)                 context instance that controls the colors/styles. It acts as the
(4)                 'renderer' instance used by many classes in the hierarchy.
(4)                 """
(4)                 fontweights = {
(8)                     100: wx.FONTWEIGHT_LIGHT,
(8)                     200: wx.FONTWEIGHT_LIGHT,
(8)                     300: wx.FONTWEIGHT_LIGHT,
(8)                     400: wx.FONTWEIGHT_NORMAL,
(8)                     500: wx.FONTWEIGHT_NORMAL,
(8)                     600: wx.FONTWEIGHT_NORMAL,
(8)                     700: wx.FONTWEIGHT_BOLD,
(8)                     800: wx.FONTWEIGHT_BOLD,
(8)                     900: wx.FONTWEIGHT_BOLD,
(8)                     'ultralight': wx.FONTWEIGHT_LIGHT,
(8)                     'light': wx.FONTWEIGHT_LIGHT,
(8)                     'normal': wx.FONTWEIGHT_NORMAL,
(8)                     'medium': wx.FONTWEIGHT_NORMAL,
(8)                     'semibold': wx.FONTWEIGHT_NORMAL,
(8)                     'bold': wx.FONTWEIGHT_BOLD,
(8)                     'heavy': wx.FONTWEIGHT_BOLD,
(8)                     'ultrabold': wx.FONTWEIGHT_BOLD,
(8)                     'black': wx.FONTWEIGHT_BOLD,
(4)                 }
(4)                 fontangles = {
(8)                     'italic': wx.FONTSTYLE_ITALIC,
(8)                     'normal': wx.FONTSTYLE_NORMAL,
(8)                     'oblique': wx.FONTSTYLE_SLANT,
(4)                 }
(4)                 fontnames = {
(8)                     'Sans': wx.FONTFAMILY_SWISS,
(8)                     'Roman': wx.FONTFAMILY_ROMAN,
(8)                     'Script': wx.FONTFAMILY_SCRIPT,
(8)                     'Decorative': wx.FONTFAMILY_DECORATIVE,
(8)                     'Modern': wx.FONTFAMILY_MODERN,
(8)                     'Courier': wx.FONTFAMILY_MODERN,
(8)                     'courier': wx.FONTFAMILY_MODERN,
```

```
(4)     }
(4)     def __init__(self, bitmap, dpi):
(8)         """Initialise a wxWindows renderer instance."""
(8)         super().__init__()
(8)         _log.debug("%s - __init__()", type(self))
(8)         self.width = bitmap.GetWidth()
(8)         self.height = bitmap.GetHeight()
(8)         self.bitmap = bitmap
(8)         self.fontd = {}
(8)         self.dpi = dpi
(8)         self.gc = None
(4)     def flipy(self):
(8)         return True
(4)     def get_text_width_height_descent(self, s, prop, ismath):
(8)         if ismath:
(12)            s = cbook.strip_math(s)
(8)         if self.gc is None:
(12)            gc = self.new_gc()
(8)         else:
(12)            gc = self.gc
(8)         gfx_ctx = gc.gfx_ctx
(8)         font = self.get_wx_font(s, prop)
(8)         gfx_ctx.SetFont(font, wx.BLACK)
(8)         w, h, descent, leading = gfx_ctx.GetFullTextExtent(s)
(8)         return w, h, descent
(4)     def get_canvas_width_height(self):
(8)         return self.width, self.height
(4)     def handle_clip_rectangle(self, gc):
(8)         new_bounds = gc.get_clip_rectangle()
(8)         if new_bounds is not None:
(12)            new_bounds = new_bounds.bounds
(8)         gfx_ctx = gc.gfx_ctx
(8)         if gfx_ctx._lastcliprect != new_bounds:
(12)            gfx_ctx._lastcliprect = new_bounds
(12)            if new_bounds is None:
(16)                gfx_ctx.ResetClip()
(12)            else:
(16)                gfx_ctx.Clip(new_bounds[0],
(29)                             self.height - new_bounds[1] - new_bounds[3],
(29)                             new_bounds[2], new_bounds[3])
(4)     @staticmethod
(4)     def convert_path(gfx_ctx, path, transform):
(8)         wxpath = gfx_ctx.CreatePath()
(8)         for points, code in path.iter_segments(transform):
(12)            if code == Path.MOVETO:
(16)                wxpath.MoveToPoint(*points)
(12)            elif code == Path.LINETO:
(16)                wxpath.AddLineToPoint(*points)
(12)            elif code == Path.CURVE3:
(16)                wxpath.AddQuadCurveToPoint(*points)
(12)            elif code == Path.CURVE4:
(16)                wxpath.AddCurveToPoint(*points)
(12)            elif code == Path.CLOSEPOLY:
(16)                wxpath.CloseSubpath()
(8)         return wxpath
(4)     def draw_path(self, gc, path, transform, rgbFace=None):
(8)         gc.select()
(8)         self.handle_clip_rectangle(gc)
(8)         gfx_ctx = gc.gfx_ctx
(8)         transform = transform + \
(12)            Affine2D().scale(1.0, -1.0).translate(0.0, self.height)
(8)         wxpath = self.convert_path(gfx_ctx, path, transform)
(8)         if rgbFace is not None:
(12)            gfx_ctx.SetBrush(wx.Brush(gc.get_wxcolour(rgbFace)))
(12)            gfx_ctx.DrawPath(wxpath)
(8)         else:
(12)            gfx_ctx.StrokePath(wxpath)
(8)         gc.unselect()
(4)     def draw_image(self, gc, x, y, im):
```

```
161: (8)                    bbox = gc.get_clip_rectangle()
162: (8)                    if bbox is not None:
163: (12)                       l, b, w, h = bbox.bounds
164: (8)                    else:
165: (12)                       l = 0
166: (12)                       b = 0
167: (12)                       w = self.width
168: (12)                       h = self.height
169: (8)                    rows, cols = im.shape[:2]
170: (8)                    bitmap = wx.Bitmap.FromBufferRGBA(cols, rows, im.tobytes())
171: (8)                    gc.select()
172: (8)                    gc.gfx_ctx.DrawBitmap(bitmap, int(l), int(self.height - b),
173: (30)                                         int(w), int(-h))
174: (8)                    gc.unselect()
175: (4)                def draw_text(self, gc, x, y, s, prop, angle, ismath=False, mtext=None):
176: (8)                    if ismath:
177: (12)                       s = cbook.strip_math(s)
178: (8)                    _log.debug("%s - draw_text()", type(self))
179: (8)                    gc.select()
180: (8)                    self.handle_clip_rectangle(gc)
181: (8)                    gfx_ctx = gc.gfx_ctx
182: (8)                    font = self.get_wx_font(s, prop)
183: (8)                    color = gc.get_wxcolour(gc.get_rgb())
184: (8)                    gfx_ctx.SetFont(font, color)
185: (8)                    w, h, d = self.get_text_width_height_descent(s, prop, ismath)
186: (8)                    x = int(x)
187: (8)                    y = int(y - h)
188: (8)                    if angle == 0.0:
189: (12)                       gfx_ctx.DrawText(s, x, y)
190: (8)                    else:
191: (12)                       rads = math.radians(angle)
192: (12)                       xo = h * math.sin(rads)
193: (12)                       yo = h * math.cos(rads)
194: (12)                       gfx_ctx.DrawRotatedText(s, x - xo, y - yo, rads)
195: (8)                    gc.unselect()
196: (4)                def new_gc(self):
197: (8)                    _log.debug("%s - new_gc()", type(self))
198: (8)                    self.gc = GraphicsContextWx(self.bitmap, self)
199: (8)                    self.gc.select()
200: (8)                    self.gc.unselect()
201: (8)                    return self.gc
202: (4)                def get_wx_font(self, s, prop):
203: (8)                    """Return a wx font.  Cache font instances for efficiency."""
204: (8)                    _log.debug("%s - get_wx_font()", type(self))
205: (8)                    key = hash(prop)
206: (8)                    font = self.fontd.get(key)
207: (8)                    if font is not None:
208: (12)                       return font
209: (8)                    size = self.points_to_pixels(prop.get_size_in_points())
210: (8)                    self.fontd[key] = font = wx.Font(  # Cache the font and gc.
211: (12)                       pointSize=round(size),
212: (12)                       family=self.fontnames.get(prop.get_name(), wx.ROMAN),
213: (12)                       style=self.fontangles[prop.get_style()],
214: (12)                       weight=self.fontweights[prop.get_weight()])
215: (8)                    return font
216: (4)                def points_to_pixels(self, points):
217: (8)                    return points * (PIXELS_PER_INCH / 72.0 * self.dpi / 72.0)
218: (0)            class GraphicsContextWx(GraphicsContextBase):
219: (4)                """
220: (4)                The graphics context provides the color, line styles, etc.
221: (4)                This class stores a reference to a wxMemoryDC, and a
222: (4)                wxGraphicsContext that draws to it.  Creating a wxGraphicsContext
223: (4)                seems to be fairly heavy, so these objects are cached based on the
224: (4)                bitmap object that is passed in.
225: (4)                The base GraphicsContext stores colors as an RGB tuple on the unit
226: (4)                interval, e.g., (0.5, 0.0, 1.0).  wxPython uses an int interval, but
227: (4)                since wxPython colour management is rather simple, I have not chosen
228: (4)                to implement a separate colour manager class.
229: (4)                """
```

```
(4)     _capd = {'butt': wx.CAP_BUTT,
(13)             'projecting': wx.CAP_PROJECTING,
(13)             'round': wx.CAP_ROUND}
(4)     _joind = {'bevel': wx.JOIN_BEVEL,
(14)              'miter': wx.JOIN_MITER,
(14)              'round': wx.JOIN_ROUND}
(4)     _cache = weakref.WeakKeyDictionary()
(4)     def __init__(self, bitmap, renderer):
(8)         super().__init__()
(8)         _log.debug("%s - __init__(): %s", type(self), bitmap)
(8)         dc, gfx_ctx = self._cache.get(bitmap, (None, None))
(8)         if dc is None:
(12)            dc = wx.MemoryDC(bitmap)
(12)            gfx_ctx = wx.GraphicsContext.Create(dc)
(12)            gfx_ctx._lastcliprect = None
(12)            self._cache[bitmap] = dc, gfx_ctx
(8)         self.bitmap = bitmap
(8)         self.dc = dc
(8)         self.gfx_ctx = gfx_ctx
(8)         self._pen = wx.Pen('BLACK', 1, wx.SOLID)
(8)         gfx_ctx.SetPen(self._pen)
(8)         self.renderer = renderer
(4)     def select(self):
(8)         """Select the current bitmap into this wxDC instance."""
(8)         if sys.platform == 'win32':
(12)            self.dc.SelectObject(self.bitmap)
(12)            self.IsSelected = True
(4)     def unselect(self):
(8)         """Select a Null bitmap into this wxDC instance."""
(8)         if sys.platform == 'win32':
(12)            self.dc.SelectObject(wx.NullBitmap)
(12)            self.IsSelected = False
(4)     def set_foreground(self, fg, isRGBA=None):
(8)         _log.debug("%s - set_foreground()", type(self))
(8)         self.select()
(8)         super().set_foreground(fg, isRGBA)
(8)         self._pen.SetColour(self.get_wxcolour(self.get_rgb()))
(8)         self.gfx_ctx.SetPen(self._pen)
(8)         self.unselect()
(4)     def set_linewidth(self, w):
(8)         w = float(w)
(8)         _log.debug("%s - set_linewidth()", type(self))
(8)         self.select()
(8)         if 0 < w < 1:
(12)            w = 1
(8)         super().set_linewidth(w)
(8)         lw = int(self.renderer.points_to_pixels(self._linewidth))
(8)         if lw == 0:
(12)            lw = 1
(8)         self._pen.SetWidth(lw)
(8)         self.gfx_ctx.SetPen(self._pen)
(8)         self.unselect()
(4)     def set_capstyle(self, cs):
(8)         _log.debug("%s - set_capstyle()", type(self))
(8)         self.select()
(8)         super().set_capstyle(cs)
(8)         self._pen.SetCap(GraphicsContextWx._capd[self._capstyle])
(8)         self.gfx_ctx.SetPen(self._pen)
(8)         self.unselect()
(4)     def set_joinstyle(self, js):
(8)         _log.debug("%s - set_joinstyle()", type(self))
(8)         self.select()
(8)         super().set_joinstyle(js)
(8)         self._pen.SetJoin(GraphicsContextWx._joind[self._joinstyle])
(8)         self.gfx_ctx.SetPen(self._pen)
(8)         self.unselect()
(4)     def get_wxcolour(self, color):
(8)         """Convert an RGB(A) color to a wx.Colour."""
(8)         _log.debug("%s - get_wx_color()", type(self))
```

```
(8)                 return wx.Colour(*[int(255 * x) for x in color])
(0)         class _FigureCanvasWxBase(FigureCanvasBase, wx.Panel):
(4)             """
(4)             The FigureCanvas contains the figure and does event handling.
(4)             In the wxPython backend, it is derived from wxPanel, and (usually) lives
(4)             inside a frame instantiated by a FigureManagerWx. The parent window
(4)             probably implements a wx.Sizer to control the displayed control size - but
(4)             we give a hint as to our preferred minimum size.
(4)             """
(4)             required_interactive_framework = "wx"
(4)             _timer_cls = TimerWx
(4)             manager_class = _api.classproperty(lambda cls: FigureManagerWx)
(4)             keyvald = {
(8)                 wx.WXK_CONTROL: 'control',
(8)                 wx.WXK_SHIFT: 'shift',
(8)                 wx.WXK_ALT: 'alt',
(8)                 wx.WXK_CAPITAL: 'caps_lock',
(8)                 wx.WXK_LEFT: 'left',
(8)                 wx.WXK_UP: 'up',
(8)                 wx.WXK_RIGHT: 'right',
(8)                 wx.WXK_DOWN: 'down',
(8)                 wx.WXK_ESCAPE: 'escape',
(8)                 wx.WXK_F1: 'f1',
(8)                 wx.WXK_F2: 'f2',
(8)                 wx.WXK_F3: 'f3',
(8)                 wx.WXK_F4: 'f4',
(8)                 wx.WXK_F5: 'f5',
(8)                 wx.WXK_F6: 'f6',
(8)                 wx.WXK_F7: 'f7',
(8)                 wx.WXK_F8: 'f8',
(8)                 wx.WXK_F9: 'f9',
(8)                 wx.WXK_F10: 'f10',
(8)                 wx.WXK_F11: 'f11',
(8)                 wx.WXK_F12: 'f12',
(8)                 wx.WXK_SCROLL: 'scroll_lock',
(8)                 wx.WXK_PAUSE: 'break',
(8)                 wx.WXK_BACK: 'backspace',
(8)                 wx.WXK_RETURN: 'enter',
(8)                 wx.WXK_INSERT: 'insert',
(8)                 wx.WXK_DELETE: 'delete',
(8)                 wx.WXK_HOME: 'home',
(8)                 wx.WXK_END: 'end',
(8)                 wx.WXK_PAGEUP: 'pageup',
(8)                 wx.WXK_PAGEDOWN: 'pagedown',
(8)                 wx.WXK_NUMPAD0: '0',
(8)                 wx.WXK_NUMPAD1: '1',
(8)                 wx.WXK_NUMPAD2: '2',
(8)                 wx.WXK_NUMPAD3: '3',
(8)                 wx.WXK_NUMPAD4: '4',
(8)                 wx.WXK_NUMPAD5: '5',
(8)                 wx.WXK_NUMPAD6: '6',
(8)                 wx.WXK_NUMPAD7: '7',
(8)                 wx.WXK_NUMPAD8: '8',
(8)                 wx.WXK_NUMPAD9: '9',
(8)                 wx.WXK_NUMPAD_ADD: '+',
(8)                 wx.WXK_NUMPAD_SUBTRACT: '-',
(8)                 wx.WXK_NUMPAD_MULTIPLY: '*',
(8)                 wx.WXK_NUMPAD_DIVIDE: '/',
(8)                 wx.WXK_NUMPAD_DECIMAL: 'dec',
(8)                 wx.WXK_NUMPAD_ENTER: 'enter',
(8)                 wx.WXK_NUMPAD_UP: 'up',
(8)                 wx.WXK_NUMPAD_RIGHT: 'right',
(8)                 wx.WXK_NUMPAD_DOWN: 'down',
(8)                 wx.WXK_NUMPAD_LEFT: 'left',
(8)                 wx.WXK_NUMPAD_PAGEUP: 'pageup',
(8)                 wx.WXK_NUMPAD_PAGEDOWN: 'pagedown',
(8)                 wx.WXK_NUMPAD_HOME: 'home',
(8)                 wx.WXK_NUMPAD_END: 'end',
(8)                 wx.WXK_NUMPAD_INSERT: 'insert',
```

```
(8)                     wx.WXK_NUMPAD_DELETE: 'delete',
(4)                 }
(4)                 def __init__(self, parent, id, figure=None):
(8)                     """
(8)                     Initialize a FigureWx instance.
(8)                     - Initialize the FigureCanvasBase and wxPanel parents.
(8)                     - Set event handlers for resize, paint, and keyboard and mouse
(10)                      interaction.
(8)                     """
(8)                     FigureCanvasBase.__init__(self, figure)
(8)                     size = wx.Size(*map(math.ceil, self.figure.bbox.size))
(8)                     if wx.Platform != '__WXMSW__':
(12)                        size = parent.FromDIP(size)
(8)                     wx.Panel.__init__(self, parent, id, size=size)
(8)                     self.bitmap = None
(8)                     self._isDrawn = False
(8)                     self._rubberband_rect = None
(8)                     self._rubberband_pen_black = wx.Pen('BLACK', 1,
wx.PENSTYLE_SHORT_DASH)
(8)                     self._rubberband_pen_white = wx.Pen('WHITE', 1, wx.PENSTYLE_SOLID)
(8)                     self.Bind(wx.EVT_SIZE, self._on_size)
(8)                     self.Bind(wx.EVT_PAINT, self._on_paint)
(8)                     self.Bind(wx.EVT_CHAR_HOOK, self._on_key_down)
(8)                     self.Bind(wx.EVT_KEY_UP, self._on_key_up)
(8)                     self.Bind(wx.EVT_LEFT_DOWN, self._on_mouse_button)
(8)                     self.Bind(wx.EVT_LEFT_DCLICK, self._on_mouse_button)
(8)                     self.Bind(wx.EVT_LEFT_UP, self._on_mouse_button)
(8)                     self.Bind(wx.EVT_MIDDLE_DOWN, self._on_mouse_button)
(8)                     self.Bind(wx.EVT_MIDDLE_DCLICK, self._on_mouse_button)
(8)                     self.Bind(wx.EVT_MIDDLE_UP, self._on_mouse_button)
(8)                     self.Bind(wx.EVT_RIGHT_DOWN, self._on_mouse_button)
(8)                     self.Bind(wx.EVT_RIGHT_DCLICK, self._on_mouse_button)
(8)                     self.Bind(wx.EVT_RIGHT_UP, self._on_mouse_button)
(8)                     self.Bind(wx.EVT_MOUSE_AUX1_DOWN, self._on_mouse_button)
(8)                     self.Bind(wx.EVT_MOUSE_AUX1_UP, self._on_mouse_button)
(8)                     self.Bind(wx.EVT_MOUSE_AUX2_DOWN, self._on_mouse_button)
(8)                     self.Bind(wx.EVT_MOUSE_AUX2_UP, self._on_mouse_button)
(8)                     self.Bind(wx.EVT_MOUSE_AUX1_DCLICK, self._on_mouse_button)
(8)                     self.Bind(wx.EVT_MOUSE_AUX2_DCLICK, self._on_mouse_button)
(8)                     self.Bind(wx.EVT_MOUSEWHEEL, self._on_mouse_wheel)
(8)                     self.Bind(wx.EVT_MOTION, self._on_motion)
(8)                     self.Bind(wx.EVT_ENTER_WINDOW, self._on_enter)
(8)                     self.Bind(wx.EVT_LEAVE_WINDOW, self._on_leave)
(8)                     self.Bind(wx.EVT_MOUSE_CAPTURE_CHANGED, self._on_capture_lost)
(8)                     self.Bind(wx.EVT_MOUSE_CAPTURE_LOST, self._on_capture_lost)
(8)                     self.SetBackgroundStyle(wx.BG_STYLE_PAINT)  # Reduce flicker.
(8)                     self.SetBackgroundColour(wx.WHITE)
(8)                     if wx.Platform == '__WXMAC__':
(12)                        dpiScale = self.GetDPIScaleFactor()
(12)                        self.SetInitialSize(self.GetSize()*(1/dpiScale))
(12)                        self._set_device_pixel_ratio(dpiScale)
(4)                 def Copy_to_Clipboard(self, event=None):
(8)                     """Copy bitmap of canvas to system clipboard."""
(8)                     bmp_obj = wx.BitmapDataObject()
(8)                     bmp_obj.SetBitmap(self.bitmap)
(8)                     if not wx.TheClipboard.IsOpened():
(12)                        open_success = wx.TheClipboard.Open()
(12)                        if open_success:
(16)                            wx.TheClipboard.SetData(bmp_obj)
(16)                            wx.TheClipboard.Flush()
(16)                            wx.TheClipboard.Close()
(4)                 def _update_device_pixel_ratio(self, *args, **kwargs):
(8)                     if self._set_device_pixel_ratio(self.GetDPIScaleFactor()):
(12)                        self.draw()
(4)                 def draw_idle(self):
(8)                     _log.debug("%s - draw_idle()", type(self))
(8)                     self._isDrawn = False  # Force redraw
(8)                     self.Refresh(eraseBackground=False)
(4)                 def flush_events(self):
```

```
436: (8)                    wx.Yield()
437: (4)                def start_event_loop(self, timeout=0):
438: (8)                    if hasattr(self, '_event_loop'):
439: (12)                       raise RuntimeError("Event loop already running")
440: (8)                    timer = wx.Timer(self, id=wx.ID_ANY)
441: (8)                    if timeout > 0:
442: (12)                       timer.Start(int(timeout * 1000), oneShot=True)
443: (12)                       self.Bind(wx.EVT_TIMER, self.stop_event_loop, id=timer.GetId())
444: (8)                    self._event_loop = wx.GUIEventLoop()
445: (8)                    self._event_loop.Run()
446: (8)                    timer.Stop()
447: (4)                def stop_event_loop(self, event=None):
448: (8)                    if hasattr(self, '_event_loop'):
449: (12)                       if self._event_loop.IsRunning():
450: (16)                           self._event_loop.Exit()
451: (12)                       del self._event_loop
452: (4)                def _get_imagesave_wildcards(self):
453: (8)                    """Return the wildcard string for the filesave dialog."""
454: (8)                    default_filetype = self.get_default_filetype()
455: (8)                    filetypes = self.get_supported_filetypes_grouped()
456: (8)                    sorted_filetypes = sorted(filetypes.items())
457: (8)                    wildcards = []
458: (8)                    extensions = []
459: (8)                    filter_index = 0
460: (8)                    for i, (name, exts) in enumerate(sorted_filetypes):
461: (12)                       ext_list = ';'.join(['*.%s' % ext for ext in exts])
462: (12)                       extensions.append(exts[0])
463: (12)                       wildcard = f'{name} ({ext_list})|{ext_list}'
464: (12)                       if default_filetype in exts:
465: (16)                           filter_index = i
466: (12)                       wildcards.append(wildcard)
467: (8)                    wildcards = '|'.join(wildcards)
468: (8)                    return wildcards, extensions, filter_index
469: (4)                def gui_repaint(self, drawDC=None):
470: (8)                    """
471: (8)                    Update the displayed image on the GUI canvas, using the supplied
472: (8)                    wx.PaintDC device context.
473: (8)                    """
474: (8)                    _log.debug("%s - gui_repaint()", type(self))
475: (8)                    if not (self and self.IsShownOnScreen()):
476: (12)                       return
477: (8)                    if not drawDC:  # not called from OnPaint use a ClientDC
478: (12)                       drawDC = wx.ClientDC(self)
479: (8)                    bmp = (self.bitmap.ConvertToImage().ConvertToBitmap()
480: (15)                          if wx.Platform == '__WXMSW__'
481: (18)                             and isinstance(self.figure.canvas.get_renderer(),
RendererWx)
482: (15)                          else self.bitmap)
483: (8)                    drawDC.DrawBitmap(bmp, 0, 0)
484: (8)                    if self._rubberband_rect is not None:
485: (12)                       x0, y0, x1, y1 = map(round, self._rubberband_rect)
486: (12)                       rect = [(x0, y0, x1, y0), (x1, y0, x1, y1),
487: (20)                               (x0, y0, x0, y1), (x0, y1, x1, y1)]
488: (12)                       drawDC.DrawLineList(rect, self._rubberband_pen_white)
489: (12)                       drawDC.DrawLineList(rect, self._rubberband_pen_black)
490: (4)                filetypes = {
491: (8)                    **FigureCanvasBase.filetypes,
492: (8)                    'bmp': 'Windows bitmap',
493: (8)                    'jpeg': 'JPEG',
494: (8)                    'jpg': 'JPEG',
495: (8)                    'pcx': 'PCX',
496: (8)                    'png': 'Portable Network Graphics',
497: (8)                    'tif': 'Tagged Image Format File',
498: (8)                    'tiff': 'Tagged Image Format File',
499: (8)                    'xpm': 'X pixmap',
500: (4)                }
501: (4)                def _on_paint(self, event):
502: (8)                    """Called when wxPaintEvt is generated."""
503: (8)                    _log.debug("%s - _on_paint()", type(self))
```

```
(8)                 drawDC = wx.PaintDC(self)
(8)                 if not self._isDrawn:
(12)                    self.draw(drawDC=drawDC)
(8)                 else:
(12)                    self.gui_repaint(drawDC=drawDC)
(8)                 drawDC.Destroy()
(4)             def _on_size(self, event):
(8)                 """
(8)                 Called when wxEventSize is generated.
(8)                 In this application we attempt to resize to fit the window, so it
(8)                 is better to take the performance hit and redraw the whole window.
(8)                 """
(8)                 self._update_device_pixel_ratio()
(8)                 _log.debug("%s - _on_size()", type(self))
(8)                 sz = self.GetParent().GetSizer()
(8)                 if sz:
(12)                    si = sz.GetItem(self)
(8)                 if sz and si and not si.Proportion and not si.Flag & wx.EXPAND:
(12)                    size = self.GetMinSize()
(8)                 else:
(12)                    size = self.GetClientSize()
(12)                    size.IncTo(self.GetMinSize())
(8)                 if getattr(self, "_width", None):
(12)                    if size == (self._width, self._height):
(16)                        return
(8)                 self._width, self._height = size
(8)                 self._isDrawn = False
(8)                 if self._width <= 1 or self._height <= 1:
(12)                    return  # Empty figure
(8)                 dpival = self.figure.dpi
(8)                 if not wx.Platform == '__WXMSW__':
(12)                    scale = self.GetDPIScaleFactor()
(12)                    dpival /= scale
(8)                 winch = self._width / dpival
(8)                 hinch = self._height / dpival
(8)                 self.figure.set_size_inches(winch, hinch, forward=False)
(8)                 self.Refresh(eraseBackground=False)
(8)                 ResizeEvent("resize_event", self)._process()
(8)                 self.draw_idle()
(4)             @staticmethod
(4)             def _mpl_modifiers(event=None, *, exclude=None):
(8)                 mod_table = [
(12)                    ("ctrl", wx.MOD_CONTROL, wx.WXK_CONTROL),
(12)                    ("alt", wx.MOD_ALT, wx.WXK_ALT),
(12)                    ("shift", wx.MOD_SHIFT, wx.WXK_SHIFT),
(8)                 ]
(8)                 if event is not None:
(12)                    modifiers = event.GetModifiers()
(12)                    return [name for name, mod, key in mod_table
(20)                            if modifiers & mod and exclude != key]
(8)                 else:
(12)                    return [name for name, mod, key in mod_table
(20)                            if wx.GetKeyState(key)]
(4)             def _get_key(self, event):
(8)                 keyval = event.KeyCode
(8)                 if keyval in self.keyvald:
(12)                    key = self.keyvald[keyval]
(8)                 elif keyval < 256:
(12)                    key = chr(keyval)
(12)                    if not event.ShiftDown():
(16)                        key = key.lower()
(8)                 else:
(12)                    return None
(8)                 mods = self._mpl_modifiers(event, exclude=keyval)
(8)                 if "shift" in mods and key.isupper():
(12)                    mods.remove("shift")
(8)                 return "+".join([*mods, key])
(4)             def _mpl_coords(self, pos=None):
(8)                 """
```

```
(8)                     Convert a wx position, defaulting to the current cursor position, to
(8)                     Matplotlib coordinates.
(8)                     """
(8)                     if pos is None:
(12)                        pos = wx.GetMouseState()
(12)                        x, y = self.ScreenToClient(pos.X, pos.Y)
(8)                     else:
(12)                        x, y = pos.X, pos.Y
(8)                     if not wx.Platform == '__WXMSW__':
(12)                        scale = self.GetDPIScaleFactor()
(12)                        return x*scale, self.figure.bbox.height - y*scale
(8)                     else:
(12)                        return x, self.figure.bbox.height - y
(4)                 def _on_key_down(self, event):
(8)                     """Capture key press."""
(8)                     KeyEvent("key_press_event", self,
(17)                             self._get_key(event), *self._mpl_coords(),
(17)                             guiEvent=event)._process()
(8)                     if self:
(12)                        event.Skip()
(4)                 def _on_key_up(self, event):
(8)                     """Release key."""
(8)                     KeyEvent("key_release_event", self,
(17)                             self._get_key(event), *self._mpl_coords(),
(17)                             guiEvent=event)._process()
(8)                     if self:
(12)                        event.Skip()
(4)                 def set_cursor(self, cursor):
(8)                     cursor = wx.Cursor(_api.check_getitem({
(12)                        cursors.MOVE: wx.CURSOR_HAND,
(12)                        cursors.HAND: wx.CURSOR_HAND,
(12)                        cursors.POINTER: wx.CURSOR_ARROW,
(12)                        cursors.SELECT_REGION: wx.CURSOR_CROSS,
(12)                        cursors.WAIT: wx.CURSOR_WAIT,
(12)                        cursors.RESIZE_HORIZONTAL: wx.CURSOR_SIZEWE,
(12)                        cursors.RESIZE_VERTICAL: wx.CURSOR_SIZENS,
(8)                     }, cursor=cursor))
(8)                     self.SetCursor(cursor)
(8)                     self.Refresh()
(4)                 def _set_capture(self, capture=True):
(8)                     """Control wx mouse capture."""
(8)                     if self.HasCapture():
(12)                        self.ReleaseMouse()
(8)                     if capture:
(12)                        self.CaptureMouse()
(4)                 def _on_capture_lost(self, event):
(8)                     """Capture changed or lost"""
(8)                     self._set_capture(False)
(4)                 def _on_mouse_button(self, event):
(8)                     """Start measuring on an axis."""
(8)                     event.Skip()
(8)                     self._set_capture(event.ButtonDown() or event.ButtonDClick())
(8)                     x, y = self._mpl_coords(event)
(8)                     button_map = {
(12)                        wx.MOUSE_BTN_LEFT: MouseButton.LEFT,
(12)                        wx.MOUSE_BTN_MIDDLE: MouseButton.MIDDLE,
(12)                        wx.MOUSE_BTN_RIGHT: MouseButton.RIGHT,
(12)                        wx.MOUSE_BTN_AUX1: MouseButton.BACK,
(12)                        wx.MOUSE_BTN_AUX2: MouseButton.FORWARD,
(8)                     }
(8)                     button = event.GetButton()
(8)                     button = button_map.get(button, button)
(8)                     modifiers = self._mpl_modifiers(event)
(8)                     if event.ButtonDown():
(12)                        MouseEvent("button_press_event", self, x, y, button,
(23)                                    modifiers=modifiers, guiEvent=event)._process()
(8)                     elif event.ButtonDClick():
(12)                        MouseEvent("button_press_event", self, x, y, button,
(23)                                    dblclick=True, modifiers=modifiers,
```

```
(23)                                guiEvent=event)._process()
(8)                     elif event.ButtonUp():
(12)                        MouseEvent("button_release_event", self, x, y, button,
(23)                                    modifiers=modifiers, guiEvent=event)._process()
(4)                 def _on_mouse_wheel(self, event):
(8)                     """Translate mouse wheel events into matplotlib events"""
(8)                     x, y = self._mpl_coords(event)
(8)                     step = event.LinesPerAction * event.WheelRotation / event.WheelDelta
(8)                     event.Skip()
(8)                     if wx.Platform == '__WXMAC__':
(12)                        if not hasattr(self, '_skipwheelevent'):
(16)                            self._skipwheelevent = True
(12)                        elif self._skipwheelevent:
(16)                            self._skipwheelevent = False
(16)                            return  # Return without processing event
(12)                        else:
(16)                            self._skipwheelevent = True
(8)                     MouseEvent("scroll_event", self, x, y, step=step,
(19)                                modifiers=self._mpl_modifiers(event),
(19)                                guiEvent=event)._process()
(4)                 def _on_motion(self, event):
(8)                     """Start measuring on an axis."""
(8)                     event.Skip()
(8)                     MouseEvent("motion_notify_event", self,
(19)                                *self._mpl_coords(event),
(19)                                modifiers=self._mpl_modifiers(event),
(19)                                guiEvent=event)._process()
(4)                 def _on_enter(self, event):
(8)                     """Mouse has entered the window."""
(8)                     event.Skip()
(8)                     LocationEvent("figure_enter_event", self,
(22)                                  *self._mpl_coords(event),
(22)                                  modifiers=self._mpl_modifiers(),
(22)                                  guiEvent=event)._process()
(4)                 def _on_leave(self, event):
(8)                     """Mouse has left the window."""
(8)                     event.Skip()
(8)                     LocationEvent("figure_leave_event", self,
(22)                                  *self._mpl_coords(event),
(22)                                  modifiers=self._mpl_modifiers(),
(22)                                  guiEvent=event)._process()
(0)             class FigureCanvasWx(_FigureCanvasWxBase):
(4)                 def draw(self, drawDC=None):
(8)                     """
(8)                     Render the figure using RendererWx instance renderer, or using a
(8)                     previously defined renderer if none is specified.
(8)                     """
(8)                     _log.debug("%s - draw()", type(self))
(8)                     self.renderer = RendererWx(self.bitmap, self.figure.dpi)
(8)                     self.figure.draw(self.renderer)
(8)                     self._isDrawn = True
(8)                     self.gui_repaint(drawDC=drawDC)
(4)                 def _print_image(self, filetype, filename):
(8)                     bitmap = wx.Bitmap(math.ceil(self.figure.bbox.width),
(27)                                       math.ceil(self.figure.bbox.height))
(8)                     self.figure.draw(RendererWx(bitmap, self.figure.dpi))
(8)                     saved_obj = (bitmap.ConvertToImage()
(21)                                 if cbook.is_writable_file_like(filename)
(21)                                 else bitmap)
(8)                     if not saved_obj.SaveFile(filename, filetype):
(12)                        raise RuntimeError(f'Could not save figure to {filename}')
(8)                     if self._isDrawn:
(12)                        self.draw()
(8)                     if self:
(12)                        self.Refresh()
(4)                 print_bmp = functools.partialmethod(
(8)                     _print_image, wx.BITMAP_TYPE_BMP)
(4)                 print_jpeg = print_jpg = functools.partialmethod(
(8)                     _print_image, wx.BITMAP_TYPE_JPEG)
```

```
(4)                 print_pcx = functools.partialmethod(
(8)                     _print_image, wx.BITMAP_TYPE_PCX)
(4)                 print_png = functools.partialmethod(
(8)                     _print_image, wx.BITMAP_TYPE_PNG)
(4)                 print_tiff = print_tif = functools.partialmethod(
(8)                     _print_image, wx.BITMAP_TYPE_TIF)
(4)                 print_xpm = functools.partialmethod(
(8)                     _print_image, wx.BITMAP_TYPE_XPM)
(0)             class FigureFrameWx(wx.Frame):
(4)                 def __init__(self, num, fig, *, canvas_class):
(8)                     if wx.Platform == '__WXMSW__':
(12)                        pos = wx.DefaultPosition
(8)                     else:
(12)                        pos = wx.Point(20, 20)
(8)                     super().__init__(parent=None, id=-1, pos=pos)
(8)                     _log.debug("%s - __init__()", type(self))
(8)                     _set_frame_icon(self)
(8)                     self.canvas = canvas_class(self, -1, fig)
(8)                     manager = FigureManagerWx(self.canvas, num, self)
(8)                     toolbar = self.canvas.manager.toolbar
(8)                     if toolbar is not None:
(12)                        self.SetToolBar(toolbar)
(8)                     w, h = map(math.ceil, fig.bbox.size)
(8)                     self.canvas.SetInitialSize(self.FromDIP(wx.Size(w, h)))
(8)                     self.canvas.SetMinSize(self.FromDIP(wx.Size(2, 2)))
(8)                     self.canvas.SetFocus()
(8)                     self.Fit()
(8)                     self.Bind(wx.EVT_CLOSE, self._on_close)
(4)                 def _on_close(self, event):
(8)                     _log.debug("%s - on_close()", type(self))
(8)                     CloseEvent("close_event", self.canvas)._process()
(8)                     self.canvas.stop_event_loop()
(8)                     self.canvas.manager.frame = None
(8)                     Gcf.destroy(self.canvas.manager)
(8)                     try:  # See issue 2941338.
(12)                        self.canvas.mpl_disconnect(self.canvas.toolbar._id_drag)
(8)                     except AttributeError:  # If there's no toolbar.
(12)                        pass
(8)                     event.Skip()
(0)             class FigureManagerWx(FigureManagerBase):
(4)                 """
(4)                 Container/controller for the FigureCanvas and GUI frame.
(4)                 It is instantiated by Gcf whenever a new figure is created.  Gcf is
(4)                 responsible for managing multiple instances of FigureManagerWx.
(4)                 Attributes
(4)                 ----------
(4)                 canvas : `FigureCanvas`
(8)                     a FigureCanvasWx(wx.Panel) instance
(4)                 window : wxFrame
(8)                     a wxFrame instance - wxpython.org/Phoenix/docs/html/Frame.html
(4)                 """
(4)                 def __init__(self, canvas, num, frame):
(8)                     _log.debug("%s - __init__()", type(self))
(8)                     self.frame = self.window = frame
(8)                     super().__init__(canvas, num)
(4)                 @classmethod
(4)                 def create_with_canvas(cls, canvas_class, figure, num):
(8)                     wxapp = wx.GetApp() or _create_wxapp()
(8)                     frame = FigureFrameWx(num, figure, canvas_class=canvas_class)
(8)                     manager = figure.canvas.manager
(8)                     if mpl.is_interactive():
(12)                        manager.frame.Show()
(12)                        figure.canvas.draw_idle()
(8)                     return manager
(4)                 @classmethod
(4)                 def start_main_loop(cls):
(8)                     if not wx.App.IsMainLoopRunning():
(12)                        wxapp = wx.GetApp()
(12)                        if wxapp is not None:
```

```
780: (16)                            wxapp.MainLoop()
781: (4)                def show(self):
782: (8)                    self.frame.Show()
783: (8)                    self.canvas.draw()
784: (8)                    if mpl.rcParams['figure.raise_window']:
785: (12)                       self.frame.Raise()
786: (4)                def destroy(self, *args):
787: (8)                    _log.debug("%s - destroy()", type(self))
788: (8)                    frame = self.frame
789: (8)                    if frame:  # Else, may have been already deleted, e.g. when closing.
790: (12)                       wx.CallAfter(frame.Close)
791: (4)                def full_screen_toggle(self):
792: (8)                    self.frame.ShowFullScreen(not self.frame.IsFullScreen())
793: (4)                def get_window_title(self):
794: (8)                    return self.window.GetTitle()
795: (4)                def set_window_title(self, title):
796: (8)                    self.window.SetTitle(title)
797: (4)                def resize(self, width, height):
798: (8)                    self.window.SetSize(self.window.ClientToWindowSize(wx.Size(
799: (12)                       math.ceil(width), math.ceil(height))))
800: (0)            def _load_bitmap(filename):
801: (4)                """
802: (4)                Load a wx.Bitmap from a file in the "images" directory of the Matplotlib
803: (4)                data.
804: (4)                """
805: (4)                return wx.Bitmap(str(cbook._get_data_path('images', filename)))
806: (0)            def _set_frame_icon(frame):
807: (4)                bundle = wx.IconBundle()
808: (4)                for image in ('matplotlib.png', 'matplotlib_large.png'):
809: (8)                    icon = wx.Icon(_load_bitmap(image))
810: (8)                    if not icon.IsOk():
811: (12)                       return
812: (8)                    bundle.AddIcon(icon)
813: (4)                frame.SetIcons(bundle)
814: (0)            class NavigationToolbar2Wx(NavigationToolbar2, wx.ToolBar):
815: (4)                def __init__(self, canvas, coordinates=True, *, style=wx.TB_BOTTOM):
816: (8)                    wx.ToolBar.__init__(self, canvas.GetParent(), -1, style=style)
817: (8)                    if wx.Platform == '__WXMAC__':
818: (12)
self.SetToolBitmapSize(self.GetToolBitmapSize()*self.GetDPIScaleFactor())
819: (8)                    self.wx_ids = {}
820: (8)                    for text, tooltip_text, image_file, callback in self.toolitems:
821: (12)                       if text is None:
822: (16)                           self.AddSeparator()
823: (16)                           continue
824: (12)                       self.wx_ids[text] = (
825: (16)                           self.AddTool(
826: (20)                               -1,
827: (20)                               bitmap=self._icon(f"{image_file}.svg"),
828: (20)                               bmpDisabled=wx.NullBitmap,
829: (20)                               label=text, shortHelp=tooltip_text,
830: (20)                               kind=(wx.ITEM_CHECK if text in ["Pan", "Zoom"]
831: (26)                                     else wx.ITEM_NORMAL))
832: (16)                           .Id)
833: (12)                       self.Bind(wx.EVT_TOOL, getattr(self, callback),
834: (22)                                 id=self.wx_ids[text])
835: (8)                    self._coordinates = coordinates
836: (8)                    if self._coordinates:
837: (12)                       self.AddStretchableSpace()
838: (12)                       self._label_text = wx.StaticText(self, style=wx.ALIGN_RIGHT)
839: (12)                       self.AddControl(self._label_text)
840: (8)                    self.Realize()
841: (8)                    NavigationToolbar2.__init__(self, canvas)
842: (4)                @staticmethod
843: (4)                def _icon(name):
844: (8)                    """
845: (8)                    Construct a `wx.Bitmap` suitable for use as icon from an image file
846: (8)                    *name*, including the extension and relative to Matplotlib's "images"
847: (8)                    data directory.
```

```
848: (8)                    """
849: (8)                    try:
850: (12)                       dark = wx.SystemSettings.GetAppearance().IsDark()
851: (8)                    except AttributeError:  # wxpython < 4.1
852: (12)                       bg = wx.SystemSettings.GetColour(wx.SYS_COLOUR_WINDOW)
853: (12)                       fg = wx.SystemSettings.GetColour(wx.SYS_COLOUR_WINDOWTEXT)
854: (12)                       bg_lum = (.299 * bg.red + .587 * bg.green + .114 * bg.blue) / 255
855: (12)                       fg_lum = (.299 * fg.red + .587 * fg.green + .114 * fg.blue) / 255
856: (12)                       dark = fg_lum - bg_lum > .2
857: (8)                    path = cbook._get_data_path('images', name)
858: (8)                    if path.suffix == '.svg':
859: (12)                       svg = path.read_bytes()
860: (12)                       if dark:
861: (16)                           svg = svg.replace(b'fill:black;', b'fill:white;')
862: (12)                       toolbarIconSize = wx.ArtProvider().GetDIPSizeHint(wx.ART_TOOLBAR)
863: (12)                       return wx.BitmapBundle.FromSVG(svg, toolbarIconSize)
864: (8)                    else:
865: (12)                       pilimg = PIL.Image.open(path)
866: (12)                       image = np.array(pilimg.convert("RGBA"))
867: (12)                       if dark:
868: (16)                           fg = wx.SystemSettings.GetColour(wx.SYS_COLOUR_WINDOWTEXT)
869: (16)                           black_mask = (image[..., :3] == 0).all(axis=-1)
870: (16)                           image[black_mask, :3] = (fg.Red(), fg.Green(), fg.Blue())
871: (12)                       return wx.Bitmap.FromBufferRGBA(
872: (16)                           image.shape[1], image.shape[0], image.tobytes())
873: (4)                def _update_buttons_checked(self):
874: (8)                    if "Pan" in self.wx_ids:
875: (12)                       self.ToggleTool(self.wx_ids["Pan"], self.mode.name == "PAN")
876: (8)                    if "Zoom" in self.wx_ids:
877: (12)                       self.ToggleTool(self.wx_ids["Zoom"], self.mode.name == "ZOOM")
878: (4)                def zoom(self, *args):
879: (8)                    super().zoom(*args)
880: (8)                    self._update_buttons_checked()
881: (4)                def pan(self, *args):
882: (8)                    super().pan(*args)
883: (8)                    self._update_buttons_checked()
884: (4)                def save_figure(self, *args):
885: (8)                    filetypes, exts, filter_index = self.canvas._get_imagesave_wildcards()
886: (8)                    default_file = self.canvas.get_default_filename()
887: (8)                    dialog = wx.FileDialog(
888: (12)                       self.canvas.GetParent(), "Save to file",
889: (12)                       mpl.rcParams["savefig.directory"], default_file, filetypes,
890: (12)                       wx.FD_SAVE | wx.FD_OVERWRITE_PROMPT)
891: (8)                    dialog.SetFilterIndex(filter_index)
892: (8)                    if dialog.ShowModal() == wx.ID_OK:
893: (12)                       path = pathlib.Path(dialog.GetPath())
894: (12)                       _log.debug('%s - Save file path: %s', type(self), path)
895: (12)                       fmt = exts[dialog.GetFilterIndex()]
896: (12)                       ext = path.suffix[1:]
897: (12)                       if ext in self.canvas.get_supported_filetypes() and fmt != ext:
898: (16)                           _log.warning('extension %s did not match the selected '
899: (29)                                        'image type %s; going with %s',
900: (29)                                        ext, fmt, ext)
901: (16)                           fmt = ext
902: (12)                       if mpl.rcParams["savefig.directory"]:
903: (16)                           mpl.rcParams["savefig.directory"] = str(path.parent)
904: (12)                       try:
905: (16)                           self.canvas.figure.savefig(path, format=fmt)
906: (12)                       except Exception as e:
907: (16)                           dialog = wx.MessageDialog(
908: (20)                               parent=self.canvas.GetParent(), message=str(e),
909: (20)                               caption='Matplotlib error')
910: (16)                           dialog.ShowModal()
911: (16)                           dialog.Destroy()
912: (4)                def draw_rubberband(self, event, x0, y0, x1, y1):
913: (8)                    height = self.canvas.figure.bbox.height
914: (8)                    sf = 1 if wx.Platform == '__WXMSW__' else
self.canvas.GetDPIScaleFactor()
915: (8)                    self.canvas._rubberband_rect = (x0/sf, (height - y0)/sf,
```

```
(40)                                                x1/sf, (height - y1)/sf)
(8)                     self.canvas.Refresh()
(4)                 def remove_rubberband(self):
(8)                     self.canvas._rubberband_rect = None
(8)                     self.canvas.Refresh()
(4)                 def set_message(self, s):
(8)                     if self._coordinates:
(12)                        self._label_text.SetLabel(s)
(4)                 def set_history_buttons(self):
(8)                     can_backward = self._nav_stack._pos > 0
(8)                     can_forward = self._nav_stack._pos < len(self._nav_stack) - 1
(8)                     if 'Back' in self.wx_ids:
(12)                        self.EnableTool(self.wx_ids['Back'], can_backward)
(8)                     if 'Forward' in self.wx_ids:
(12)                        self.EnableTool(self.wx_ids['Forward'], can_forward)
(0)             class ToolbarWx(ToolContainerBase, wx.ToolBar):
(4)                 _icon_extension = '.svg'
(4)                 def __init__(self, toolmanager, parent=None, style=wx.TB_BOTTOM):
(8)                     if parent is None:
(12)                        parent = toolmanager.canvas.GetParent()
(8)                     ToolContainerBase.__init__(self, toolmanager)
(8)                     wx.ToolBar.__init__(self, parent, -1, style=style)
(8)                     self._space = self.AddStretchableSpace()
(8)                     self._label_text = wx.StaticText(self, style=wx.ALIGN_RIGHT)
(8)                     self.AddControl(self._label_text)
(8)                     self._toolitems = {}
(8)                     self._groups = {}  # Mapping of groups to the separator after them.
(4)                 def _get_tool_pos(self, tool):
(8)                     """
(8)                     Find the position (index) of a wx.ToolBarToolBase in a ToolBar.
(8)                     ``ToolBar.GetToolPos`` is not useful because wx assigns the same Id to
(8)                     all Separators and StretchableSpaces.
(8)                     """
(8)                     pos, = [pos for pos in range(self.ToolsCount)
(16)                            if self.GetToolByPos(pos) == tool]
(8)                     return pos
(4)                 def add_toolitem(self, name, group, position, image_file, description,
(21)                                 toggle):
(8)                     if group not in self._groups:
(12)                        self._groups[group] = self.InsertSeparator(
(16)                            self._get_tool_pos(self._space))
(8)                     sep = self._groups[group]
(8)                     seps = [t for t in map(self.GetToolByPos, range(self.ToolsCount))
(16)                            if t.IsSeparator() and not t.IsStretchableSpace()]
(8)                     if position >= 0:
(12)                        start = (0 if sep == seps[0]
(21)                                 else self._get_tool_pos(seps[seps.index(sep) - 1]) + 1)
(8)                     else:
(12)                        start = self._get_tool_pos(sep) + 1
(8)                     idx = start + position
(8)                     if image_file:
(12)                        bmp = NavigationToolbar2Wx._icon(image_file)
(12)                        kind = wx.ITEM_NORMAL if not toggle else wx.ITEM_CHECK
(12)                        tool = self.InsertTool(idx, -1, name, bmp, wx.NullBitmap, kind,
(35)                                               description or "")
(8)                     else:
(12)                        size = (self.GetTextExtent(name)[0] + 10, -1)
(12)                        if toggle:
(16)                            control = wx.ToggleButton(self, -1, name, size=size)
(12)                        else:
(16)                            control = wx.Button(self, -1, name, size=size)
(12)                        tool = self.InsertControl(idx, control, label=name)
(8)                     self.Realize()
(8)                     def handler(event):
(12)                        self.trigger_tool(name)
(8)                     if image_file:
(12)                        self.Bind(wx.EVT_TOOL, handler, tool)
(8)                     else:
(12)                        control.Bind(wx.EVT_LEFT_DOWN, handler)
```

```
(8)                 self._toolitems.setdefault(name, [])
(8)                 self._toolitems[name].append((tool, handler))
(4)             def toggle_toolitem(self, name, toggled):
(8)                 if name not in self._toolitems:
(12)                    return
(8)                 for tool, handler in self._toolitems[name]:
(12)                    if not tool.IsControl():
(16)                        self.ToggleTool(tool.Id, toggled)
(12)                    else:
(16)                        tool.GetControl().SetValue(toggled)
(8)                 self.Refresh()
(4)             def remove_toolitem(self, name):
(8)                 for tool, handler in self._toolitems.pop(name, []):
(12)                    self.DeleteTool(tool.Id)
(4)             def set_message(self, s):
(8)                 self._label_text.SetLabel(s)
(0)         @backend_tools._register_tool_class(_FigureCanvasWxBase)
(0)         class ConfigureSubplotsWx(backend_tools.ConfigureSubplotsBase):
(4)             def trigger(self, *args):
(8)                 NavigationToolbar2Wx.configure_subplots(self)
(0)         @backend_tools._register_tool_class(_FigureCanvasWxBase)
(0)         class SaveFigureWx(backend_tools.SaveFigureBase):
(4)             def trigger(self, *args):
(8)                 NavigationToolbar2Wx.save_figure(
(12)                    self._make_classic_style_pseudo_toolbar())
(0)         @backend_tools._register_tool_class(_FigureCanvasWxBase)
(0)         class RubberbandWx(backend_tools.RubberbandBase):
(4)             def draw_rubberband(self, x0, y0, x1, y1):
(8)                 NavigationToolbar2Wx.draw_rubberband(
(12)                    self._make_classic_style_pseudo_toolbar(), None, x0, y0, x1, y1)
(4)             def remove_rubberband(self):
(8)                 NavigationToolbar2Wx.remove_rubberband(
(12)                    self._make_classic_style_pseudo_toolbar())
(0)         class _HelpDialog(wx.Dialog):
(4)             _instance = None  # a reference to an open dialog singleton
(4)             headers = [("Action", "Shortcuts", "Description")]
(4)             widths = [100, 140, 300]
(4)             def __init__(self, parent, help_entries):
(8)                 super().__init__(parent, title="Help",
(25)                                 style=wx.DEFAULT_DIALOG_STYLE | wx.RESIZE_BORDER)
(8)                 sizer = wx.BoxSizer(wx.VERTICAL)
(8)                 grid_sizer = wx.FlexGridSizer(0, 3, 8, 6)
(8)                 bold = self.GetFont().MakeBold()
(8)                 for r, row in enumerate(self.headers + help_entries):
(12)                    for (col, width) in zip(row, self.widths):
(16)                        label = wx.StaticText(self, label=col)
(16)                        if r == 0:
(20)                            label.SetFont(bold)
(16)                        label.Wrap(width)
(16)                        grid_sizer.Add(label, 0, 0, 0)
(8)                 sizer.Add(grid_sizer, 0, wx.ALL, 6)
(8)                 ok = wx.Button(self, wx.ID_OK)
(8)                 sizer.Add(ok, 0, wx.ALIGN_CENTER_HORIZONTAL | wx.ALL, 8)
(8)                 self.SetSizer(sizer)
(8)                 sizer.Fit(self)
(8)                 self.Layout()
(8)                 self.Bind(wx.EVT_CLOSE, self._on_close)
(8)                 ok.Bind(wx.EVT_BUTTON, self._on_close)
(4)             def _on_close(self, event):
(8)                 _HelpDialog._instance = None  # remove global reference
(8)                 self.DestroyLater()
(8)                 event.Skip()
(4)             @classmethod
(4)             def show(cls, parent, help_entries):
(8)                 if cls._instance:
(12)                    cls._instance.Raise()
(12)                    return
(8)                 cls._instance = cls(parent, help_entries)
(8)                 cls._instance.Show()
```

```
1054: (0)           @backend_tools._register_tool_class(_FigureCanvasWxBase)
1055: (0)           class HelpWx(backend_tools.ToolHelpBase):
1056: (4)               def trigger(self, *args):
1057: (8)                   _HelpDialog.show(self.figure.canvas.GetTopLevelParent(),
1058: (25)                                   self._get_help_entries())
1059: (0)           @backend_tools._register_tool_class(_FigureCanvasWxBase)
1060: (0)           class ToolCopyToClipboardWx(backend_tools.ToolCopyToClipboardBase):
1061: (4)               def trigger(self, *args, **kwargs):
1062: (8)                   if not self.canvas._isDrawn:
1063: (12)                      self.canvas.draw()
1064: (8)                   if not self.canvas.bitmap.IsOk() or not wx.TheClipboard.Open():
1065: (12)                      return
1066: (8)                   try:
1067: (12)                      wx.TheClipboard.SetData(wx.BitmapDataObject(self.canvas.bitmap))
1068: (8)                   finally:
1069: (12)                      wx.TheClipboard.Close()
1070: (0)           FigureManagerWx._toolbar2_class = NavigationToolbar2Wx
1071: (0)           FigureManagerWx._toolmanager_toolbar_class = ToolbarWx
1072: (0)           @_Backend.export
1073: (0)           class _BackendWx(_Backend):
1074: (4)               FigureCanvas = FigureCanvasWx
1075: (4)               FigureManager = FigureManagerWx
1076: (4)               mainloop = FigureManagerWx.start_main_loop
```

File 109 - backend_wxagg.py:

```
1: (0)              import wx
2: (0)              from .backend_agg import FigureCanvasAgg
3: (0)              from .backend_wx import _BackendWx, _FigureCanvasWxBase
4: (0)              from .backend_wx import (  # noqa: F401 # pylint: disable=W0611
5: (4)                  NavigationToolbar2Wx as NavigationToolbar2WxAgg)
6: (0)              class FigureCanvasWxAgg(FigureCanvasAgg, _FigureCanvasWxBase):
7: (4)                  def draw(self, drawDC=None):
8: (8)                      """
9: (8)                      Render the figure using agg.
10: (8)                     """
11: (8)                     FigureCanvasAgg.draw(self)
12: (8)                     self.bitmap = self._create_bitmap()
13: (8)                     self._isDrawn = True
14: (8)                     self.gui_repaint(drawDC=drawDC)
15: (4)                 def blit(self, bbox=None):
16: (8)                     bitmap = self._create_bitmap()
17: (8)                     if bbox is None:
18: (12)                        self.bitmap = bitmap
19: (8)                     else:
20: (12)                        srcDC = wx.MemoryDC(bitmap)
21: (12)                        destDC = wx.MemoryDC(self.bitmap)
22: (12)                        x = int(bbox.x0)
23: (12)                        y = int(self.bitmap.GetHeight() - bbox.y1)
24: (12)                        destDC.Blit(x, y, int(bbox.width), int(bbox.height), srcDC, x, y)
25: (12)                        destDC.SelectObject(wx.NullBitmap)
26: (12)                        srcDC.SelectObject(wx.NullBitmap)
27: (8)                     self.gui_repaint()
28: (4)                 def _create_bitmap(self):
29: (8)                     """Create a wx.Bitmap from the renderer RGBA buffer"""
30: (8)                     rgba = self.get_renderer().buffer_rgba()
31: (8)                     h, w, _ = rgba.shape
32: (8)                     bitmap = wx.Bitmap.FromBufferRGBA(w, h, rgba)
33: (8)                     bitmap.SetScaleFactor(self.GetDPIScaleFactor())
34: (8)                     return bitmap
35: (0)             @_BackendWx.export
36: (0)             class _BackendWxAgg(_BackendWx):
37: (4)                 FigureCanvas = FigureCanvasWxAgg
```

File 110 - qt_compat.py:

```
1: (0)              """
2: (0)              Qt binding and backend selector.
3: (0)              The selection logic is as follows:
4: (0)              - if any of PyQt6, PySide6, PyQt5, or PySide2 have already been
5: (2)                imported (checked in that order), use it;
6: (0)              - otherwise, if the QT_API environment variable (used by Enthought) is set,
use
7: (2)                it to determine which binding to use;
8: (0)              - otherwise, use whatever the rcParams indicate.
9: (0)              """
10: (0)             import operator
11: (0)             import os
12: (0)             import platform
13: (0)             import sys
14: (0)             from packaging.version import parse as parse_version
15: (0)             import matplotlib as mpl
16: (0)             from . import _QT_FORCE_QT5_BINDING
17: (0)             QT_API_PYQT6 = "PyQt6"
18: (0)             QT_API_PYSIDE6 = "PySide6"
19: (0)             QT_API_PYQT5 = "PyQt5"
20: (0)             QT_API_PYSIDE2 = "PySide2"
21: (0)             QT_API_ENV = os.environ.get("QT_API")
22: (0)             if QT_API_ENV is not None:
23: (4)                 QT_API_ENV = QT_API_ENV.lower()
24: (0)             _ETS = {  # Mapping of QT_API_ENV to requested binding.
25: (4)                 "pyqt6": QT_API_PYQT6, "pyside6": QT_API_PYSIDE6,
26: (4)                 "pyqt5": QT_API_PYQT5, "pyside2": QT_API_PYSIDE2,
27: (0)             }
28: (0)             if sys.modules.get("PyQt6.QtCore"):
29: (4)                 QT_API = QT_API_PYQT6
30: (0)             elif sys.modules.get("PySide6.QtCore"):
31: (4)                 QT_API = QT_API_PYSIDE6
32: (0)             elif sys.modules.get("PyQt5.QtCore"):
33: (4)                 QT_API = QT_API_PYQT5
34: (0)             elif sys.modules.get("PySide2.QtCore"):
35: (4)                 QT_API = QT_API_PYSIDE2
36: (0)             elif (mpl.rcParams._get_backend_or_none() or "").lower().startswith("qt5"):
37: (4)                 if QT_API_ENV in ["pyqt5", "pyside2"]:
38: (8)                     QT_API = _ETS[QT_API_ENV]
39: (4)                 else:
40: (8)                     _QT_FORCE_QT5_BINDING = True  # noqa
41: (8)                     QT_API = None
42: (0)             elif QT_API_ENV is None:
43: (4)                 QT_API = None
44: (0)             elif QT_API_ENV in _ETS:
45: (4)                 QT_API = _ETS[QT_API_ENV]
46: (0)             else:
47: (4)                 raise RuntimeError(
48: (8)                     "The environment variable QT_API has the unrecognized value {!r}; "
49: (8)                     "valid values are {}".format(QT_API_ENV, ", ".join(_ETS)))
50: (0)             def _setup_pyqt5plus():
51: (4)                 global QtCore, QtGui, QtWidgets, __version__
52: (4)                 global _isdeleted, _to_int
53: (4)                 if QT_API == QT_API_PYQT6:
54: (8)                     from PyQt6 import QtCore, QtGui, QtWidgets, sip
55: (8)                     __version__ = QtCore.PYQT_VERSION_STR
56: (8)                     QtCore.Signal = QtCore.pyqtSignal
57: (8)                     QtCore.Slot = QtCore.pyqtSlot
58: (8)                     QtCore.Property = QtCore.pyqtProperty
59: (8)                     _isdeleted = sip.isdeleted
60: (8)                     _to_int = operator.attrgetter('value')
61: (4)                 elif QT_API == QT_API_PYSIDE6:
62: (8)                     from PySide6 import QtCore, QtGui, QtWidgets, __version__
63: (8)                     import shiboken6
64: (8)                     def _isdeleted(obj): return not shiboken6.isValid(obj)
65: (8)                     if parse_version(__version__) >= parse_version('6.4'):
66: (12)                        _to_int = operator.attrgetter('value')
67: (8)                     else:
```

```
68: (12)                        _to_int = int
69: (4)                 elif QT_API == QT_API_PYQT5:
70: (8)                     from PyQt5 import QtCore, QtGui, QtWidgets
71: (8)                     import sip
72: (8)                     __version__ = QtCore.PYQT_VERSION_STR
73: (8)                     QtCore.Signal = QtCore.pyqtSignal
74: (8)                     QtCore.Slot = QtCore.pyqtSlot
75: (8)                     QtCore.Property = QtCore.pyqtProperty
76: (8)                     _isdeleted = sip.isdeleted
77: (8)                     _to_int = int
78: (4)                 elif QT_API == QT_API_PYSIDE2:
79: (8)                     from PySide2 import QtCore, QtGui, QtWidgets, __version__
80: (8)                     try:
81: (12)                        from PySide2 import shiboken2
82: (8)                     except ImportError:
83: (12)                        import shiboken2
84: (8)                     def _isdeleted(obj):
85: (12)                        return not shiboken2.isValid(obj)
86: (8)                     _to_int = int
87: (4)                 else:
88: (8)                     raise AssertionError(f"Unexpected QT_API: {QT_API}")
89: (0)             if QT_API in [QT_API_PYQT6, QT_API_PYQT5, QT_API_PYSIDE6, QT_API_PYSIDE2]:
90: (4)                 _setup_pyqt5plus()
91: (0)             elif QT_API is None:  # See above re: dict.__getitem__.
92: (4)                 if _QT_FORCE_QT5_BINDING:
93: (8)                     _candidates = [
94: (12)                        (_setup_pyqt5plus, QT_API_PYQT5),
95: (12)                        (_setup_pyqt5plus, QT_API_PYSIDE2),
96: (8)                     ]
97: (4)                 else:
98: (8)                     _candidates = [
99: (12)                        (_setup_pyqt5plus, QT_API_PYQT6),
100: (12)                       (_setup_pyqt5plus, QT_API_PYSIDE6),
101: (12)                       (_setup_pyqt5plus, QT_API_PYQT5),
102: (12)                       (_setup_pyqt5plus, QT_API_PYSIDE2),
103: (8)                    ]
104: (4)                for _setup, QT_API in _candidates:
105: (8)                    try:
106: (12)                       _setup()
107: (8)                    except ImportError:
108: (12)                       continue
109: (8)                    break
110: (4)                else:
111: (8)                    raise ImportError(
112: (12)                       "Failed to import any of the following Qt binding modules: {}"
113: (12)                       .format(", ".join([QT_API for _, QT_API in _candidates]))
114: (8)                    )
115: (0)            else:  # We should not get there.
116: (4)                raise AssertionError(f"Unexpected QT_API: {QT_API}")
117: (0)            _version_info = tuple(QtCore.QLibraryInfo.version().segments())
118: (0)            if _version_info < (5, 12):
119: (4)                raise ImportError(
120: (8)                    f"The Qt version imported is "
121: (8)                    f"{QtCore.QLibraryInfo.version().toString()} but Matplotlib requires "
122: (8)                    f"Qt>=5.12")
123: (0)            if (sys.platform == 'darwin' and
124: (8)                    parse_version(platform.mac_ver()[0]) >= parse_version("10.16") and
125: (8)                    _version_info < (5, 15, 2)):
126: (4)                os.environ.setdefault("QT_MAC_WANTS_LAYER", "1")
127: (0)            def _exec(obj):
128: (4)                obj.exec() if hasattr(obj, "exec") else obj.exec_()
```

File 111 - backend_wxcairo.py:

```
1: (0)              import wx.lib.wxcairo as wxcairo
2: (0)              from .backend_cairo import cairo, FigureCanvasCairo
3: (0)              from .backend_wx import _BackendWx, _FigureCanvasWxBase
```

```
4: (0)              from .backend_wx import (  # noqa: F401 # pylint: disable=W0611
5: (4)                  NavigationToolbar2Wx as NavigationToolbar2WxCairo)
6: (0)              class FigureCanvasWxCairo(FigureCanvasCairo, _FigureCanvasWxBase):
7: (4)                  def draw(self, drawDC=None):
8: (8)                      size = self.figure.bbox.size.astype(int)
9: (8)                      surface = cairo.ImageSurface(cairo.FORMAT_ARGB32, *size)
10: (8)                     self._renderer.set_context(cairo.Context(surface))
11: (8)                     self._renderer.dpi = self.figure.dpi
12: (8)                     self.figure.draw(self._renderer)
13: (8)                     self.bitmap = wxcairo.BitmapFromImageSurface(surface)
14: (8)                     self._isDrawn = True
15: (8)                     self.gui_repaint(drawDC=drawDC)
16: (0)             @_BackendWx.export
17: (0)             class _BackendWxCairo(_BackendWx):
18: (4)                 FigureCanvas = FigureCanvasWxCairo
```

File 112 - registry.py:

```
1: (0)              from enum import Enum
2: (0)              import importlib
3: (0)              class BackendFilter(Enum):
4: (4)                  """
5: (4)                  Filter used with
:meth:`~matplotlib.backends.registry.BackendRegistry.list_builtin`
6: (4)                  .. versionadded:: 3.9
7: (4)                  """
8: (4)                  INTERACTIVE = 0
9: (4)                  NON_INTERACTIVE = 1
10: (0)             class BackendRegistry:
11: (4)                 """
12: (4)                 Registry of backends available within Matplotlib.
13: (4)                 This is the single source of truth for available backends.
14: (4)                 All use of ``BackendRegistry`` should be via the singleton instance
15: (4)                 ``backend_registry`` which can be imported from ``matplotlib.backends``.
16: (4)                 Each backend has a name, a module name containing the backend code, and an
17: (4)                 optional GUI framework that must be running if the backend is interactive.
18: (4)                 There are three sources of backends: built-in (source code is within the
19: (4)                 Matplotlib repository), explicit ``module://some.backend`` syntax (backend
is
20: (4)                 obtained by loading the module), or via an entry point (self-registering
21: (4)                 backend in an external package).
22: (4)                 .. versionadded:: 3.9
23: (4)                 """
24: (4)                 _BUILTIN_BACKEND_TO_GUI_FRAMEWORK = {
25: (8)                     "gtk3agg": "gtk3",
26: (8)                     "gtk3cairo": "gtk3",
27: (8)                     "gtk4agg": "gtk4",
28: (8)                     "gtk4cairo": "gtk4",
29: (8)                     "macosx": "macosx",
30: (8)                     "nbagg": "nbagg",
31: (8)                     "notebook": "nbagg",
32: (8)                     "qtagg": "qt",
33: (8)                     "qtcairo": "qt",
34: (8)                     "qt5agg": "qt5",
35: (8)                     "qt5cairo": "qt5",
36: (8)                     "tkagg": "tk",
37: (8)                     "tkcairo": "tk",
38: (8)                     "webagg": "webagg",
39: (8)                     "wx": "wx",
40: (8)                     "wxagg": "wx",
41: (8)                     "wxcairo": "wx",
42: (8)                     "agg": "headless",
43: (8)                     "cairo": "headless",
44: (8)                     "pdf": "headless",
45: (8)                     "pgf": "headless",
46: (8)                     "ps": "headless",
47: (8)                     "svg": "headless",
```

```
48: (8)                     "template": "headless",
49: (4)                 }
50: (4)                 _GUI_FRAMEWORK_TO_BACKEND = {
51: (8)                     "gtk3": "gtk3agg",
52: (8)                     "gtk4": "gtk4agg",
53: (8)                     "headless": "agg",
54: (8)                     "macosx": "macosx",
55: (8)                     "qt": "qtagg",
56: (8)                     "qt5": "qt5agg",
57: (8)                     "qt6": "qtagg",
58: (8)                     "tk": "tkagg",
59: (8)                     "wx": "wxagg",
60: (4)                 }
61: (4)                 def __init__(self):
62: (8)                     self._loaded_entry_points = False
63: (8)                     self._backend_to_gui_framework = {}
64: (8)                     self._name_to_module = {
65: (12)                        "notebook": "nbagg",
66: (8)                     }
67: (4)                 def _backend_module_name(self, backend):
68: (8)                     if backend.startswith("module://"):
69: (12)                        return backend[9:]
70: (8)                     backend = backend.lower()
71: (8)                     backend = self._name_to_module.get(backend, backend)
72: (8)                     return (backend[9:] if backend.startswith("module://")
73: (16)                            else f"matplotlib.backends.backend_{backend}")
74: (4)                 def _clear(self):
75: (8)                     self.__init__()
76: (4)                 def _ensure_entry_points_loaded(self):
77: (8)                     if not self._loaded_entry_points:
78: (12)                        entries = self._read_entry_points()
79: (12)                        self._validate_and_store_entry_points(entries)
80: (12)                        self._loaded_entry_points = True
81: (4)                 def _get_gui_framework_by_loading(self, backend):
82: (8)                     module = self.load_backend_module(backend)
83: (8)                     canvas_class = module.FigureCanvas
84: (8)                     return canvas_class.required_interactive_framework or "headless"
85: (4)                 def _read_entry_points(self):
86: (8)                     import importlib.metadata as im
87: (8)                     import sys
88: (8)                     group = "matplotlib.backend"
89: (8)                     if sys.version_info >= (3, 10):
90: (12)                        entry_points = im.entry_points(group=group)
91: (8)                     else:
92: (12)                        entry_points = im.entry_points().get(group, ())
93: (8)                     entries = [(entry.name, entry.value) for entry in entry_points]
94: (8)                     def backward_compatible_entry_points(
95: (16)                            entries, module_name, threshold_version, names, target):
96: (12)                        from matplotlib import _parse_to_version_info
97: (12)                        try:
98: (16)                            module_version = im.version(module_name)
99: (16)                            if _parse_to_version_info(module_version) < threshold_version:
100: (20)                               for name in names:
101: (24)                                   entries.append((name, target))
102: (12)                       except im.PackageNotFoundError:
103: (16)                           pass
104: (8)                    names = [entry[0] for entry in entries]
105: (8)                    if "inline" not in names:
106: (12)                       backward_compatible_entry_points(
107: (16)                           entries, "matplotlib_inline", (0, 1, 7), ["inline"],
108: (16)                           "matplotlib_inline.backend_inline")
109: (8)                    if "ipympl" not in names:
110: (12)                       backward_compatible_entry_points(
111: (16)                           entries, "ipympl", (0, 9, 4), ["ipympl", "widget"],
112: (16)                           "ipympl.backend_nbagg")
113: (8)                    return entries
114: (4)                def _validate_and_store_entry_points(self, entries):
115: (8)                    for name, module in set(entries):
116: (12)                       name = name.lower()
```

```
117: (12)                        if name.startswith("module://"):
118: (16)                            raise RuntimeError(
119: (20)                                f"Entry point name '{name}' cannot start with
'module://'")
120: (12)                        if name in self._BUILTIN_BACKEND_TO_GUI_FRAMEWORK:
121: (16)                            raise RuntimeError(f"Entry point name '{name}' is a built-in
backend")
122: (12)                        if name in self._backend_to_gui_framework:
123: (16)                            raise RuntimeError(f"Entry point name '{name}' duplicated")
124: (12)                        self._name_to_module[name] = "module://" + module
125: (12)                        self._backend_to_gui_framework[name] = "unknown"
126: (4)                def backend_for_gui_framework(self, framework):
127: (8)                    """
128: (8)                    Return the name of the backend corresponding to the specified GUI
framework.
129: (8)                    Parameters
130: (8)                    ----------
131: (8)                    framework : str
132: (12)                       GUI framework such as "qt".
133: (8)                    Returns
134: (8)                    -------
135: (8)                    str or None
136: (12)                       Backend name or None if GUI framework not recognised.
137: (8)                    """
138: (8)                    return self._GUI_FRAMEWORK_TO_BACKEND.get(framework.lower())
139: (4)                def is_valid_backend(self, backend):
140: (8)                    """
141: (8)                    Return True if the backend name is valid, False otherwise.
142: (8)                    A backend name is valid if it is one of the built-in backends or has
been
143: (8)                    dynamically added via an entry point. Those beginning with
``module://`` are
144: (8)                    always considered valid and are added to the current list of all
backends
145: (8)                    within this function.
146: (8)                    Even if a name is valid, it may not be importable or usable. This can
only be
147: (8)                    determined by loading and using the backend module.
148: (8)                    Parameters
149: (8)                    ----------
150: (8)                    backend : str
151: (12)                       Name of backend.
152: (8)                    Returns
153: (8)                    -------
154: (8)                    bool
155: (12)                       True if backend is valid, False otherwise.
156: (8)                    """
157: (8)                    if not backend.startswith("module://"):
158: (12)                       backend = backend.lower()
159: (8)                    backwards_compat = {
160: (12)                       "module://ipympl.backend_nbagg": "widget",
161: (12)                       "module://matplotlib_inline.backend_inline": "inline",
162: (8)                    }
163: (8)                    backend = backwards_compat.get(backend, backend)
164: (8)                    if (backend in self._BUILTIN_BACKEND_TO_GUI_FRAMEWORK or
165: (16)                           backend in self._backend_to_gui_framework):
166: (12)                       return True
167: (8)                    if backend.startswith("module://"):
168: (12)                       self._backend_to_gui_framework[backend] = "unknown"
169: (12)                       return True
170: (8)                    self._ensure_entry_points_loaded()
171: (8)                    if backend in self._backend_to_gui_framework:
172: (12)                       return True
173: (8)                    return False
174: (4)                def list_all(self):
175: (8)                    """
176: (8)                    Return list of all known backends.
177: (8)                    These include built-in backends and those obtained at runtime either
from entry
```

```
(8)                     points or explicit ``module://some.backend`` syntax.
(8)                     Entry points will be loaded if they haven't been already.
(8)                     Returns
(8)                     -------
(8)                     list of str
(12)                        Backend names.
(8)                     """
(8)                     self._ensure_entry_points_loaded()
(8)                     return [*self.list_builtin(), *self._backend_to_gui_framework]
(4)                 def list_builtin(self, filter_=None):
(8)                     """
(8)                     Return list of backends that are built into Matplotlib.
(8)                     Parameters
(8)                     ----------
(8)                     filter_ : `~.BackendFilter`, optional
(12)                        Filter to apply to returned backends. For example, to return only
(12)                        non-interactive backends use `.BackendFilter.NON_INTERACTIVE`.
(8)                     Returns
(8)                     -------
(8)                     list of str
(12)                        Backend names.
(8)                     """
(8)                     if filter_ == BackendFilter.INTERACTIVE:
(12)                        return [k for k, v in self._BUILTIN_BACKEND_TO_GUI_FRAMEWORK.items()
(20)                                    if v != "headless"]
(8)                     elif filter_ == BackendFilter.NON_INTERACTIVE:
(12)                        return [k for k, v in self._BUILTIN_BACKEND_TO_GUI_FRAMEWORK.items()
(20)                                    if v == "headless"]
(8)                     return [*self._BUILTIN_BACKEND_TO_GUI_FRAMEWORK]
(4)                 def list_gui_frameworks(self):
(8)                     """
(8)                     Return list of GUI frameworks used by Matplotlib backends.
(8)                     Returns
(8)                     -------
(8)                     list of str
(12)                        GUI framework names.
(8)                     """
(8)                     return [k for k in self._GUI_FRAMEWORK_TO_BACKEND if k != "headless"]
(4)                 def load_backend_module(self, backend):
(8)                     """
(8)                     Load and return the module containing the specified backend.
(8)                     Parameters
(8)                     ----------
(8)                     backend : str
(12)                        Name of backend to load.
(8)                     Returns
(8)                     -------
(8)                     Module
(12)                        Module containing backend.
(8)                     """
(8)                     module_name = self._backend_module_name(backend)
(8)                     return importlib.import_module(module_name)
(4)                 def resolve_backend(self, backend):
(8)                     """
(8)                     Return the backend and GUI framework for the specified backend name.
(8)                     If the GUI framework is not yet known then it will be determined by loading the
(8)                     backend module and checking the ``FigureCanvas.required_interactive_framework``
(8)                     attribute.
(8)                     This function only loads entry points if they have not already been loaded and
(8)                     the backend is not built-in and not of ``module://some.backend`` format.
(8)                     Parameters
(8)                     ----------
(8)                     backend : str or None
```

```
241: (12)                        Name of backend, or None to use the default backend.
242: (8)                    Returns
243: (8)                    -------
244: (8)                    backend : str
245: (12)                        The backend name.
246: (8)                    framework : str or None
247: (12)                        The GUI framework, which will be None for a backend that is non-
interactive.
248: (8)                    """
249: (8)                    if isinstance(backend, str):
250: (12)                       if not backend.startswith("module://"):
251: (16)                           backend = backend.lower()
252: (8)                    else:  # Might be _auto_backend_sentinel or None
253: (12)                       from matplotlib import get_backend
254: (12)                       backend = get_backend()
255: (8)                    gui = (self._BUILTIN_BACKEND_TO_GUI_FRAMEWORK.get(backend) or
256: (15)                          self._backend_to_gui_framework.get(backend))
257: (8)                    if gui is None and isinstance(backend, str) and
backend.startswith("module://"):
258: (12)                       gui = "unknown"
259: (8)                    if gui is None and not self._loaded_entry_points:
260: (12)                       self._ensure_entry_points_loaded()
261: (12)                       gui = self._backend_to_gui_framework.get(backend)
262: (8)                    if gui == "unknown":
263: (12)                       gui = self._get_gui_framework_by_loading(backend)
264: (12)                       self._backend_to_gui_framework[backend] = gui
265: (8)                    if gui is None:
266: (12)                       raise RuntimeError(f"'{backend}' is not a recognised backend
name")
267: (8)                    return backend, gui if gui != "headless" else None
268: (4)                def resolve_gui_or_backend(self, gui_or_backend):
269: (8)                    """
270: (8)                    Return the backend and GUI framework for the specified string that may
be
271: (8)                    either a GUI framework or a backend name, tested in that order.
272: (8)                    This is for use with the IPython %matplotlib magic command which may
be a GUI
273: (8)                    framework such as ``%matplotlib qt`` or a backend name such as
274: (8)                    ``%matplotlib qtagg``.
275: (8)                    This function only loads entry points if they have not already been
loaded and
276: (8)                    the backend is not built-in and not of ``module://some.backend``
format.
277: (8)                    Parameters
278: (8)                    ----------
279: (8)                    gui_or_backend : str or None
280: (12)                       Name of GUI framework or backend, or None to use the default
backend.
281: (8)                    Returns
282: (8)                    -------
283: (8)                    backend : str
284: (12)                       The backend name.
285: (8)                    framework : str or None
286: (12)                       The GUI framework, which will be None for a backend that is non-
interactive.
287: (8)                    """
288: (8)                    if not gui_or_backend.startswith("module://"):
289: (12)                       gui_or_backend = gui_or_backend.lower()
290: (8)                    backend = self.backend_for_gui_framework(gui_or_backend)
291: (8)                    if backend is not None:
292: (12)                       return backend, gui_or_backend if gui_or_backend != "headless"
else None
293: (8)                    try:
294: (12)                       return self.resolve_backend(gui_or_backend)
295: (8)                    except Exception:  # KeyError ?
296: (12)                       raise RuntimeError(
297: (16)                           f"'{gui_or_backend} is not a recognised GUI loop or backend
name")
298: (0)            backend_registry = BackendRegistry()
```

```
----------------------------------------

File 113 - __init__.py:

1: (0)

----------------------------------------

File 114 - _formlayout.py:

1: (0)              """
2: (0)              formlayout
3: (0)              ==========
4: (0)              Module creating Qt form dialogs/layouts to edit various type of parameters
5: (0)              formlayout License Agreement (MIT License)
6: (0)              ----------------------------------------
7: (0)              Copyright (c) 2009 Pierre Raybaut
8: (0)              Permission is hereby granted, free of charge, to any person
9: (0)              obtaining a copy of this software and associated documentation
10: (0)             files (the "Software"), to deal in the Software without
11: (0)             restriction, including without limitation the rights to use,
12: (0)             copy, modify, merge, publish, distribute, sublicense, and/or sell
13: (0)             copies of the Software, and to permit persons to whom the
14: (0)             Software is furnished to do so, subject to the following
15: (0)             conditions:
16: (0)             The above copyright notice and this permission notice shall be
17: (0)             included in all copies or substantial portions of the Software.
18: (0)             THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
19: (0)             EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
20: (0)             OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
21: (0)             NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
22: (0)             HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
23: (0)             WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
24: (0)             FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
25: (0)             OTHER DEALINGS IN THE SOFTWARE.
26: (0)             """
27: (0)             __version__ = '1.0.10'
28: (0)             __license__ = __doc__
29: (0)             from ast import literal_eval
30: (0)             import copy
31: (0)             import datetime
32: (0)             import logging
33: (0)             from numbers import Integral, Real
34: (0)             from matplotlib import _api, colors as mcolors
35: (0)             from matplotlib.backends.qt_compat import _to_int, QtGui, QtWidgets, QtCore
36: (0)             _log = logging.getLogger(__name__)
37: (0)             BLACKLIST = {"title", "label"}
38: (0)             class ColorButton(QtWidgets.QPushButton):
39: (4)                 """
40: (4)                 Color choosing push button
41: (4)                 """
42: (4)                 colorChanged = QtCore.Signal(QtGui.QColor)
43: (4)                 def __init__(self, parent=None):
44: (8)                     super().__init__(parent)
45: (8)                     self.setFixedSize(20, 20)
46: (8)                     self.setIconSize(QtCore.QSize(12, 12))
47: (8)                     self.clicked.connect(self.choose_color)
48: (8)                     self._color = QtGui.QColor()
49: (4)                 def choose_color(self):
50: (8)                     color = QtWidgets.QColorDialog.getColor(
51: (12)                        self._color, self.parentWidget(), "",
52: (12)                        QtWidgets.QColorDialog.ColorDialogOption.ShowAlphaChannel)
53: (8)                     if color.isValid():
54: (12)                        self.set_color(color)
55: (4)                 def get_color(self):
56: (8)                     return self._color
57: (4)                 @QtCore.Slot(QtGui.QColor)
58: (4)                 def set_color(self, color):
```

```
59: (8)                     if color != self._color:
60: (12)                        self._color = color
61: (12)                        self.colorChanged.emit(self._color)
62: (12)                        pixmap = QtGui.QPixmap(self.iconSize())
63: (12)                        pixmap.fill(color)
64: (12)                        self.setIcon(QtGui.QIcon(pixmap))
65: (4)                 color = QtCore.Property(QtGui.QColor, get_color, set_color)
66: (0)             def to_qcolor(color):
67: (4)                 """Create a QColor from a matplotlib color"""
68: (4)                 qcolor = QtGui.QColor()
69: (4)                 try:
70: (8)                     rgba = mcolors.to_rgba(color)
71: (4)                 except ValueError:
72: (8)                     _api.warn_external(f'Ignoring invalid color {color!r}')
73: (8)                     return qcolor  # return invalid QColor
74: (4)                 qcolor.setRgbF(*rgba)
75: (4)                 return qcolor
76: (0)             class ColorLayout(QtWidgets.QHBoxLayout):
77: (4)                 """Color-specialized QLineEdit layout"""
78: (4)                 def __init__(self, color, parent=None):
79: (8)                     super().__init__()
80: (8)                     assert isinstance(color, QtGui.QColor)
81: (8)                     self.lineedit = QtWidgets.QLineEdit(
82: (12)                        mcolors.to_hex(color.getRgbF(), keep_alpha=True), parent)
83: (8)                     self.lineedit.editingFinished.connect(self.update_color)
84: (8)                     self.addWidget(self.lineedit)
85: (8)                     self.colorbtn = ColorButton(parent)
86: (8)                     self.colorbtn.color = color
87: (8)                     self.colorbtn.colorChanged.connect(self.update_text)
88: (8)                     self.addWidget(self.colorbtn)
89: (4)                 def update_color(self):
90: (8)                     color = self.text()
91: (8)                     qcolor = to_qcolor(color)  # defaults to black if not qcolor.isValid()
92: (8)                     self.colorbtn.color = qcolor
93: (4)                 def update_text(self, color):
94: (8)                     self.lineedit.setText(mcolors.to_hex(color.getRgbF(),
keep_alpha=True))
95: (4)                 def text(self):
96: (8)                     return self.lineedit.text()
97: (0)             def font_is_installed(font):
98: (4)                 """Check if font is installed"""
99: (4)                 return [fam for fam in QtGui.QFontDatabase().families()
100: (12)                       if str(fam) == font]
101: (0)            def tuple_to_qfont(tup):
102: (4)                """
103: (4)                Create a QFont from tuple:
104: (8)                    (family [string], size [int], italic [bool], bold [bool])
105: (4)                """
106: (4)                if not (isinstance(tup, tuple) and len(tup) == 4
107: (12)                       and font_is_installed(tup[0])
108: (12)                       and isinstance(tup[1], Integral)
109: (12)                       and isinstance(tup[2], bool)
110: (12)                       and isinstance(tup[3], bool)):
111: (8)                    return None
112: (4)                font = QtGui.QFont()
113: (4)                family, size, italic, bold = tup
114: (4)                font.setFamily(family)
115: (4)                font.setPointSize(size)
116: (4)                font.setItalic(italic)
117: (4)                font.setBold(bold)
118: (4)                return font
119: (0)            def qfont_to_tuple(font):
120: (4)                return (str(font.family()), int(font.pointSize()),
121: (12)                       font.italic(), font.bold())
122: (0)            class FontLayout(QtWidgets.QGridLayout):
123: (4)                """Font selection"""
124: (4)                def __init__(self, value, parent=None):
125: (8)                    super().__init__()
126: (8)                    font = tuple_to_qfont(value)
```

```
(8)                     assert font is not None
(8)                     self.family = QtWidgets.QFontComboBox(parent)
(8)                     self.family.setCurrentFont(font)
(8)                     self.addWidget(self.family, 0, 0, 1, -1)
(8)                     self.size = QtWidgets.QComboBox(parent)
(8)                     self.size.setEditable(True)
(8)                     sizelist = [*range(6, 12), *range(12, 30, 2), 36, 48, 72]
(8)                     size = font.pointSize()
(8)                     if size not in sizelist:
(12)                        sizelist.append(size)
(12)                        sizelist.sort()
(8)                     self.size.addItems([str(s) for s in sizelist])
(8)                     self.size.setCurrentIndex(sizelist.index(size))
(8)                     self.addWidget(self.size, 1, 0)
(8)                     self.italic = QtWidgets.QCheckBox(self.tr("Italic"), parent)
(8)                     self.italic.setChecked(font.italic())
(8)                     self.addWidget(self.italic, 1, 1)
(8)                     self.bold = QtWidgets.QCheckBox(self.tr("Bold"), parent)
(8)                     self.bold.setChecked(font.bold())
(8)                     self.addWidget(self.bold, 1, 2)
(4)                 def get_font(self):
(8)                     font = self.family.currentFont()
(8)                     font.setItalic(self.italic.isChecked())
(8)                     font.setBold(self.bold.isChecked())
(8)                     font.setPointSize(int(self.size.currentText()))
(8)                     return qfont_to_tuple(font)
(0)             def is_edit_valid(edit):
(4)                 text = edit.text()
(4)                 state = edit.validator().validate(text, 0)[0]
(4)                 return state == QtGui.QDoubleValidator.State.Acceptable
(0)             class FormWidget(QtWidgets.QWidget):
(4)                 update_buttons = QtCore.Signal()
(4)                 def __init__(self, data, comment="", with_margin=False, parent=None):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     data : list of (label, value) pairs
(12)                        The data to be edited in the form.
(8)                     comment : str, optional
(8)                     with_margin : bool, default: False
(12)                        If False, the form elements reach to the border of the widget.
(12)                        This is the desired behavior if the FormWidget is used as a widget
(12)                        alongside with other widgets such as a QComboBox, which also do
(12)                        not have a margin around them.
(12)                        However, a margin can be desired if the FormWidget is the only
(12)                        widget within a container, e.g. a tab in a QTabWidget.
(8)                     parent : QWidget or None
(12)                        The parent widget.
(8)                     """
(8)                     super().__init__(parent)
(8)                     self.data = copy.deepcopy(data)
(8)                     self.widgets = []
(8)                     self.formlayout = QtWidgets.QFormLayout(self)
(8)                     if not with_margin:
(12)                        self.formlayout.setContentsMargins(0, 0, 0, 0)
(8)                     if comment:
(12)                        self.formlayout.addRow(QtWidgets.QLabel(comment))
(12)                        self.formlayout.addRow(QtWidgets.QLabel(" "))
(4)                 def get_dialog(self):
(8)                     """Return FormDialog instance"""
(8)                     dialog = self.parent()
(8)                     while not isinstance(dialog, QtWidgets.QDialog):
(12)                        dialog = dialog.parent()
(8)                     return dialog
(4)                 def setup(self):
(8)                     for label, value in self.data:
(12)                        if label is None and value is None:
(16)                            self.formlayout.addRow(QtWidgets.QLabel(" "),
(39)                                                   QtWidgets.QLabel(" "))
```

```
196: (16)                            self.widgets.append(None)
197: (16)                            continue
198: (12)                        elif label is None:
199: (16)                            self.formlayout.addRow(QtWidgets.QLabel(value))
200: (16)                            self.widgets.append(None)
201: (16)                            continue
202: (12)                        elif tuple_to_qfont(value) is not None:
203: (16)                            field = FontLayout(value, self)
204: (12)                        elif (label.lower() not in BLACKLIST
205: (18)                              and mcolors.is_color_like(value)):
206: (16)                            field = ColorLayout(to_qcolor(value), self)
207: (12)                        elif isinstance(value, str):
208: (16)                            field = QtWidgets.QLineEdit(value, self)
209: (12)                        elif isinstance(value, (list, tuple)):
210: (16)                            if isinstance(value, tuple):
211: (20)                                value = list(value)
212: (16)                            selindex = value.pop(0)
213: (16)                            field = QtWidgets.QComboBox(self)
214: (16)                            if isinstance(value[0], (list, tuple)):
215: (20)                                keys = [key for key, _val in value]
216: (20)                                value = [val for _key, val in value]
217: (16)                            else:
218: (20)                                keys = value
219: (16)                            field.addItems(value)
220: (16)                            if selindex in value:
221: (20)                                selindex = value.index(selindex)
222: (16)                            elif selindex in keys:
223: (20)                                selindex = keys.index(selindex)
224: (16)                            elif not isinstance(selindex, Integral):
225: (20)                                _log.warning(
226: (24)                                    "index '%s' is invalid (label: %s, value: %s)",
227: (24)                                    selindex, label, value)
228: (20)                                selindex = 0
229: (16)                            field.setCurrentIndex(selindex)
230: (12)                        elif isinstance(value, bool):
231: (16)                            field = QtWidgets.QCheckBox(self)
232: (16)                            field.setChecked(value)
233: (12)                        elif isinstance(value, Integral):
234: (16)                            field = QtWidgets.QSpinBox(self)
235: (16)                            field.setRange(-10**9, 10**9)
236: (16)                            field.setValue(value)
237: (12)                        elif isinstance(value, Real):
238: (16)                            field = QtWidgets.QLineEdit(repr(value), self)
239: (16)                            field.setCursorPosition(0)
240: (16)                            field.setValidator(QtGui.QDoubleValidator(field))
241: (16)                            field.validator().setLocale(QtCore.QLocale("C"))
242: (16)                            dialog = self.get_dialog()
243: (16)                            dialog.register_float_field(field)
244: (16)                            field.textChanged.connect(lambda text:
dialog.update_buttons())
245: (12)                        elif isinstance(value, datetime.datetime):
246: (16)                            field = QtWidgets.QDateTimeEdit(self)
247: (16)                            field.setDateTime(value)
248: (12)                        elif isinstance(value, datetime.date):
249: (16)                            field = QtWidgets.QDateEdit(self)
250: (16)                            field.setDate(value)
251: (12)                        else:
252: (16)                            field = QtWidgets.QLineEdit(repr(value), self)
253: (12)                        self.formlayout.addRow(label, field)
254: (12)                        self.widgets.append(field)
255: (4)                 def get(self):
256: (8)                     valuelist = []
257: (8)                     for index, (label, value) in enumerate(self.data):
258: (12)                        field = self.widgets[index]
259: (12)                        if label is None:
260: (16)                            continue
261: (12)                        elif tuple_to_qfont(value) is not None:
262: (16)                            value = field.get_font()
263: (12)                        elif isinstance(value, str) or mcolors.is_color_like(value):
```

```
(16)                            value = str(field.text())
(12)                        elif isinstance(value, (list, tuple)):
(16)                            index = int(field.currentIndex())
(16)                            if isinstance(value[0], (list, tuple)):
(20)                                value = value[index][0]
(16)                            else:
(20)                                value = value[index]
(12)                        elif isinstance(value, bool):
(16)                            value = field.isChecked()
(12)                        elif isinstance(value, Integral):
(16)                            value = int(field.value())
(12)                        elif isinstance(value, Real):
(16)                            value = float(str(field.text()))
(12)                        elif isinstance(value, datetime.datetime):
(16)                            datetime_ = field.dateTime()
(16)                            if hasattr(datetime_, "toPyDateTime"):
(20)                                value = datetime_.toPyDateTime()
(16)                            else:
(20)                                value = datetime_.toPython()
(12)                        elif isinstance(value, datetime.date):
(16)                            date_ = field.date()
(16)                            if hasattr(date_, "toPyDate"):
(20)                                value = date_.toPyDate()
(16)                            else:
(20)                                value = date_.toPython()
(12)                        else:
(16)                            value = literal_eval(str(field.text()))
(12)                        valuelist.append(value)
(8)                     return valuelist
(0)             class FormComboWidget(QtWidgets.QWidget):
(4)                 update_buttons = QtCore.Signal()
(4)                 def __init__(self, datalist, comment="", parent=None):
(8)                     super().__init__(parent)
(8)                     layout = QtWidgets.QVBoxLayout()
(8)                     self.setLayout(layout)
(8)                     self.combobox = QtWidgets.QComboBox()
(8)                     layout.addWidget(self.combobox)
(8)                     self.stackwidget = QtWidgets.QStackedWidget(self)
(8)                     layout.addWidget(self.stackwidget)
(8)                     self.combobox.currentIndexChanged.connect(
(12)                        self.stackwidget.setCurrentIndex)
(8)                     self.widgetlist = []
(8)                     for data, title, comment in datalist:
(12)                        self.combobox.addItem(title)
(12)                        widget = FormWidget(data, comment=comment, parent=self)
(12)                        self.stackwidget.addWidget(widget)
(12)                        self.widgetlist.append(widget)
(4)                 def setup(self):
(8)                     for widget in self.widgetlist:
(12)                        widget.setup()
(4)                 def get(self):
(8)                     return [widget.get() for widget in self.widgetlist]
(0)             class FormTabWidget(QtWidgets.QWidget):
(4)                 update_buttons = QtCore.Signal()
(4)                 def __init__(self, datalist, comment="", parent=None):
(8)                     super().__init__(parent)
(8)                     layout = QtWidgets.QVBoxLayout()
(8)                     self.tabwidget = QtWidgets.QTabWidget()
(8)                     layout.addWidget(self.tabwidget)
(8)                     layout.setContentsMargins(0, 0, 0, 0)
(8)                     self.setLayout(layout)
(8)                     self.widgetlist = []
(8)                     for data, title, comment in datalist:
(12)                        if len(data[0]) == 3:
(16)                            widget = FormComboWidget(data, comment=comment, parent=self)
(12)                        else:
(16)                            widget = FormWidget(data, with_margin=True, comment=comment,
(36)                                                parent=self)
(12)                        index = self.tabwidget.addTab(widget, title)
```

```
(12)                        self.tabwidget.setTabToolTip(index, comment)
(12)                        self.widgetlist.append(widget)
(4)                 def setup(self):
(8)                     for widget in self.widgetlist:
(12)                        widget.setup()
(4)                 def get(self):
(8)                     return [widget.get() for widget in self.widgetlist]
(0)             class FormDialog(QtWidgets.QDialog):
(4)                 """Form Dialog"""
(4)                 def __init__(self, data, title="", comment="",
(17)                            icon=None, parent=None, apply=None):
(8)                     super().__init__(parent)
(8)                     self.apply_callback = apply
(8)                     if isinstance(data[0][0], (list, tuple)):
(12)                        self.formwidget = FormTabWidget(data, comment=comment,
(44)                                                        parent=self)
(8)                     elif len(data[0]) == 3:
(12)                        self.formwidget = FormComboWidget(data, comment=comment,
(46)                                                          parent=self)
(8)                     else:
(12)                        self.formwidget = FormWidget(data, comment=comment,
(41)                                                     parent=self)
(8)                     layout = QtWidgets.QVBoxLayout()
(8)                     layout.addWidget(self.formwidget)
(8)                     self.float_fields = []
(8)                     self.formwidget.setup()
(8)                     self.bbox = bbox = QtWidgets.QDialogButtonBox(
(12)                        QtWidgets.QDialogButtonBox.StandardButton(
(20)                                _to_int(QtWidgets.QDialogButtonBox.StandardButton.Ok) |
(20)                                _to_int(QtWidgets.QDialogButtonBox.StandardButton.Cancel)
(12)                        ))
(8)                     self.formwidget.update_buttons.connect(self.update_buttons)
(8)                     if self.apply_callback is not None:
(12)                        apply_btn = bbox.addButton(
(16)                            QtWidgets.QDialogButtonBox.StandardButton.Apply)
(12)                        apply_btn.clicked.connect(self.apply)
(8)                     bbox.accepted.connect(self.accept)
(8)                     bbox.rejected.connect(self.reject)
(8)                     layout.addWidget(bbox)
(8)                     self.setLayout(layout)
(8)                     self.setWindowTitle(title)
(8)                     if not isinstance(icon, QtGui.QIcon):
(12)                        icon = QtWidgets.QWidget().style().standardIcon(
(16)                            QtWidgets.QStyle.SP_MessageBoxQuestion)
(8)                     self.setWindowIcon(icon)
(4)                 def register_float_field(self, field):
(8)                     self.float_fields.append(field)
(4)                 def update_buttons(self):
(8)                     valid = True
(8)                     for field in self.float_fields:
(12)                        if not is_edit_valid(field):
(16)                            valid = False
(8)                     for btn_type in ["Ok", "Apply"]:
(12)                        btn = self.bbox.button(
(16)                            getattr(QtWidgets.QDialogButtonBox.StandardButton,
(24)                                    btn_type))
(12)                        if btn is not None:
(16)                            btn.setEnabled(valid)
(4)                 def accept(self):
(8)                     self.data = self.formwidget.get()
(8)                     self.apply_callback(self.data)
(8)                     super().accept()
(4)                 def reject(self):
(8)                     self.data = None
(8)                     super().reject()
(4)                 def apply(self):
(8)                     self.apply_callback(self.formwidget.get())
(4)                 def get(self):
(8)                     """Return form result"""
```

```
402: (8)                    return self.data
403: (0)            def fedit(data, title="", comment="", icon=None, parent=None, apply=None):
404: (4)                """
405: (4)                Create form dialog
406: (4)                data: datalist, datagroup
407: (4)                title: str
408: (4)                comment: str
409: (4)                icon: QIcon instance
410: (4)                parent: parent QWidget
411: (4)                apply: apply callback (function)
412: (4)                datalist: list/tuple of (field_name, field_value)
413: (4)                datagroup: list/tuple of (datalist *or* datagroup, title, comment)
414: (4)                -> one field for each member of a datalist
415: (4)                -> one tab for each member of a top-level datagroup
416: (4)                -> one page (of a multipage widget, each page can be selected with a combo
417: (7)                   box) for each member of a datagroup inside a datagroup
418: (4)                Supported types for field_value:
419: (6)                  - int, float, str, bool
420: (6)                  - colors: in Qt-compatible text form, i.e. in hex format or name
421: (16)                           (red, ...) (automatically detected from a string)
422: (6)                  - list/tuple:
423: (10)                     * the first element will be the selected index (or value)
424: (10)                     * the other elements can be couples (key, value) or only values
425: (4)                """
426: (4)                if QtWidgets.QApplication.startingUp():
427: (8)                    _app = QtWidgets.QApplication([])
428: (4)                dialog = FormDialog(data, title, comment, icon, parent, apply)
429: (4)                if parent is not None:
430: (8)                    if hasattr(parent, "_fedit_dialog"):
431: (12)                       parent._fedit_dialog.close()
432: (8)                    parent._fedit_dialog = dialog
433: (4)                dialog.show()
434: (0)            if __name__ == "__main__":
435: (4)                _app = QtWidgets.QApplication([])
436: (4)                def create_datalist_example():
437: (8)                    return [('str', 'this is a string'),
438: (16)                           ('list', [0, '1', '3', '4']),
439: (16)                           ('list2', ['--', ('none', 'None'), ('--', 'Dashed'),
440: (27)                                      ('-.', 'DashDot'), ('-', 'Solid'),
441: (27)                                      ('steps', 'Steps'), (':', 'Dotted')]),
442: (16)                           ('float', 1.2),
443: (16)                           (None, 'Other:'),
444: (16)                           ('int', 12),
445: (16)                           ('font', ('Arial', 10, False, True)),
446: (16)                           ('color', '#123409'),
447: (16)                           ('bool', True),
448: (16)                           ('date', datetime.date(2010, 10, 10)),
449: (16)                           ('datetime', datetime.datetime(2010, 10, 10)),
450: (16)                           ]
451: (4)                def create_datagroup_example():
452: (8)                    datalist = create_datalist_example()
453: (8)                    return ((datalist, "Category 1", "Category 1 comment"),
454: (16)                           (datalist, "Category 2", "Category 2 comment"),
455: (16)                           (datalist, "Category 3", "Category 3 comment"))
456: (4)                datalist = create_datalist_example()
457: (4)                def apply_test(data):
458: (8)                    print("data:", data)
459: (4)                fedit(datalist, title="Example",
460: (10)                     comment="This is just an <b>example</b>.",
461: (10)                     apply=apply_test)
462: (4)                _app.exec()
463: (4)                datagroup = create_datagroup_example()
464: (4)                fedit(datagroup, "Global title",
465: (10)                     apply=apply_test)
466: (4)                _app.exec()
467: (4)                datalist = create_datalist_example()
468: (4)                datagroup = create_datagroup_example()
469: (4)                fedit(((datagroup, "Title 1", "Tab 1 comment"),
470: (11)                      (datalist, "Title 2", "Tab 2 comment"),
```

```
471: (11)                     (datalist, "Title 3", "Tab 3 comment")),
472: (10)                    "Global title",
473: (10)                    apply=apply_test)
474: (4)                _app.exec()
```

File 115 - figureoptions.py:

```
1: (0)              """Module that provides a GUI-based editor for Matplotlib's figure options."""
2: (0)              from itertools import chain
3: (0)              from matplotlib import cbook, cm, colors as mcolors, markers, image as mimage
4: (0)              from matplotlib.backends.qt_compat import QtGui
5: (0)              from matplotlib.backends.qt_editor import _formlayout
6: (0)              from matplotlib.dates import DateConverter, num2date
7: (0)              LINESTYLES = {'-': 'Solid',
8: (14)                           '--': 'Dashed',
9: (14)                           '-.': 'DashDot',
10: (14)                          ':': 'Dotted',
11: (14)                          'None': 'None',
12: (14)                          }
13: (0)             DRAWSTYLES = {
14: (4)                 'default': 'Default',
15: (4)                 'steps-pre': 'Steps (Pre)', 'steps': 'Steps (Pre)',
16: (4)                 'steps-mid': 'Steps (Mid)',
17: (4)                 'steps-post': 'Steps (Post)'}
18: (0)             MARKERS = markers.MarkerStyle.markers
19: (0)             def figure_edit(axes, parent=None):
20: (4)                 """Edit matplotlib figure options"""
21: (4)                 sep = (None, None)  # separator
22: (4)                 def convert_limits(lim, converter):
23: (8)                     """Convert axis limits for correct input editors."""
24: (8)                     if isinstance(converter, DateConverter):
25: (12)                        return map(num2date, lim)
26: (8)                     return map(float, lim)
27: (4)                 axis_map = axes._axis_map
28: (4)                 axis_limits = {
29: (8)                     name: tuple(convert_limits(
30: (12)                        getattr(axes, f'get_{name}lim')(), axis.converter
31: (8)                     ))
32: (8)                     for name, axis in axis_map.items()
33: (4)                 }
34: (4)                 general = [
35: (8)                     ('Title', axes.get_title()),
36: (8)                     sep,
37: (8)                     *chain.from_iterable([
38: (12)                        (
39: (16)                            (None, f"<b>{name.title()}-Axis</b>"),
40: (16)                            ('Min', axis_limits[name][0]),
41: (16)                            ('Max', axis_limits[name][1]),
42: (16)                            ('Label', axis.get_label().get_text()),
43: (16)                            ('Scale', [axis.get_scale(),
44: (27)                                       'linear', 'log', 'symlog', 'logit']),
45: (16)                            sep,
46: (12)                        )
47: (12)                        for name, axis in axis_map.items()
48: (8)                     ]),
49: (8)                     ('(Re-)Generate automatic legend', False),
50: (4)                 ]
51: (4)                 axis_converter = {
52: (8)                     name: axis.converter
53: (8)                     for name, axis in axis_map.items()
54: (4)                 }
55: (4)                 axis_units = {
56: (8)                     name: axis.get_units()
57: (8)                     for name, axis in axis_map.items()
58: (4)                 }
59: (4)                 labeled_lines = []
60: (4)                 for line in axes.get_lines():
```

```
 61: (8)                     label = line.get_label()
 62: (8)                     if label == '_nolegend_':
 63: (12)                        continue
 64: (8)                     labeled_lines.append((label, line))
 65: (4)                 curves = []
 66: (4)                 def prepare_data(d, init):
 67: (8)                     """
 68: (8)                     Prepare entry for FormLayout.
 69: (8)                     *d* is a mapping of shorthands to style names (a single style may
 70: (8)                     have multiple shorthands, in particular the shorthands `None`,
 71: (8)                     `"None"`, `"none"` and `""` are synonyms); *init* is one shorthand
 72: (8)                     of the initial style.
 73: (8)                     This function returns an list suitable for initializing a
 74: (8)                     FormLayout combobox, namely `[initial_name, (shorthand,
 75: (8)                     style_name), (shorthand, style_name), ...]`.
 76: (8)                     """
 77: (8)                     if init not in d:
 78: (12)                        d = {**d, init: str(init)}
 79: (8)                     name2short = {name: short for short, name in d.items()}
 80: (8)                     short2name = {short: name for name, short in name2short.items()}
 81: (8)                     canonical_init = name2short[d[init]]
 82: (8)                     return ([canonical_init] +
 83: (16)                            sorted(short2name.items(),
 84: (23)                                   key=lambda short_and_name: short_and_name[1]))
 85: (4)                 for label, line in labeled_lines:
 86: (8)                     color = mcolors.to_hex(
 87: (12)                        mcolors.to_rgba(line.get_color(), line.get_alpha()),
 88: (12)                        keep_alpha=True)
 89: (8)                     ec = mcolors.to_hex(
 90: (12)                        mcolors.to_rgba(line.get_markeredgecolor(), line.get_alpha()),
 91: (12)                        keep_alpha=True)
 92: (8)                     fc = mcolors.to_hex(
 93: (12)                        mcolors.to_rgba(line.get_markerfacecolor(), line.get_alpha()),
 94: (12)                        keep_alpha=True)
 95: (8)                     curvedata = [
 96: (12)                        ('Label', label),
 97: (12)                        sep,
 98: (12)                        (None, '<b>Line</b>'),
 99: (12)                        ('Line style', prepare_data(LINESTYLES, line.get_linestyle())),
100: (12)                        ('Draw style', prepare_data(DRAWSTYLES, line.get_drawstyle())),
101: (12)                        ('Width', line.get_linewidth()),
102: (12)                        ('Color (RGBA)', color),
103: (12)                        sep,
104: (12)                        (None, '<b>Marker</b>'),
105: (12)                        ('Style', prepare_data(MARKERS, line.get_marker())),
106: (12)                        ('Size', line.get_markersize()),
107: (12)                        ('Face color (RGBA)', fc),
108: (12)                        ('Edge color (RGBA)', ec)]
109: (8)                     curves.append([curvedata, label, ""])
110: (4)                 has_curve = bool(curves)
111: (4)                 labeled_mappables = []
112: (4)                 for mappable in [*axes.images, *axes.collections]:
113: (8)                     label = mappable.get_label()
114: (8)                     if label == '_nolegend_' or mappable.get_array() is None:
115: (12)                        continue
116: (8)                     labeled_mappables.append((label, mappable))
117: (4)                 mappables = []
118: (4)                 cmaps = [(cmap, name) for name, cmap in sorted(cm._colormaps.items())]
119: (4)                 for label, mappable in labeled_mappables:
120: (8)                     cmap = mappable.get_cmap()
121: (8)                     if cmap not in cm._colormaps.values():
122: (12)                        cmaps = [(cmap, cmap.name), *cmaps]
123: (8)                     low, high = mappable.get_clim()
124: (8)                     mappabledata = [
125: (12)                        ('Label', label),
126: (12)                        ('Colormap', [cmap.name] + cmaps),
127: (12)                        ('Min. value', low),
128: (12)                        ('Max. value', high),
129: (8)                     ]
```

```
130: (8)                    if hasattr(mappable, "get_interpolation"):  # Images.
131: (12)                       interpolations = [
132: (16)                           (name, name) for name in sorted(mimage.interpolations_names)]
133: (12)                       mappabledata.append((
134: (16)                           'Interpolation',
135: (16)                           [mappable.get_interpolation(), *interpolations]))
136: (12)                       interpolation_stages = ['data', 'rgba']
137: (12)                       mappabledata.append((
138: (16)                           'Interpolation stage',
139: (16)                           [mappable.get_interpolation_stage(), *interpolation_stages]))
140: (8)                    mappables.append([mappabledata, label, ""])
141: (4)                has_sm = bool(mappables)
142: (4)                datalist = [(general, "Axes", "")]
143: (4)                if curves:
144: (8)                    datalist.append((curves, "Curves", ""))
145: (4)                if mappables:
146: (8)                    datalist.append((mappables, "Images, etc.", ""))
147: (4)                def apply_callback(data):
148: (8)                    """A callback to apply changes."""
149: (8)                    orig_limits = {
150: (12)                       name: getattr(axes, f"get_{name}lim")()
151: (12)                       for name in axis_map
152: (8)                    }
153: (8)                    general = data.pop(0)
154: (8)                    curves = data.pop(0) if has_curve else []
155: (8)                    mappables = data.pop(0) if has_sm else []
156: (8)                    if data:
157: (12)                       raise ValueError("Unexpected field")
158: (8)                    title = general.pop(0)
159: (8)                    axes.set_title(title)
160: (8)                    generate_legend = general.pop()
161: (8)                    for i, (name, axis) in enumerate(axis_map.items()):
162: (12)                       axis_min = general[4*i]
163: (12)                       axis_max = general[4*i + 1]
164: (12)                       axis_label = general[4*i + 2]
165: (12)                       axis_scale = general[4*i + 3]
166: (12)                       if axis.get_scale() != axis_scale:
167: (16)                           getattr(axes, f"set_{name}scale")(axis_scale)
168: (12)                       axis._set_lim(axis_min, axis_max, auto=False)
169: (12)                       axis.set_label_text(axis_label)
170: (12)                       axis.converter = axis_converter[name]
171: (12)                       axis.set_units(axis_units[name])
172: (8)                    for index, curve in enumerate(curves):
173: (12)                       line = labeled_lines[index][1]
174: (12)                       (label, linestyle, drawstyle, linewidth, color, marker,
markersize,
175: (13)                        markerfacecolor, markeredgecolor) = curve
176: (12)                       line.set_label(label)
177: (12)                       line.set_linestyle(linestyle)
178: (12)                       line.set_drawstyle(drawstyle)
179: (12)                       line.set_linewidth(linewidth)
180: (12)                       rgba = mcolors.to_rgba(color)
181: (12)                       line.set_alpha(None)
182: (12)                       line.set_color(rgba)
183: (12)                       if marker != 'none':
184: (16)                           line.set_marker(marker)
185: (16)                           line.set_markersize(markersize)
186: (16)                           line.set_markerfacecolor(markerfacecolor)
187: (16)                           line.set_markeredgecolor(markeredgecolor)
188: (8)                    for index, mappable_settings in enumerate(mappables):
189: (12)                       mappable = labeled_mappables[index][1]
190: (12)                       if len(mappable_settings) == 6:
191: (16)                           label, cmap, low, high, interpolation, interpolation_stage = \
192: (18)                             mappable_settings
193: (16)                           mappable.set_interpolation(interpolation)
194: (16)                           mappable.set_interpolation_stage(interpolation_stage)
195: (12)                       elif len(mappable_settings) == 4:
196: (16)                           label, cmap, low, high = mappable_settings
197: (12)                       mappable.set_label(label)
```

```
198: (12)                        mappable.set_cmap(cmap)
199: (12)                        mappable.set_clim(*sorted([low, high]))
200: (8)                    if generate_legend:
201: (12)                        draggable = None
202: (12)                        ncols = 1
203: (12)                        if axes.legend_ is not None:
204: (16)                            old_legend = axes.get_legend()
205: (16)                            draggable = old_legend._draggable is not None
206: (16)                            ncols = old_legend._ncols
207: (12)                        new_legend = axes.legend(ncols=ncols)
208: (12)                        if new_legend:
209: (16)                            new_legend.set_draggable(draggable)
210: (8)                    figure = axes.get_figure()
211: (8)                    figure.canvas.draw()
212: (8)                    for name in axis_map:
213: (12)                        if getattr(axes, f"get_{name}lim")() != orig_limits[name]:
214: (16)                            figure.canvas.toolbar.push_current()
215: (16)                            break
216: (4)                _formlayout.fedit(
217: (8)                    datalist, title="Figure options", parent=parent,
218: (8)                    icon=QtGui.QIcon(
219: (12)                        str(cbook._get_data_path('images', 'qt4_editor_options.svg'))),
220: (8)                    apply=apply_callback)
```

File 116 - __init__.py:

```
1: (0)              """
2: (0)              Non-separable transforms that map from data space to screen space.
3: (0)              Projections are defined as `~.axes.Axes` subclasses.  They include the
4: (0)              following elements:
5: (0)              - A transformation from data coordinates into display coordinates.
6: (0)              - An inverse of that transformation.  This is used, for example, to convert
7: (2)                mouse positions from screen space back into data space.
8: (0)              - Transformations for the gridlines, ticks and ticklabels.  Custom projections
9: (2)                will often need to place these elements in special locations, and Matplotlib
10: (2)               has a facility to help with doing so.
11: (0)             - Setting up default values (overriding `~.axes.Axes.cla`), since the defaults
12: (2)               for a rectilinear Axes may not be appropriate.
13: (0)             - Defining the shape of the Axes, for example, an elliptical Axes, that will
be
14: (2)               used to draw the background of the plot and for clipping any data elements.
15: (0)             - Defining custom locators and formatters for the projection.  For example, in
16: (2)               a geographic projection, it may be more convenient to display the grid in
17: (2)               degrees, even if the data is in radians.
18: (0)             - Set up interactive panning and zooming.  This is left as an "advanced"
19: (2)               feature left to the reader, but there is an example of this for polar plots
20: (2)               in `matplotlib.projections.polar`.
21: (0)             - Any additional methods for additional convenience or features.
22: (0)             Once the projection Axes is defined, it can be used in one of two ways:
23: (0)             - By defining the class attribute ``name``, the projection Axes can be
24: (2)               registered with `matplotlib.projections.register_projection` and
subsequently
25: (2)               simply invoked by name::
26: (6)                   fig.add_subplot(projection="my_proj_name")
27: (0)             - For more complex, parameterisable projections, a generic "projection" object
28: (2)               may be defined which includes the method ``_as_mpl_axes``. ``_as_mpl_axes``
29: (2)               should take no arguments and return the projection's Axes subclass and a
30: (2)               dictionary of additional arguments to pass to the subclass' ``__init__``
31: (2)               method.  Subsequently a parameterised projection can be initialised with::
32: (6)                   fig.add_subplot(projection=MyProjection(param1=param1_value))
33: (2)               where MyProjection is an object which implements a ``_as_mpl_axes`` method.
34: (0)             A full-fledged and heavily annotated example is in
35: (0)             :doc:`/gallery/misc/custom_projection`.  The polar plot functionality in
36: (0)             `matplotlib.projections.polar` may also be of interest.
37: (0)             """
38: (0)             from .. import axes, _docstring
39: (0)             from .geo import AitoffAxes, HammerAxes, LambertAxes, MollweideAxes
```

```
40: (0)             from .polar import PolarAxes
41: (0)             try:
42: (4)                 from mpl_toolkits.mplot3d import Axes3D
43: (0)             except Exception:
44: (4)                 import warnings
45: (4)                 warnings.warn("Unable to import Axes3D. This may be due to multiple
versions of "
46: (18)                              "Matplotlib being installed (e.g. as a system package and as
a pip "
47: (18)                              "package). As a result, the 3D projection is not
available.")
48: (4)                 Axes3D = None
49: (0)             class ProjectionRegistry:
50: (4)                 """A mapping of registered projection names to projection classes."""
51: (4)                 def __init__(self):
52: (8)                     self._all_projection_types = {}
53: (4)                 def register(self, *projections):
54: (8)                     """Register a new set of projections."""
55: (8)                     for projection in projections:
56: (12)                        name = projection.name
57: (12)                        self._all_projection_types[name] = projection
58: (4)                 def get_projection_class(self, name):
59: (8)                     """Get a projection class from its *name*."""
60: (8)                     return self._all_projection_types[name]
61: (4)                 def get_projection_names(self):
62: (8)                     """Return the names of all projections currently registered."""
63: (8)                     return sorted(self._all_projection_types)
64: (0)             projection_registry = ProjectionRegistry()
65: (0)             projection_registry.register(
66: (4)                 axes.Axes,
67: (4)                 PolarAxes,
68: (4)                 AitoffAxes,
69: (4)                 HammerAxes,
70: (4)                 LambertAxes,
71: (4)                 MollweideAxes,
72: (0)             )
73: (0)             if Axes3D is not None:
74: (4)                 projection_registry.register(Axes3D)
75: (0)             else:
76: (4)                 del Axes3D
77: (0)             def register_projection(cls):
78: (4)                 projection_registry.register(cls)
79: (0)             def get_projection_class(projection=None):
80: (4)                 """
81: (4)                 Get a projection class from its name.
82: (4)                 If *projection* is None, a standard rectilinear projection is returned.
83: (4)                 """
84: (4)                 if projection is None:
85: (8)                     projection = 'rectilinear'
86: (4)                 try:
87: (8)                     return projection_registry.get_projection_class(projection)
88: (4)                 except KeyError as err:
89: (8)                     raise ValueError("Unknown projection %r" % projection) from err
90: (0)             get_projection_names = projection_registry.get_projection_names
91: (0)             _docstring.interpd.update(projection_names=get_projection_names())
```

File 117 - geo.py:

```
1: (0)              import numpy as np
2: (0)              import matplotlib as mpl
3: (0)              from matplotlib import _api
4: (0)              from matplotlib.axes import Axes
5: (0)              import matplotlib.axis as maxis
6: (0)              from matplotlib.patches import Circle
7: (0)              from matplotlib.path import Path
8: (0)              import matplotlib.spines as mspines
9: (0)              from matplotlib.ticker import (
```

```
 10: (4)                Formatter, NullLocator, FixedLocator, NullFormatter)
 11: (0)            from matplotlib.transforms import Affine2D, BboxTransformTo, Transform
 12: (0)            class GeoAxes(Axes):
 13: (4)                """An abstract base class for geographic projections."""
 14: (4)                class ThetaFormatter(Formatter):
 15: (8)                    """
 16: (8)                    Used to format the theta tick labels.  Converts the native
 17: (8)                    unit of radians into degrees and adds a degree symbol.
 18: (8)                    """
 19: (8)                    def __init__(self, round_to=1.0):
 20: (12)                       self._round_to = round_to
 21: (8)                    def __call__(self, x, pos=None):
 22: (12)                       degrees = round(np.rad2deg(x) / self._round_to) * self._round_to
 23: (12)                       return f"{degrees:0.0f}\N{DEGREE SIGN}"
 24: (4)                RESOLUTION = 75
 25: (4)                def _init_axis(self):
 26: (8)                    self.xaxis = maxis.XAxis(self, clear=False)
 27: (8)                    self.yaxis = maxis.YAxis(self, clear=False)
 28: (8)                    self.spines['geo'].register_axis(self.yaxis)
 29: (4)                def clear(self):
 30: (8)                    super().clear()
 31: (8)                    self.set_longitude_grid(30)
 32: (8)                    self.set_latitude_grid(15)
 33: (8)                    self.set_longitude_grid_ends(75)
 34: (8)                    self.xaxis.set_minor_locator(NullLocator())
 35: (8)                    self.yaxis.set_minor_locator(NullLocator())
 36: (8)                    self.xaxis.set_ticks_position('none')
 37: (8)                    self.yaxis.set_ticks_position('none')
 38: (8)                    self.yaxis.set_tick_params(label1On=True)
 39: (8)                    self.grid(mpl.rcParams['axes.grid'])
 40: (8)                    Axes.set_xlim(self, -np.pi, np.pi)
 41: (8)                    Axes.set_ylim(self, -np.pi / 2.0, np.pi / 2.0)
 42: (4)                def _set_lim_and_transforms(self):
 43: (8)                    self.transProjection = self._get_core_transform(self.RESOLUTION)
 44: (8)                    self.transAffine = self._get_affine_transform()
 45: (8)                    self.transAxes = BboxTransformTo(self.bbox)
 46: (8)                    self.transData = \
 47: (12)                       self.transProjection + \
 48: (12)                       self.transAffine + \
 49: (12)                       self.transAxes
 50: (8)                    self._xaxis_pretransform = \
 51: (12)                       Affine2D() \
 52: (12)                       .scale(1, self._longitude_cap * 2) \
 53: (12)                       .translate(0, -self._longitude_cap)
 54: (8)                    self._xaxis_transform = \
 55: (12)                       self._xaxis_pretransform + \
 56: (12)                       self.transData
 57: (8)                    self._xaxis_text1_transform = \
 58: (12)                       Affine2D().scale(1, 0) + \
 59: (12)                       self.transData + \
 60: (12)                       Affine2D().translate(0, 4)
 61: (8)                    self._xaxis_text2_transform = \
 62: (12)                       Affine2D().scale(1, 0) + \
 63: (12)                       self.transData + \
 64: (12)                       Affine2D().translate(0, -4)
 65: (8)                    yaxis_stretch = Affine2D().scale(np.pi * 2, 1).translate(-np.pi, 0)
 66: (8)                    yaxis_space = Affine2D().scale(1, 1.1)
 67: (8)                    self._yaxis_transform = \
 68: (12)                       yaxis_stretch + \
 69: (12)                       self.transData
 70: (8)                    yaxis_text_base = \
 71: (12)                       yaxis_stretch + \
 72: (12)                       self.transProjection + \
 73: (12)                       (yaxis_space +
 74: (13)                        self.transAffine +
 75: (13)                        self.transAxes)
 76: (8)                    self._yaxis_text1_transform = \
 77: (12)                       yaxis_text_base + \
 78: (12)                       Affine2D().translate(-8, 0)
```

```
79: (8)                     self._yaxis_text2_transform = \
80: (12)                        yaxis_text_base + \
81: (12)                        Affine2D().translate(8, 0)
82: (4)                 def _get_affine_transform(self):
83: (8)                     transform = self._get_core_transform(1)
84: (8)                     xscale, _ = transform.transform((np.pi, 0))
85: (8)                     _, yscale = transform.transform((0, np.pi/2))
86: (8)                     return Affine2D() \
87: (12)                        .scale(0.5 / xscale, 0.5 / yscale) \
88: (12)                        .translate(0.5, 0.5)
89: (4)                 def get_xaxis_transform(self, which='grid'):
90: (8)                     _api.check_in_list(['tick1', 'tick2', 'grid'], which=which)
91: (8)                     return self._xaxis_transform
92: (4)                 def get_xaxis_text1_transform(self, pad):
93: (8)                     return self._xaxis_text1_transform, 'bottom', 'center'
94: (4)                 def get_xaxis_text2_transform(self, pad):
95: (8)                     return self._xaxis_text2_transform, 'top', 'center'
96: (4)                 def get_yaxis_transform(self, which='grid'):
97: (8)                     _api.check_in_list(['tick1', 'tick2', 'grid'], which=which)
98: (8)                     return self._yaxis_transform
99: (4)                 def get_yaxis_text1_transform(self, pad):
100: (8)                    return self._yaxis_text1_transform, 'center', 'right'
101: (4)                def get_yaxis_text2_transform(self, pad):
102: (8)                    return self._yaxis_text2_transform, 'center', 'left'
103: (4)                def _gen_axes_patch(self):
104: (8)                    return Circle((0.5, 0.5), 0.5)
105: (4)                def _gen_axes_spines(self):
106: (8)                    return {'geo': mspines.Spine.circular_spine(self, (0.5, 0.5), 0.5)}
107: (4)                def set_yscale(self, *args, **kwargs):
108: (8)                    if args[0] != 'linear':
109: (12)                       raise NotImplementedError
110: (4)                set_xscale = set_yscale
111: (4)                def set_xlim(self, *args, **kwargs):
112: (8)                    """Not supported. Please consider using Cartopy."""
113: (8)                    raise TypeError("Changing axes limits of a geographic projection is "
114: (24)                                    "not supported.  Please consider using Cartopy.")
115: (4)                set_ylim = set_xlim
116: (4)                def format_coord(self, lon, lat):
117: (8)                    """Return a format string formatting the coordinate."""
118: (8)                    lon, lat = np.rad2deg([lon, lat])
119: (8)                    ns = 'N' if lat >= 0.0 else 'S'
120: (8)                    ew = 'E' if lon >= 0.0 else 'W'
121: (8)                    return ('%f\N{DEGREE SIGN}%s, %f\N{DEGREE SIGN}%s'
122: (16)                           % (abs(lat), ns, abs(lon), ew))
123: (4)                def set_longitude_grid(self, degrees):
124: (8)                    """
125: (8)                    Set the number of degrees between each longitude grid.
126: (8)                    """
127: (8)                    grid = np.arange(-180 + degrees, 180, degrees)
128: (8)                    self.xaxis.set_major_locator(FixedLocator(np.deg2rad(grid)))
129: (8)                    self.xaxis.set_major_formatter(self.ThetaFormatter(degrees))
130: (4)                def set_latitude_grid(self, degrees):
131: (8)                    """
132: (8)                    Set the number of degrees between each latitude grid.
133: (8)                    """
134: (8)                    grid = np.arange(-90 + degrees, 90, degrees)
135: (8)                    self.yaxis.set_major_locator(FixedLocator(np.deg2rad(grid)))
136: (8)                    self.yaxis.set_major_formatter(self.ThetaFormatter(degrees))
137: (4)                def set_longitude_grid_ends(self, degrees):
138: (8)                    """
139: (8)                    Set the latitude(s) at which to stop drawing the longitude grids.
140: (8)                    """
141: (8)                    self._longitude_cap = np.deg2rad(degrees)
142: (8)                    self._xaxis_pretransform \
143: (12)                       .clear() \
144: (12)                       .scale(1.0, self._longitude_cap * 2.0) \
145: (12)                       .translate(0.0, -self._longitude_cap)
146: (4)                def get_data_ratio(self):
147: (8)                    """Return the aspect ratio of the data itself."""
```

```
148: (8)                    return 1.0
149: (4)                def can_zoom(self):
150: (8)                    """
151: (8)                    Return whether this Axes supports the zoom box button functionality.
152: (8)                    This Axes object does not support interactive zoom box.
153: (8)                    """
154: (8)                    return False
155: (4)                def can_pan(self):
156: (8)                    """
157: (8)                    Return whether this Axes supports the pan/zoom button functionality.
158: (8)                    This Axes object does not support interactive pan/zoom.
159: (8)                    """
160: (8)                    return False
161: (4)                def start_pan(self, x, y, button):
162: (8)                    pass
163: (4)                def end_pan(self):
164: (8)                    pass
165: (4)                def drag_pan(self, button, key, x, y):
166: (8)                    pass
167: (0)            class _GeoTransform(Transform):
168: (4)                input_dims = output_dims = 2
169: (4)                def __init__(self, resolution):
170: (8)                    """
171: (8)                    Create a new geographical transform.
172: (8)                    Resolution is the number of steps to interpolate between each input
173: (8)                    line segment to approximate its path in curved space.
174: (8)                    """
175: (8)                    super().__init__()
176: (8)                    self._resolution = resolution
177: (4)                def __str__(self):
178: (8)                    return f"{type(self).__name__}({self._resolution})"
179: (4)                def transform_path_non_affine(self, path):
180: (8)                    ipath = path.interpolated(self._resolution)
181: (8)                    return Path(self.transform(ipath.vertices), ipath.codes)
182: (0)            class AitoffAxes(GeoAxes):
183: (4)                name = 'aitoff'
184: (4)                class AitoffTransform(_GeoTransform):
185: (8)                    """The base Aitoff transform."""
186: (8)                    @_api.rename_parameter("3.8", "ll", "values")
187: (8)                    def transform_non_affine(self, values):
188: (12)                       longitude, latitude = values.T
189: (12)                       half_long = longitude / 2.0
190: (12)                       cos_latitude = np.cos(latitude)
191: (12)                       alpha = np.arccos(cos_latitude * np.cos(half_long))
192: (12)                       sinc_alpha = np.sinc(alpha / np.pi)  # np.sinc is
sin(pi*x)/(pi*x).
193: (12)                       x = (cos_latitude * np.sin(half_long)) / sinc_alpha
194: (12)                       y = np.sin(latitude) / sinc_alpha
195: (12)                       return np.column_stack([x, y])
196: (8)                    def inverted(self):
197: (12)                       return AitoffAxes.InvertedAitoffTransform(self._resolution)
198: (4)                class InvertedAitoffTransform(_GeoTransform):
199: (8)                    @_api.rename_parameter("3.8", "xy", "values")
200: (8)                    def transform_non_affine(self, values):
201: (12)                       return np.full_like(values, np.nan)
202: (8)                    def inverted(self):
203: (12)                       return AitoffAxes.AitoffTransform(self._resolution)
204: (4)                def __init__(self, *args, **kwargs):
205: (8)                    self._longitude_cap = np.pi / 2.0
206: (8)                    super().__init__(*args, **kwargs)
207: (8)                    self.set_aspect(0.5, adjustable='box', anchor='C')
208: (8)                    self.clear()
209: (4)                def _get_core_transform(self, resolution):
210: (8)                    return self.AitoffTransform(resolution)
211: (0)            class HammerAxes(GeoAxes):
212: (4)                name = 'hammer'
213: (4)                class HammerTransform(_GeoTransform):
214: (8)                    """The base Hammer transform."""
215: (8)                    @_api.rename_parameter("3.8", "ll", "values")
```

```
(8)                     def transform_non_affine(self, values):
(12)                        longitude, latitude = values.T
(12)                        half_long = longitude / 2.0
(12)                        cos_latitude = np.cos(latitude)
(12)                        sqrt2 = np.sqrt(2.0)
(12)                        alpha = np.sqrt(1.0 + cos_latitude * np.cos(half_long))
(12)                        x = (2.0 * sqrt2) * (cos_latitude * np.sin(half_long)) / alpha
(12)                        y = (sqrt2 * np.sin(latitude)) / alpha
(12)                        return np.column_stack([x, y])
(8)                     def inverted(self):
(12)                        return HammerAxes.InvertedHammerTransform(self._resolution)
(4)                 class InvertedHammerTransform(_GeoTransform):
(8)                     @_api.rename_parameter("3.8", "xy", "values")
(8)                     def transform_non_affine(self, values):
(12)                        x, y = values.T
(12)                        z = np.sqrt(1 - (x / 4) ** 2 - (y / 2) ** 2)
(12)                        longitude = 2 * np.arctan((z * x) / (2 * (2 * z ** 2 - 1)))
(12)                        latitude = np.arcsin(y*z)
(12)                        return np.column_stack([longitude, latitude])
(8)                     def inverted(self):
(12)                        return HammerAxes.HammerTransform(self._resolution)
(4)                 def __init__(self, *args, **kwargs):
(8)                     self._longitude_cap = np.pi / 2.0
(8)                     super().__init__(*args, **kwargs)
(8)                     self.set_aspect(0.5, adjustable='box', anchor='C')
(8)                     self.clear()
(4)                 def _get_core_transform(self, resolution):
(8)                     return self.HammerTransform(resolution)
(0)             class MollweideAxes(GeoAxes):
(4)                 name = 'mollweide'
(4)                 class MollweideTransform(_GeoTransform):
(8)                     """The base Mollweide transform."""
(8)                     @_api.rename_parameter("3.8", "ll", "values")
(8)                     def transform_non_affine(self, values):
(12)                        def d(theta):
(16)                            delta = (-(theta + np.sin(theta) - pi_sin_l)
(25)                                     / (1 + np.cos(theta)))
(16)                            return delta, np.abs(delta) > 0.001
(12)                        longitude, latitude = values.T
(12)                        clat = np.pi/2 - np.abs(latitude)
(12)                        ihigh = clat < 0.087  # within 5 degrees of the poles
(12)                        ilow = ~ihigh
(12)                        aux = np.empty(latitude.shape, dtype=float)
(12)                        if ilow.any():  # Newton-Raphson iteration
(16)                            pi_sin_l = np.pi * np.sin(latitude[ilow])
(16)                            theta = 2.0 * latitude[ilow]
(16)                            delta, large_delta = d(theta)
(16)                            while np.any(large_delta):
(20)                                theta[large_delta] += delta[large_delta]
(20)                                delta, large_delta = d(theta)
(16)                            aux[ilow] = theta / 2
(12)                        if ihigh.any():  # Taylor series-based approx. solution
(16)                            e = clat[ihigh]
(16)                            d = 0.5 * (3 * np.pi * e**2) ** (1.0/3)
(16)                            aux[ihigh] = (np.pi/2 - d) * np.sign(latitude[ihigh])
(12)                        xy = np.empty(values.shape, dtype=float)
(12)                        xy[:, 0] = (2.0 * np.sqrt(2.0) / np.pi) * longitude * np.cos(aux)
(12)                        xy[:, 1] = np.sqrt(2.0) * np.sin(aux)
(12)                        return xy
(8)                     def inverted(self):
(12)                        return MollweideAxes.InvertedMollweideTransform(self._resolution)
(4)                 class InvertedMollweideTransform(_GeoTransform):
(8)                     @_api.rename_parameter("3.8", "xy", "values")
(8)                     def transform_non_affine(self, values):
(12)                        x, y = values.T
(12)                        theta = np.arcsin(y / np.sqrt(2))
(12)                        longitude = (np.pi / (2 * np.sqrt(2))) * x / np.cos(theta)
(12)                        latitude = np.arcsin((2 * theta + np.sin(2 * theta)) / np.pi)
(12)                        return np.column_stack([longitude, latitude])
```

```
285: (8)                    def inverted(self):
286: (12)                       return MollweideAxes.MollweideTransform(self._resolution)
287: (4)                def __init__(self, *args, **kwargs):
288: (8)                    self._longitude_cap = np.pi / 2.0
289: (8)                    super().__init__(*args, **kwargs)
290: (8)                    self.set_aspect(0.5, adjustable='box', anchor='C')
291: (8)                    self.clear()
292: (4)                def _get_core_transform(self, resolution):
293: (8)                    return self.MollweideTransform(resolution)
294: (0)            class LambertAxes(GeoAxes):
295: (4)                name = 'lambert'
296: (4)                class LambertTransform(_GeoTransform):
297: (8)                    """The base Lambert transform."""
298: (8)                    def __init__(self, center_longitude, center_latitude, resolution):
299: (12)                       """
300: (12)                       Create a new Lambert transform.  Resolution is the number of steps
301: (12)                       to interpolate between each input line segment to approximate its
302: (12)                       path in curved Lambert space.
303: (12)                       """
304: (12)                       _GeoTransform.__init__(self, resolution)
305: (12)                       self._center_longitude = center_longitude
306: (12)                       self._center_latitude = center_latitude
307: (8)                    @_api.rename_parameter("3.8", "ll", "values")
308: (8)                    def transform_non_affine(self, values):
309: (12)                       longitude, latitude = values.T
310: (12)                       clong = self._center_longitude
311: (12)                       clat = self._center_latitude
312: (12)                       cos_lat = np.cos(latitude)
313: (12)                       sin_lat = np.sin(latitude)
314: (12)                       diff_long = longitude - clong
315: (12)                       cos_diff_long = np.cos(diff_long)
316: (12)                       inner_k = np.maximum(  # Prevent divide-by-zero problems
317: (16)                           1 + np.sin(clat)*sin_lat + np.cos(clat)*cos_lat*cos_diff_long,
318: (16)                           1e-15)
319: (12)                       k = np.sqrt(2 / inner_k)
320: (12)                       x = k * cos_lat*np.sin(diff_long)
321: (12)                       y = k * (np.cos(clat)*sin_lat -
np.sin(clat)*cos_lat*cos_diff_long)
322: (12)                       return np.column_stack([x, y])
323: (8)                    def inverted(self):
324: (12)                       return LambertAxes.InvertedLambertTransform(
325: (16)                           self._center_longitude,
326: (16)                           self._center_latitude,
327: (16)                           self._resolution)
328: (4)                class InvertedLambertTransform(_GeoTransform):
329: (8)                    def __init__(self, center_longitude, center_latitude, resolution):
330: (12)                       _GeoTransform.__init__(self, resolution)
331: (12)                       self._center_longitude = center_longitude
332: (12)                       self._center_latitude = center_latitude
333: (8)                    @_api.rename_parameter("3.8", "xy", "values")
334: (8)                    def transform_non_affine(self, values):
335: (12)                       x, y = values.T
336: (12)                       clong = self._center_longitude
337: (12)                       clat = self._center_latitude
338: (12)                       p = np.maximum(np.hypot(x, y), 1e-9)
339: (12)                       c = 2 * np.arcsin(0.5 * p)
340: (12)                       sin_c = np.sin(c)
341: (12)                       cos_c = np.cos(c)
342: (12)                       latitude = np.arcsin(cos_c*np.sin(clat) +
343: (33)                                            ((y*sin_c*np.cos(clat)) / p))
344: (12)                       longitude = clong + np.arctan(
345: (16)                           (x*sin_c) / (p*np.cos(clat)*cos_c - y*np.sin(clat)*sin_c))
346: (12)                       return np.column_stack([longitude, latitude])
347: (8)                    def inverted(self):
348: (12)                       return LambertAxes.LambertTransform(
349: (16)                           self._center_longitude,
350: (16)                           self._center_latitude,
351: (16)                           self._resolution)
352: (4)                def __init__(self, *args, center_longitude=0, center_latitude=0,
```

```
**kwargs):
353: (8)                    self._longitude_cap = np.pi / 2
354: (8)                    self._center_longitude = center_longitude
355: (8)                    self._center_latitude = center_latitude
356: (8)                    super().__init__(*args, **kwargs)
357: (8)                    self.set_aspect('equal', adjustable='box', anchor='C')
358: (8)                    self.clear()
359: (4)                def clear(self):
360: (8)                    super().clear()
361: (8)                    self.yaxis.set_major_formatter(NullFormatter())
362: (4)                def _get_core_transform(self, resolution):
363: (8)                    return self.LambertTransform(
364: (12)                       self._center_longitude,
365: (12)                       self._center_latitude,
366: (12)                       resolution)
367: (4)                def _get_affine_transform(self):
368: (8)                    return Affine2D() \
369: (12)                       .scale(0.25) \
370: (12)                       .translate(0.5, 0.5)

----------------------------------------

File 118 - polar.py:

1: (0)              import math
2: (0)              import types
3: (0)              import numpy as np
4: (0)              import matplotlib as mpl
5: (0)              from matplotlib import _api, cbook
6: (0)              from matplotlib.axes import Axes
7: (0)              import matplotlib.axis as maxis
8: (0)              import matplotlib.markers as mmarkers
9: (0)              import matplotlib.patches as mpatches
10: (0)             from matplotlib.path import Path
11: (0)             import matplotlib.ticker as mticker
12: (0)             import matplotlib.transforms as mtransforms
13: (0)             from matplotlib.spines import Spine
14: (0)             def _apply_theta_transforms_warn():
15: (4)                 _api.warn_deprecated(
16: (16)                            "3.9",
17: (16)                            message=(
18: (20)                                "Passing `apply_theta_transforms=True` (the default) "
19: (20)                                "is deprecated since Matplotlib %(since)s. "
20: (20)                                "Support for this will be removed in Matplotlib %
(removal)s. "
21: (20)                                "To prevent this warning, set
`apply_theta_transforms=False`, "
22: (20)                                "and make sure to shift theta values before being passed
to "
23: (20)                                "this transform."
24: (16)                            )
25: (12)                        )
26: (0)             class PolarTransform(mtransforms.Transform):
27: (4)                 r"""
28: (4)                 The base polar transform.
29: (4)                 This transform maps polar coordinates :math:`\theta, r` into Cartesian
30: (4)                 coordinates :math:`x, y = r \cos(\theta), r \sin(\theta)`
31: (4)                 (but does not fully transform into Axes coordinates or
32: (4)                 handle positioning in screen space).
33: (4)                 This transformation is designed to be applied to data after any scaling
34: (4)                 along the radial axis (e.g. log-scaling) has been applied to the input
35: (4)                 data.
36: (4)                 Path segments at a fixed radius are automatically transformed to circular
37: (4)                 arcs as long as ``path._interpolation_steps > 1``.
38: (4)                 """
39: (4)                 input_dims = output_dims = 2
40: (4)                 def __init__(self, axis=None, use_rmin=True, *,
41: (17)                             apply_theta_transforms=True, scale_transform=None):
42: (8)                     """
```

```
(8)                     Parameters
(8)                     ----------
(8)                     axis : `~matplotlib.axis.Axis`, optional
(12)                        Axis associated with this transform. This is used to get the
(12)                        minimum radial limit.
(8)                     use_rmin : `bool`, optional
(12)                        If ``True``, subtract the minimum radial axis limit before
(12)                        transforming to Cartesian coordinates. *axis* must also be
(12)                        specified for this to take effect.
(8)                     """
(8)                     super().__init__()
(8)                     self._axis = axis
(8)                     self._use_rmin = use_rmin
(8)                     self._apply_theta_transforms = apply_theta_transforms
(8)                     self._scale_transform = scale_transform
(8)                     if apply_theta_transforms:
(12)                        _apply_theta_transforms_warn()
(4)                 __str__ = mtransforms._make_str_method(
(8)                     "_axis",
(8)                     use_rmin="_use_rmin",
(8)                     apply_theta_transforms="_apply_theta_transforms")
(4)                 def _get_rorigin(self):
(8)                     return self._scale_transform.transform(
(12)                        (0, self._axis.get_rorigin()))[1]
(4)                 @_api.rename_parameter("3.8", "tr", "values")
(4)                 def transform_non_affine(self, values):
(8)                     theta, r = np.transpose(values)
(8)                     if self._apply_theta_transforms and self._axis is not None:
(12)                        theta *= self._axis.get_theta_direction()
(12)                        theta += self._axis.get_theta_offset()
(8)                     if self._use_rmin and self._axis is not None:
(12)                        r = (r - self._get_rorigin()) * self._axis.get_rsign()
(8)                     r = np.where(r >= 0, r, np.nan)
(8)                     return np.column_stack([r * np.cos(theta), r * np.sin(theta)])
(4)                 def transform_path_non_affine(self, path):
(8)                     if not len(path) or path._interpolation_steps == 1:
(12)                        return Path(self.transform_non_affine(path.vertices), path.codes)
(8)                     xys = []
(8)                     codes = []
(8)                     last_t = last_r = None
(8)                     for trs, c in path.iter_segments():
(12)                        trs = trs.reshape((-1, 2))
(12)                        if c == Path.LINETO:
(16)                            (t, r), = trs
(16)                            if t == last_t:  # Same angle: draw a straight line.
(20)                                xys.extend(self.transform_non_affine(trs))
(20)                                codes.append(Path.LINETO)
(16)                            elif r == last_r:  # Same radius: draw an arc.
(20)                                last_td, td = np.rad2deg([last_t, t])
(20)                                if self._use_rmin and self._axis is not None:
(24)                                    r = ((r - self._get_rorigin())
(29)                                         * self._axis.get_rsign())
(20)                                if last_td <= td:
(24)                                    while td - last_td > 360:
(28)                                        arc = Path.arc(last_td, last_td + 360)
(28)                                        xys.extend(arc.vertices[1:] * r)
(28)                                        codes.extend(arc.codes[1:])
(28)                                        last_td += 360
(24)                                    arc = Path.arc(last_td, td)
(24)                                    xys.extend(arc.vertices[1:] * r)
(24)                                    codes.extend(arc.codes[1:])
(20)                                else:
(24)                                    while last_td - td > 360:
(28)                                        arc = Path.arc(last_td - 360, last_td)
(28)                                        xys.extend(arc.vertices[::-1][1:] * r)
(28)                                        codes.extend(arc.codes[1:])
(28)                                        last_td -= 360
(24)                                    arc = Path.arc(td, last_td)
(24)                                    xys.extend(arc.vertices[::-1][1:] * r)
```

```
(24)                                codes.extend(arc.codes[1:])
(16)                        else:  # Interpolate.
(20)                            trs = cbook.simple_linear_interpolation(
(24)                                np.vstack([(last_t, last_r), trs]),
(24)                                path._interpolation_steps)[1:]
(20)                            xys.extend(self.transform_non_affine(trs))
(20)                            codes.extend([Path.LINETO] * len(trs))
(12)                    else:  # Not a straight line.
(16)                        xys.extend(self.transform_non_affine(trs))
(16)                        codes.extend([c] * len(trs))
(12)                    last_t, last_r = trs[-1]
(8)                 return Path(xys, codes)
(4)             def inverted(self):
(8)                 return PolarAxes.InvertedPolarTransform(
(12)                    self._axis, self._use_rmin,
(12)                    apply_theta_transforms=self._apply_theta_transforms
(8)                 )
(0)         class PolarAffine(mtransforms.Affine2DBase):
(4)             r"""
(4)             The affine part of the polar projection.
(4)             Scales the output so that maximum radius rests on the edge of the Axes
(4)             circle and the origin is mapped to (0.5, 0.5). The transform applied is
(4)             the same to x and y components and given by:
(4)             .. math::
(8)                 x_{1} = 0.5 \left [ \frac{x_{0}}{(r_{\max} - r_{\min})} + 1 \right ]
(4)             :math:`r_{\min}, r_{\max}` are the minimum and maximum radial limits after
(4)             any scaling (e.g. log scaling) has been removed.
(4)             """
(4)             def __init__(self, scale_transform, limits):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 scale_transform : `~matplotlib.transforms.Transform`
(12)                    Scaling transform for the data. This is used to remove any scaling
(12)                    from the radial view limits.
(8)                 limits : `~matplotlib.transforms.BboxBase`
(12)                    View limits of the data. The only part of its bounds that is used
(12)                    is the y limits (for the radius limits).
(8)                 """
(8)                 super().__init__()
(8)                 self._scale_transform = scale_transform
(8)                 self._limits = limits
(8)                 self.set_children(scale_transform, limits)
(8)                 self._mtx = None
(4)             __str__ = mtransforms._make_str_method("_scale_transform", "_limits")
(4)             def get_matrix(self):
(8)                 if self._invalid:
(12)                    limits_scaled = self._limits.transformed(self._scale_transform)
(12)                    yscale = limits_scaled.ymax - limits_scaled.ymin
(12)                    affine = mtransforms.Affine2D() \
(16)                        .scale(0.5 / yscale) \
(16)                        .translate(0.5, 0.5)
(12)                    self._mtx = affine.get_matrix()
(12)                    self._inverted = None
(12)                    self._invalid = 0
(8)                 return self._mtx
(0)         class InvertedPolarTransform(mtransforms.Transform):
(4)             """
(4)             The inverse of the polar transform, mapping Cartesian
(4)             coordinate space *x* and *y* back to *theta* and *r*.
(4)             """
(4)             input_dims = output_dims = 2
(4)             def __init__(self, axis=None, use_rmin=True,
(17)                         *, apply_theta_transforms=True):
(8)                 """
(8)                 Parameters
(8)                 ----------
(8)                 axis : `~matplotlib.axis.Axis`, optional
(12)                    Axis associated with this transform. This is used to get the
```

```
(12)                        minimum radial limit.
(8)                     use_rmin : `bool`, optional
(12)                        If ``True``, add the minimum radial axis limit after
(12)                        transforming from Cartesian coordinates. *axis* must also be
(12)                        specified for this to take effect.
(8)                     """
(8)                     super().__init__()
(8)                     self._axis = axis
(8)                     self._use_rmin = use_rmin
(8)                     self._apply_theta_transforms = apply_theta_transforms
(8)                     if apply_theta_transforms:
(12)                        _apply_theta_transforms_warn()
(4)                 __str__ = mtransforms._make_str_method(
(8)                     "_axis",
(8)                     use_rmin="_use_rmin",
(8)                     apply_theta_transforms="_apply_theta_transforms")
(4)                 @_api.rename_parameter("3.8", "xy", "values")
(4)                 def transform_non_affine(self, values):
(8)                     x, y = values.T
(8)                     r = np.hypot(x, y)
(8)                     theta = (np.arctan2(y, x) + 2 * np.pi) % (2 * np.pi)
(8)                     if self._apply_theta_transforms and self._axis is not None:
(12)                        theta -= self._axis.get_theta_offset()
(12)                        theta *= self._axis.get_theta_direction()
(12)                        theta %= 2 * np.pi
(8)                     if self._use_rmin and self._axis is not None:
(12)                        r += self._axis.get_rorigin()
(12)                        r *= self._axis.get_rsign()
(8)                     return np.column_stack([theta, r])
(4)                 def inverted(self):
(8)                     return PolarAxes.PolarTransform(
(12)                        self._axis, self._use_rmin,
(12)                        apply_theta_transforms=self._apply_theta_transforms
(8)                     )
(0)             class ThetaFormatter(mticker.Formatter):
(4)                 """
(4)                 Used to format the *theta* tick labels.  Converts the native
(4)                 unit of radians into degrees and adds a degree symbol.
(4)                 """
(4)                 def __call__(self, x, pos=None):
(8)                     vmin, vmax = self.axis.get_view_interval()
(8)                     d = np.rad2deg(abs(vmax - vmin))
(8)                     digits = max(-int(np.log10(d) - 1.5), 0)
(8)                     return f"{np.rad2deg(x):0.{digits}f}\N{DEGREE SIGN}"
(0)             class _AxisWrapper:
(4)                 def __init__(self, axis):
(8)                     self._axis = axis
(4)                 def get_view_interval(self):
(8)                     return np.rad2deg(self._axis.get_view_interval())
(4)                 def set_view_interval(self, vmin, vmax):
(8)                     self._axis.set_view_interval(*np.deg2rad((vmin, vmax)))
(4)                 def get_minpos(self):
(8)                     return np.rad2deg(self._axis.get_minpos())
(4)                 def get_data_interval(self):
(8)                     return np.rad2deg(self._axis.get_data_interval())
(4)                 def set_data_interval(self, vmin, vmax):
(8)                     self._axis.set_data_interval(*np.deg2rad((vmin, vmax)))
(4)                 def get_tick_space(self):
(8)                     return self._axis.get_tick_space()
(0)             class ThetaLocator(mticker.Locator):
(4)                 """
(4)                 Used to locate theta ticks.
(4)                 This will work the same as the base locator except in the case that the
(4)                 view spans the entire circle. In such cases, the previously used default
(4)                 locations of every 45 degrees are returned.
(4)                 """
(4)                 def __init__(self, base):
(8)                     self.base = base
(8)                     self.axis = self.base.axis = _AxisWrapper(self.base.axis)
```

```
(4)     def set_axis(self, axis):
(8)         self.axis = _AxisWrapper(axis)
(8)         self.base.set_axis(self.axis)
(4)     def __call__(self):
(8)         lim = self.axis.get_view_interval()
(8)         if _is_full_circle_deg(lim[0], lim[1]):
(12)            return np.deg2rad(min(lim)) + np.arange(8) * 2 * np.pi / 8
(8)         else:
(12)            return np.deg2rad(self.base())
(4)     def view_limits(self, vmin, vmax):
(8)         vmin, vmax = np.rad2deg((vmin, vmax))
(8)         return np.deg2rad(self.base.view_limits(vmin, vmax))
(0) class ThetaTick(maxis.XTick):
(4)     """
(4)     A theta-axis tick.
(4)     This subclass of `.XTick` provides angular ticks with some small
(4)     modification to their re-positioning such that ticks are rotated based on
(4)     tick location. This results in ticks that are correctly perpendicular to
(4)     the arc spine.
(4)     When 'auto' rotation is enabled, labels are also rotated to be parallel to
(4)     the spine. The label padding is also applied here since it's not possible
(4)     to use a generic axes transform to produce tick-specific padding.
(4)     """
(4)     def __init__(self, axes, *args, **kwargs):
(8)         self._text1_translate = mtransforms.ScaledTranslation(
(12)            0, 0, axes.figure.dpi_scale_trans)
(8)         self._text2_translate = mtransforms.ScaledTranslation(
(12)            0, 0, axes.figure.dpi_scale_trans)
(8)         super().__init__(axes, *args, **kwargs)
(8)         self.label1.set(
(12)            rotation_mode='anchor',
(12)            transform=self.label1.get_transform() + self._text1_translate)
(8)         self.label2.set(
(12)            rotation_mode='anchor',
(12)            transform=self.label2.get_transform() + self._text2_translate)
(4)     def _apply_params(self, **kwargs):
(8)         super()._apply_params(**kwargs)
(8)         trans = self.label1.get_transform()
(8)         if not trans.contains_branch(self._text1_translate):
(12)            self.label1.set_transform(trans + self._text1_translate)
(8)         trans = self.label2.get_transform()
(8)         if not trans.contains_branch(self._text2_translate):
(12)            self.label2.set_transform(trans + self._text2_translate)
(4)     def _update_padding(self, pad, angle):
(8)         padx = pad * np.cos(angle) / 72
(8)         pady = pad * np.sin(angle) / 72
(8)         self._text1_translate._t = (padx, pady)
(8)         self._text1_translate.invalidate()
(8)         self._text2_translate._t = (-padx, -pady)
(8)         self._text2_translate.invalidate()
(4)     def update_position(self, loc):
(8)         super().update_position(loc)
(8)         axes = self.axes
(8)         angle = loc * axes.get_theta_direction() + axes.get_theta_offset()
(8)         text_angle = np.rad2deg(angle) % 360 - 90
(8)         angle -= np.pi / 2
(8)         marker = self.tick1line.get_marker()
(8)         if marker in (mmarkers.TICKUP, '|'):
(12)            trans = mtransforms.Affine2D().scale(1, 1).rotate(angle)
(8)         elif marker == mmarkers.TICKDOWN:
(12)            trans = mtransforms.Affine2D().scale(1, -1).rotate(angle)
(8)         else:
(12)            trans = self.tick1line._marker._transform
(8)         self.tick1line._marker._transform = trans
(8)         marker = self.tick2line.get_marker()
(8)         if marker in (mmarkers.TICKUP, '|'):
(12)            trans = mtransforms.Affine2D().scale(1, 1).rotate(angle)
(8)         elif marker == mmarkers.TICKDOWN:
(12)            trans = mtransforms.Affine2D().scale(1, -1).rotate(angle)
```

```
319: (8)                    else:
320: (12)                       trans = self.tick2line._marker._transform
321: (8)                    self.tick2line._marker._transform = trans
322: (8)                    mode, user_angle = self._labelrotation
323: (8)                    if mode == 'default':
324: (12)                       text_angle = user_angle
325: (8)                    else:
326: (12)                       if text_angle > 90:
327: (16)                           text_angle -= 180
328: (12)                       elif text_angle < -90:
329: (16)                           text_angle += 180
330: (12)                       text_angle += user_angle
331: (8)                    self.label1.set_rotation(text_angle)
332: (8)                    self.label2.set_rotation(text_angle)
333: (8)                    pad = self._pad + 7
334: (8)                    self._update_padding(pad,
335: (29)                                        self._loc * axes.get_theta_direction() +
336: (29)                                        axes.get_theta_offset())
337: (0)            class ThetaAxis(maxis.XAxis):
338: (4)                """
339: (4)                A theta Axis.
340: (4)                This overrides certain properties of an `.XAxis` to provide special-casing
341: (4)                for an angular axis.
342: (4)                """
343: (4)                __name__ = 'thetaaxis'
344: (4)                axis_name = 'theta'  #: Read-only name identifying the axis.
345: (4)                _tick_class = ThetaTick
346: (4)                def _wrap_locator_formatter(self):
347: (8)                    self.set_major_locator(ThetaLocator(self.get_major_locator()))
348: (8)                    self.set_major_formatter(ThetaFormatter())
349: (8)                    self.isDefault_majloc = True
350: (8)                    self.isDefault_majfmt = True
351: (4)                def clear(self):
352: (8)                    super().clear()
353: (8)                    self.set_ticks_position('none')
354: (8)                    self._wrap_locator_formatter()
355: (4)                def _set_scale(self, value, **kwargs):
356: (8)                    if value != 'linear':
357: (12)                       raise NotImplementedError(
358: (16)                           "The xscale cannot be set on a polar plot")
359: (8)                    super()._set_scale(value, **kwargs)
360: (8)                    self.get_major_locator().set_params(steps=[1, 1.5, 3, 4.5, 9, 10])
361: (8)                    self._wrap_locator_formatter()
362: (4)                def _copy_tick_props(self, src, dest):
363: (8)                    """Copy the props from src tick to dest tick."""
364: (8)                    if src is None or dest is None:
365: (12)                       return
366: (8)                    super()._copy_tick_props(src, dest)
367: (8)                    trans = dest._get_text1_transform()[0]
368: (8)                    dest.label1.set_transform(trans + dest._text1_translate)
369: (8)                    trans = dest._get_text2_transform()[0]
370: (8)                    dest.label2.set_transform(trans + dest._text2_translate)
371: (0)            class RadialLocator(mticker.Locator):
372: (4)                """
373: (4)                Used to locate radius ticks.
374: (4)                Ensures that all ticks are strictly positive.  For all other tasks, it
375: (4)                delegates to the base `.Locator` (which may be different depending on the
376: (4)                scale of the *r*-axis).
377: (4)                """
378: (4)                def __init__(self, base, axes=None):
379: (8)                    self.base = base
380: (8)                    self._axes = axes
381: (4)                def set_axis(self, axis):
382: (8)                    self.base.set_axis(axis)
383: (4)                def __call__(self):
384: (8)                    if self._axes:
385: (12)                       if _is_full_circle_rad(*self._axes.viewLim.intervalx):
386: (16)                           rorigin = self._axes.get_rorigin() * self._axes.get_rsign()
387: (16)                           if self._axes.get_rmin() <= rorigin:
```

```
(20)                                return [tick for tick in self.base() if tick > rorigin]
(8)                     return self.base()
(4)                 def _zero_in_bounds(self):
(8)                     """
(8)                     Return True if zero is within the valid values for the
(8)                     scale of the radial axis.
(8)                     """
(8)                     vmin, vmax = self._axes.yaxis._scale.limit_range_for_scale(0, 1, 1e-5)
(8)                     return vmin == 0
(4)                 def nonsingular(self, vmin, vmax):
(8)                     if self._zero_in_bounds() and (vmin, vmax) == (-np.inf, np.inf):
(12)                        return (0, 1)
(8)                     else:
(12)                        return self.base.nonsingular(vmin, vmax)
(4)                 def view_limits(self, vmin, vmax):
(8)                     vmin, vmax = self.base.view_limits(vmin, vmax)
(8)                     if self._zero_in_bounds() and vmax > vmin:
(12)                        vmin = min(0, vmin)
(8)                     return mtransforms.nonsingular(vmin, vmax)
(0)             class _ThetaShift(mtransforms.ScaledTranslation):
(4)                 """
(4)                 Apply a padding shift based on axes theta limits.
(4)                 This is used to create padding for radial ticks.
(4)                 Parameters
(4)                 ----------
(4)                 axes : `~matplotlib.axes.Axes`
(8)                     The owning Axes; used to determine limits.
(4)                 pad : float
(8)                     The padding to apply, in points.
(4)                 mode : {'min', 'max', 'rlabel'}
(8)                     Whether to shift away from the start (``'min'``) or the end
(``'max'``)
(8)                     of the axes, or using the rlabel position (``'rlabel'``).
(4)                 """
(4)                 def __init__(self, axes, pad, mode):
(8)                     super().__init__(pad, pad, axes.figure.dpi_scale_trans)
(8)                     self.set_children(axes._realViewLim)
(8)                     self.axes = axes
(8)                     self.mode = mode
(8)                     self.pad = pad
(4)                 __str__ = mtransforms._make_str_method("axes", "pad", "mode")
(4)                 def get_matrix(self):
(8)                     if self._invalid:
(12)                        if self.mode == 'rlabel':
(16)                            angle = (
(20)                                np.deg2rad(self.axes.get_rlabel_position()
(31)                                           * self.axes.get_theta_direction())
(20)                                + self.axes.get_theta_offset()
(20)                                - np.pi / 2
(16)                            )
(12)                        elif self.mode == 'min':
(16)                            angle = self.axes._realViewLim.xmin - np.pi / 2
(12)                        elif self.mode == 'max':
(16)                            angle = self.axes._realViewLim.xmax + np.pi / 2
(12)                        self._t = (self.pad * np.cos(angle) / 72, self.pad * np.sin(angle)
/ 72)
(8)                     return super().get_matrix()
(0)             class RadialTick(maxis.YTick):
(4)                 """
(4)                 A radial-axis tick.
(4)                 This subclass of `.YTick` provides radial ticks with some small
(4)                 modification to their re-positioning such that ticks are rotated based on
(4)                 axes limits.  This results in ticks that are correctly perpendicular to
(4)                 the spine. Labels are also rotated to be perpendicular to the spine, when
(4)                 'auto' rotation is enabled.
(4)                 """
(4)                 def __init__(self, *args, **kwargs):
(8)                     super().__init__(*args, **kwargs)
(8)                     self.label1.set_rotation_mode('anchor')
```

```
(8)                 self.label2.set_rotation_mode('anchor')
(4)             def _determine_anchor(self, mode, angle, start):
(8)                 if mode == 'auto':
(12)                    if start:
(16)                        if -90 <= angle <= 90:
(20)                            return 'left', 'center'
(16)                        else:
(20)                            return 'right', 'center'
(12)                    else:
(16)                        if -90 <= angle <= 90:
(20)                            return 'right', 'center'
(16)                        else:
(20)                            return 'left', 'center'
(8)                 else:
(12)                    if start:
(16)                        if angle < -68.5:
(20)                            return 'center', 'top'
(16)                        elif angle < -23.5:
(20)                            return 'left', 'top'
(16)                        elif angle < 22.5:
(20)                            return 'left', 'center'
(16)                        elif angle < 67.5:
(20)                            return 'left', 'bottom'
(16)                        elif angle < 112.5:
(20)                            return 'center', 'bottom'
(16)                        elif angle < 157.5:
(20)                            return 'right', 'bottom'
(16)                        elif angle < 202.5:
(20)                            return 'right', 'center'
(16)                        elif angle < 247.5:
(20)                            return 'right', 'top'
(16)                        else:
(20)                            return 'center', 'top'
(12)                    else:
(16)                        if angle < -68.5:
(20)                            return 'center', 'bottom'
(16)                        elif angle < -23.5:
(20)                            return 'right', 'bottom'
(16)                        elif angle < 22.5:
(20)                            return 'right', 'center'
(16)                        elif angle < 67.5:
(20)                            return 'right', 'top'
(16)                        elif angle < 112.5:
(20)                            return 'center', 'top'
(16)                        elif angle < 157.5:
(20)                            return 'left', 'top'
(16)                        elif angle < 202.5:
(20)                            return 'left', 'center'
(16)                        elif angle < 247.5:
(20)                            return 'left', 'bottom'
(16)                        else:
(20)                            return 'center', 'bottom'
(4)             def update_position(self, loc):
(8)                 super().update_position(loc)
(8)                 axes = self.axes
(8)                 thetamin = axes.get_thetamin()
(8)                 thetamax = axes.get_thetamax()
(8)                 direction = axes.get_theta_direction()
(8)                 offset_rad = axes.get_theta_offset()
(8)                 offset = np.rad2deg(offset_rad)
(8)                 full = _is_full_circle_deg(thetamin, thetamax)
(8)                 if full:
(12)                    angle = (axes.get_rlabel_position() * direction +
(21)                             offset) % 360 - 90
(12)                    tick_angle = 0
(8)                 else:
(12)                    angle = (thetamin * direction + offset) % 360 - 90
(12)                    if direction > 0:
(16)                        tick_angle = np.deg2rad(angle)
```

```
(12)                        else:
(16)                            tick_angle = np.deg2rad(angle + 180)
(8)                     text_angle = (angle + 90) % 180 - 90  # between -90 and +90.
(8)                     mode, user_angle = self._labelrotation
(8)                     if mode == 'auto':
(12)                        text_angle += user_angle
(8)                     else:
(12)                        text_angle = user_angle
(8)                     if full:
(12)                        ha = self.label1.get_horizontalalignment()
(12)                        va = self.label1.get_verticalalignment()
(8)                     else:
(12)                        ha, va = self._determine_anchor(mode, angle, direction > 0)
(8)                     self.label1.set_horizontalalignment(ha)
(8)                     self.label1.set_verticalalignment(va)
(8)                     self.label1.set_rotation(text_angle)
(8)                     marker = self.tick1line.get_marker()
(8)                     if marker == mmarkers.TICKLEFT:
(12)                        trans = mtransforms.Affine2D().rotate(tick_angle)
(8)                     elif marker == '_':
(12)                        trans = mtransforms.Affine2D().rotate(tick_angle + np.pi / 2)
(8)                     elif marker == mmarkers.TICKRIGHT:
(12)                        trans = mtransforms.Affine2D().scale(-1, 1).rotate(tick_angle)
(8)                     else:
(12)                        trans = self.tick1line._marker._transform
(8)                     self.tick1line._marker._transform = trans
(8)                     if full:
(12)                        self.label2.set_visible(False)
(12)                        self.tick2line.set_visible(False)
(8)                     angle = (thetamax * direction + offset) % 360 - 90
(8)                     if direction > 0:
(12)                        tick_angle = np.deg2rad(angle)
(8)                     else:
(12)                        tick_angle = np.deg2rad(angle + 180)
(8)                     text_angle = (angle + 90) % 180 - 90  # between -90 and +90.
(8)                     mode, user_angle = self._labelrotation
(8)                     if mode == 'auto':
(12)                        text_angle += user_angle
(8)                     else:
(12)                        text_angle = user_angle
(8)                     ha, va = self._determine_anchor(mode, angle, direction < 0)
(8)                     self.label2.set_ha(ha)
(8)                     self.label2.set_va(va)
(8)                     self.label2.set_rotation(text_angle)
(8)                     marker = self.tick2line.get_marker()
(8)                     if marker == mmarkers.TICKLEFT:
(12)                        trans = mtransforms.Affine2D().rotate(tick_angle)
(8)                     elif marker == '_':
(12)                        trans = mtransforms.Affine2D().rotate(tick_angle + np.pi / 2)
(8)                     elif marker == mmarkers.TICKRIGHT:
(12)                        trans = mtransforms.Affine2D().scale(-1, 1).rotate(tick_angle)
(8)                     else:
(12)                        trans = self.tick2line._marker._transform
(8)                     self.tick2line._marker._transform = trans
(0)             class RadialAxis(maxis.YAxis):
(4)                 """
(4)                 A radial Axis.
(4)                 This overrides certain properties of a `.YAxis` to provide special-casing
(4)                 for a radial axis.
(4)                 """
(4)                 __name__ = 'radialaxis'
(4)                 axis_name = 'radius'  #: Read-only name identifying the axis.
(4)                 _tick_class = RadialTick
(4)                 def __init__(self, *args, **kwargs):
(8)                     super().__init__(*args, **kwargs)
(8)                     self.sticky_edges.y.append(0)
(4)                 def _wrap_locator_formatter(self):
(8)                     self.set_major_locator(RadialLocator(self.get_major_locator(),
(45)                                                          self.axes))
```

```
593: (8)                    self.isDefault_majloc = True
594: (4)                def clear(self):
595: (8)                    super().clear()
596: (8)                    self.set_ticks_position('none')
597: (8)                    self._wrap_locator_formatter()
598: (4)                def _set_scale(self, value, **kwargs):
599: (8)                    super()._set_scale(value, **kwargs)
600: (8)                    self._wrap_locator_formatter()
601: (0)            def _is_full_circle_deg(thetamin, thetamax):
602: (4)                """
603: (4)                Determine if a wedge (in degrees) spans the full circle.
604: (4)                The condition is derived from :class:`~matplotlib.patches.Wedge`.
605: (4)                """
606: (4)                return abs(abs(thetamax - thetamin) - 360.0) < 1e-12
607: (0)            def _is_full_circle_rad(thetamin, thetamax):
608: (4)                """
609: (4)                Determine if a wedge (in radians) spans the full circle.
610: (4)                The condition is derived from :class:`~matplotlib.patches.Wedge`.
611: (4)                """
612: (4)                return abs(abs(thetamax - thetamin) - 2 * np.pi) < 1.74e-14
613: (0)            class _WedgeBbox(mtransforms.Bbox):
614: (4)                """
615: (4)                Transform (theta, r) wedge Bbox into Axes bounding box.
616: (4)                Parameters
617: (4)                ----------
618: (4)                center : (float, float)
619: (8)                    Center of the wedge
620: (4)                viewLim : `~matplotlib.transforms.Bbox`
621: (8)                    Bbox determining the boundaries of the wedge
622: (4)                originLim : `~matplotlib.transforms.Bbox`
623: (8)                    Bbox determining the origin for the wedge, if different from *viewLim*
624: (4)                """
625: (4)                def __init__(self, center, viewLim, originLim, **kwargs):
626: (8)                    super().__init__([[0, 0], [1, 1]], **kwargs)
627: (8)                    self._center = center
628: (8)                    self._viewLim = viewLim
629: (8)                    self._originLim = originLim
630: (8)                    self.set_children(viewLim, originLim)
631: (4)                __str__ = mtransforms._make_str_method("_center", "_viewLim",
"_originLim")
632: (4)                def get_points(self):
633: (8)                    if self._invalid:
634: (12)                       points = self._viewLim.get_points().copy()
635: (12)                       points[:, 0] *= 180 / np.pi
636: (12)                       if points[0, 0] > points[1, 0]:
637: (16)                           points[:, 0] = points[::-1, 0]
638: (12)                       points[:, 1] -= self._originLim.y0
639: (12)                       rscale = 0.5 / points[1, 1]
640: (12)                       points[:, 1] *= rscale
641: (12)                       width = min(points[1, 1] - points[0, 1], 0.5)
642: (12)                       wedge = mpatches.Wedge(self._center, points[1, 1],
643: (35)                                              points[0, 0], points[1, 0],
644: (35)                                              width=width)
645: (12)                       self.update_from_path(wedge.get_path())
646: (12)                       w, h = self._points[1] - self._points[0]
647: (12)                       deltah = max(w - h, 0) / 2
648: (12)                       deltaw = max(h - w, 0) / 2
649: (12)                       self._points += np.array([[-deltaw, -deltah], [deltaw, deltah]])
650: (12)                       self._invalid = 0
651: (8)                    return self._points
652: (0)            class PolarAxes(Axes):
653: (4)                """
654: (4)                A polar graph projection, where the input dimensions are *theta*, *r*.
655: (4)                Theta starts pointing east and goes anti-clockwise.
656: (4)                """
657: (4)                name = 'polar'
658: (4)                def __init__(self, *args,
659: (17)                            theta_offset=0, theta_direction=1, rlabel_position=22.5,
660: (17)                            **kwargs):
```

```
661: (8)                    self._default_theta_offset = theta_offset
662: (8)                    self._default_theta_direction = theta_direction
663: (8)                    self._default_rlabel_position = np.deg2rad(rlabel_position)
664: (8)                    super().__init__(*args, **kwargs)
665: (8)                    self.use_sticky_edges = True
666: (8)                    self.set_aspect('equal', adjustable='box', anchor='C')
667: (8)                    self.clear()
668: (4)                def clear(self):
669: (8)                    super().clear()
670: (8)                    self.title.set_y(1.05)
671: (8)                    start = self.spines.get('start', None)
672: (8)                    if start:
673: (12)                       start.set_visible(False)
674: (8)                    end = self.spines.get('end', None)
675: (8)                    if end:
676: (12)                       end.set_visible(False)
677: (8)                    self.set_xlim(0.0, 2 * np.pi)
678: (8)                    self.grid(mpl.rcParams['polaraxes.grid'])
679: (8)                    inner = self.spines.get('inner', None)
680: (8)                    if inner:
681: (12)                       inner.set_visible(False)
682: (8)                    self.set_rorigin(None)
683: (8)                    self.set_theta_offset(self._default_theta_offset)
684: (8)                    self.set_theta_direction(self._default_theta_direction)
685: (4)                def _init_axis(self):
686: (8)                    self.xaxis = ThetaAxis(self, clear=False)
687: (8)                    self.yaxis = RadialAxis(self, clear=False)
688: (8)                    self.spines['polar'].register_axis(self.yaxis)
689: (4)                def _set_lim_and_transforms(self):
690: (8)                    self._originViewLim = mtransforms.LockableBbox(self.viewLim)
691: (8)                    self._direction = mtransforms.Affine2D() \
692: (12)                       .scale(self._default_theta_direction, 1.0)
693: (8)                    self._theta_offset = mtransforms.Affine2D() \
694: (12)                       .translate(self._default_theta_offset, 0.0)
695: (8)                    self.transShift = self._direction + self._theta_offset
696: (8)                    self._realViewLim = mtransforms.TransformedBbox(self.viewLim,
697: (56)                                                                   self.transShift)
698: (8)                    self.transScale = mtransforms.TransformWrapper(
699: (12)                       mtransforms.IdentityTransform())
700: (8)                    self.axesLim = _WedgeBbox((0.5, 0.5),
701: (34)                                             self._realViewLim, self._originViewLim)
702: (8)                    self.transWedge = mtransforms.BboxTransformFrom(self.axesLim)
703: (8)                    self.transAxes = mtransforms.BboxTransformTo(self.bbox)
704: (8)                    self.transProjection = self.PolarTransform(
705: (12)                       self,
706: (12)                       apply_theta_transforms=False,
707: (12)                       scale_transform=self.transScale
708: (8)                    )
709: (8)                    self.transProjection.set_children(self._originViewLim)
710: (8)                    self.transProjectionAffine = self.PolarAffine(self.transScale,
711: (54)                                                                 self._originViewLim)
712: (8)                    self.transData = (
713: (12)                       self.transScale +
714: (12)                       self.transShift +
715: (12)                       self.transProjection +
716: (12)                       (
717: (16)                           self.transProjectionAffine +
718: (16)                           self.transWedge +
719: (16)                           self.transAxes
720: (12)                       )
721: (8)                    )
722: (8)                    self._xaxis_transform = (
723: (12)                       mtransforms.blended_transform_factory(
724: (16)                           mtransforms.IdentityTransform(),
725: (16)                           mtransforms.BboxTransformTo(self.viewLim)) +
726: (12)                       self.transData)
727: (8)                    flipr_transform = mtransforms.Affine2D() \
728: (12)                       .translate(0.0, -0.5) \
729: (12)                       .scale(1.0, -1.0) \
```

```
(12)                        .translate(0.0, 0.5)
(8)                     self._xaxis_text_transform = flipr_transform + self._xaxis_transform
(8)                     self._yaxis_transform = (
(12)                        mtransforms.blended_transform_factory(
(16)                            mtransforms.BboxTransformTo(self.viewLim),
(16)                            mtransforms.IdentityTransform()) +
(12)                        self.transData)
(8)                     self._r_label_position = mtransforms.Affine2D() \
(12)                        .translate(self._default_rlabel_position, 0.0)
(8)                     self._yaxis_text_transform = mtransforms.TransformWrapper(
(12)                        self._r_label_position + self.transData)
(4)                 def get_xaxis_transform(self, which='grid'):
(8)                     _api.check_in_list(['tick1', 'tick2', 'grid'], which=which)
(8)                     return self._xaxis_transform
(4)                 def get_xaxis_text1_transform(self, pad):
(8)                     return self._xaxis_text_transform, 'center', 'center'
(4)                 def get_xaxis_text2_transform(self, pad):
(8)                     return self._xaxis_text_transform, 'center', 'center'
(4)                 def get_yaxis_transform(self, which='grid'):
(8)                     if which in ('tick1', 'tick2'):
(12)                        return self._yaxis_text_transform
(8)                     elif which == 'grid':
(12)                        return self._yaxis_transform
(8)                     else:
(12)                        _api.check_in_list(['tick1', 'tick2', 'grid'], which=which)
(4)                 def get_yaxis_text1_transform(self, pad):
(8)                     thetamin, thetamax = self._realViewLim.intervalx
(8)                     if _is_full_circle_rad(thetamin, thetamax):
(12)                        return self._yaxis_text_transform, 'bottom', 'left'
(8)                     elif self.get_theta_direction() > 0:
(12)                        halign = 'left'
(12)                        pad_shift = _ThetaShift(self, pad, 'min')
(8)                     else:
(12)                        halign = 'right'
(12)                        pad_shift = _ThetaShift(self, pad, 'max')
(8)                     return self._yaxis_text_transform + pad_shift, 'center', halign
(4)                 def get_yaxis_text2_transform(self, pad):
(8)                     if self.get_theta_direction() > 0:
(12)                        halign = 'right'
(12)                        pad_shift = _ThetaShift(self, pad, 'max')
(8)                     else:
(12)                        halign = 'left'
(12)                        pad_shift = _ThetaShift(self, pad, 'min')
(8)                     return self._yaxis_text_transform + pad_shift, 'center', halign
(4)                 def draw(self, renderer):
(8)                     self._unstale_viewLim()
(8)                     thetamin, thetamax = np.rad2deg(self._realViewLim.intervalx)
(8)                     if thetamin > thetamax:
(12)                        thetamin, thetamax = thetamax, thetamin
(8)                     rmin, rmax = ((self._realViewLim.intervaly - self.get_rorigin()) *
(22)                                  self.get_rsign())
(8)                     if isinstance(self.patch, mpatches.Wedge):
(12)                        center = self.transWedge.transform((0.5, 0.5))
(12)                        self.patch.set_center(center)
(12)                        self.patch.set_theta1(thetamin)
(12)                        self.patch.set_theta2(thetamax)
(12)                        edge, _ = self.transWedge.transform((1, 0))
(12)                        radius = edge - center[0]
(12)                        width = min(radius * (rmax - rmin) / rmax, radius)
(12)                        self.patch.set_radius(radius)
(12)                        self.patch.set_width(width)
(12)                        inner_width = radius - width
(12)                        inner = self.spines.get('inner', None)
(12)                        if inner:
(16)                            inner.set_visible(inner_width != 0.0)
(8)                     visible = not _is_full_circle_deg(thetamin, thetamax)
(8)                     start = self.spines.get('start', None)
(8)                     end = self.spines.get('end', None)
(8)                     if start:
```

```
799: (12)                        start.set_visible(visible)
800: (8)                    if end:
801: (12)                        end.set_visible(visible)
802: (8)                    if visible:
803: (12)                        yaxis_text_transform = self._yaxis_transform
804: (8)                    else:
805: (12)                        yaxis_text_transform = self._r_label_position + self.transData
806: (8)                    if self._yaxis_text_transform != yaxis_text_transform:
807: (12)                        self._yaxis_text_transform.set(yaxis_text_transform)
808: (12)                        self.yaxis.reset_ticks()
809: (12)                        self.yaxis.set_clip_path(self.patch)
810: (8)                    super().draw(renderer)
811: (4)                def _gen_axes_patch(self):
812: (8)                    return mpatches.Wedge((0.5, 0.5), 0.5, 0.0, 360.0)
813: (4)                def _gen_axes_spines(self):
814: (8)                    spines = {
815: (12)                        'polar': Spine.arc_spine(self, 'top', (0.5, 0.5), 0.5, 0, 360),
816: (12)                        'start': Spine.linear_spine(self, 'left'),
817: (12)                        'end': Spine.linear_spine(self, 'right'),
818: (12)                        'inner': Spine.arc_spine(self, 'bottom', (0.5, 0.5), 0.0, 0, 360),
819: (8)                    }
820: (8)                    spines['polar'].set_transform(self.transWedge + self.transAxes)
821: (8)                    spines['inner'].set_transform(self.transWedge + self.transAxes)
822: (8)                    spines['start'].set_transform(self._yaxis_transform)
823: (8)                    spines['end'].set_transform(self._yaxis_transform)
824: (8)                    return spines
825: (4)                def set_thetamax(self, thetamax):
826: (8)                    """Set the maximum theta limit in degrees."""
827: (8)                    self.viewLim.x1 = np.deg2rad(thetamax)
828: (4)                def get_thetamax(self):
829: (8)                    """Return the maximum theta limit in degrees."""
830: (8)                    return np.rad2deg(self.viewLim.xmax)
831: (4)                def set_thetamin(self, thetamin):
832: (8)                    """Set the minimum theta limit in degrees."""
833: (8)                    self.viewLim.x0 = np.deg2rad(thetamin)
834: (4)                def get_thetamin(self):
835: (8)                    """Get the minimum theta limit in degrees."""
836: (8)                    return np.rad2deg(self.viewLim.xmin)
837: (4)                def set_thetalim(self, *args, **kwargs):
838: (8)                    r"""
839: (8)                    Set the minimum and maximum theta values.
840: (8)                    Can take the following signatures:
841: (8)                    - ``set_thetalim(minval, maxval)``: Set the limits in radians.
842: (8)                    - ``set_thetalim(thetamin=minval, thetamax=maxval)``: Set the limits
843: (10)                     in degrees.
844: (8)                    where minval and maxval are the minimum and maximum limits. Values are
845: (8)                    wrapped in to the range :math:`[0, 2\pi]` (in radians), so for example
846: (8)                    it is possible to do ``set_thetalim(-np.pi / 2, np.pi / 2)`` to have
847: (8)                    an axis symmetric around 0. A ValueError is raised if the absolute
848: (8)                    angle difference is larger than a full circle.
849: (8)                    """
850: (8)                    orig_lim = self.get_xlim()  # in radians
851: (8)                    if 'thetamin' in kwargs:
852: (12)                        kwargs['xmin'] = np.deg2rad(kwargs.pop('thetamin'))
853: (8)                    if 'thetamax' in kwargs:
854: (12)                        kwargs['xmax'] = np.deg2rad(kwargs.pop('thetamax'))
855: (8)                    new_min, new_max = self.set_xlim(*args, **kwargs)
856: (8)                    if abs(new_max - new_min) > 2 * np.pi:
857: (12)                        self.set_xlim(orig_lim)  # un-accept the change
858: (12)                        raise ValueError("The angle range must be less than a full
circle")
859: (8)                    return tuple(np.rad2deg((new_min, new_max)))
860: (4)                def set_theta_offset(self, offset):
861: (8)                    """
862: (8)                    Set the offset for the location of 0 in radians.
863: (8)                    """
864: (8)                    mtx = self._theta_offset.get_matrix()
865: (8)                    mtx[0, 2] = offset
866: (8)                    self._theta_offset.invalidate()
```

```
(4)     def get_theta_offset(self):
(8)         """
(8)         Get the offset for the location of 0 in radians.
(8)         """
(8)         return self._theta_offset.get_matrix()[0, 2]
(4)     def set_theta_zero_location(self, loc, offset=0.0):
(8)         """
(8)         Set the location of theta's zero.
(8)         This simply calls `set_theta_offset` with the correct value in
radians.
(8)         Parameters
(8)         ----------
(8)         loc : str
(12)            May be one of "N", "NW", "W", "SW", "S", "SE", "E", or "NE".
(8)         offset : float, default: 0
(12)            An offset in degrees to apply from the specified *loc*. **Note:**
(12)            this offset is *always* applied counter-clockwise regardless of
(12)            the direction setting.
(8)         """
(8)         mapping = {
(12)            'N': np.pi * 0.5,
(12)            'NW': np.pi * 0.75,
(12)            'W': np.pi,
(12)            'SW': np.pi * 1.25,
(12)            'S': np.pi * 1.5,
(12)            'SE': np.pi * 1.75,
(12)            'E': 0,
(12)            'NE': np.pi * 0.25}
(8)         return self.set_theta_offset(mapping[loc] + np.deg2rad(offset))
(4)     def set_theta_direction(self, direction):
(8)         """
(8)         Set the direction in which theta increases.
(8)         clockwise, -1:
(11)           Theta increases in the clockwise direction
(8)         counterclockwise, anticlockwise, 1:
(11)           Theta increases in the counterclockwise direction
(8)         """
(8)         mtx = self._direction.get_matrix()
(8)         if direction in ('clockwise', -1):
(12)            mtx[0, 0] = -1
(8)         elif direction in ('counterclockwise', 'anticlockwise', 1):
(12)            mtx[0, 0] = 1
(8)         else:
(12)            _api.check_in_list(
(16)                [-1, 1, 'clockwise', 'counterclockwise', 'anticlockwise'],
(16)                direction=direction)
(8)         self._direction.invalidate()
(4)     def get_theta_direction(self):
(8)         """
(8)         Get the direction in which theta increases.
(8)         -1:
(11)           Theta increases in the clockwise direction
(8)         1:
(11)           Theta increases in the counterclockwise direction
(8)         """
(8)         return self._direction.get_matrix()[0, 0]
(4)     def set_rmax(self, rmax):
(8)         """
(8)         Set the outer radial limit.
(8)         Parameters
(8)         ----------
(8)         rmax : float
(8)         """
(8)         self.viewLim.y1 = rmax
(4)     def get_rmax(self):
(8)         """
(8)         Returns
(8)         -------
(8)         float
```

```
935: (12)                        Outer radial limit.
936: (8)                    """
937: (8)                    return self.viewLim.ymax
938: (4)                def set_rmin(self, rmin):
939: (8)                    """
940: (8)                    Set the inner radial limit.
941: (8)                    Parameters
942: (8)                    ----------
943: (8)                    rmin : float
944: (8)                    """
945: (8)                    self.viewLim.y0 = rmin
946: (4)                def get_rmin(self):
947: (8)                    """
948: (8)                    Returns
949: (8)                    -------
950: (8)                    float
951: (12)                        The inner radial limit.
952: (8)                    """
953: (8)                    return self.viewLim.ymin
954: (4)                def set_rorigin(self, rorigin):
955: (8)                    """
956: (8)                    Update the radial origin.
957: (8)                    Parameters
958: (8)                    ----------
959: (8)                    rorigin : float
960: (8)                    """
961: (8)                    self._originViewLim.locked_y0 = rorigin
962: (4)                def get_rorigin(self):
963: (8)                    """
964: (8)                    Returns
965: (8)                    -------
966: (8)                    float
967: (8)                    """
968: (8)                    return self._originViewLim.y0
969: (4)                def get_rsign(self):
970: (8)                    return np.sign(self._originViewLim.y1 - self._originViewLim.y0)
971: (4)                def set_rlim(self, bottom=None, top=None, *,
972: (17)                            emit=True, auto=False, **kwargs):
973: (8)                    """
974: (8)                    Set the radial axis view limits.
975: (8)                    This function behaves like `.Axes.set_ylim`, but additionally supports
976: (8)                    *rmin* and *rmax* as aliases for *bottom* and *top*.
977: (8)                    See Also
978: (8)                    --------
979: (8)                    .Axes.set_ylim
980: (8)                    """
981: (8)                    if 'rmin' in kwargs:
982: (12)                       if bottom is None:
983: (16)                           bottom = kwargs.pop('rmin')
984: (12)                       else:
985: (16)                           raise ValueError('Cannot supply both positional "bottom"'
986: (33)                                            'argument and kwarg "rmin"')
987: (8)                    if 'rmax' in kwargs:
988: (12)                       if top is None:
989: (16)                           top = kwargs.pop('rmax')
990: (12)                       else:
991: (16)                           raise ValueError('Cannot supply both positional "top"'
992: (33)                                            'argument and kwarg "rmax"')
993: (8)                    return self.set_ylim(bottom=bottom, top=top, emit=emit, auto=auto,
994: (29)                                        **kwargs)
995: (4)                def get_rlabel_position(self):
996: (8)                    """
997: (8)                    Returns
998: (8)                    -------
999: (8)                    float
1000: (12)                      The theta position of the radius labels in degrees.
1001: (8)                   """
1002: (8)                   return np.rad2deg(self._r_label_position.get_matrix()[0, 2])
1003: (4)               def set_rlabel_position(self, value):
```

```
(8)                 """
(8)                 Update the theta position of the radius labels.
(8)                 Parameters
(8)                 ----------
(8)                 value : number
(12)                    The angular position of the radius labels in degrees.
(8)                 """
(8)                 self._r_label_position.clear().translate(np.deg2rad(value), 0.0)
(4)             def set_yscale(self, *args, **kwargs):
(8)                 super().set_yscale(*args, **kwargs)
(8)                 self.yaxis.set_major_locator(
(12)                    self.RadialLocator(self.yaxis.get_major_locator(), self))
(4)             def set_rscale(self, *args, **kwargs):
(8)                 return Axes.set_yscale(self, *args, **kwargs)
(4)             def set_rticks(self, *args, **kwargs):
(8)                 return Axes.set_yticks(self, *args, **kwargs)
(4)             def set_thetagrids(self, angles, labels=None, fmt=None, **kwargs):
(8)                 """
(8)                 Set the theta gridlines in a polar plot.
(8)                 Parameters
(8)                 ----------
(8)                 angles : tuple with floats, degrees
(12)                    The angles of the theta gridlines.
(8)                 labels : tuple with strings or None
(12)                    The labels to use at each theta gridline. The
(12)                    `.projections.polar.ThetaFormatter` will be used if None.
(8)                 fmt : str or None
(12)                    Format string used in `matplotlib.ticker.FormatStrFormatter`.
(12)                    For example '%f'. Note that the angle that is used is in
(12)                    radians.
(8)                 Returns
(8)                 -------
(8)                 lines : list of `.lines.Line2D`
(12)                    The theta gridlines.
(8)                 labels : list of `.text.Text`
(12)                    The tick labels.
(8)                 Other Parameters
(8)                 ----------------
(8)                 **kwargs
(12)                    *kwargs* are optional `.Text` properties for the labels.
(12)                    .. warning::
(16)                        This only sets the properties of the current ticks.
(16)                        Ticks are not guaranteed to be persistent. Various operations
(16)                        can create, delete and modify the Tick instances. There is an
(16)                        imminent risk that these settings can get lost if you work on
(16)                        the figure further (including also panning/zooming on a
(16)                        displayed figure).
(16)                        Use `.set_tick_params` instead if possible.
(8)                 See Also
(8)                 --------
(8)                 .PolarAxes.set_rgrids
(8)                 .Axis.get_gridlines
(8)                 .Axis.get_ticklabels
(8)                 """
(8)                 angles = self.convert_yunits(angles)
(8)                 angles = np.deg2rad(angles)
(8)                 self.set_xticks(angles)
(8)                 if labels is not None:
(12)                    self.set_xticklabels(labels)
(8)                 elif fmt is not None:
(12)                    self.xaxis.set_major_formatter(mticker.FormatStrFormatter(fmt))
(8)                 for t in self.xaxis.get_ticklabels():
(12)                    t._internal_update(kwargs)
(8)                 return self.xaxis.get_ticklines(), self.xaxis.get_ticklabels()
(4)             def set_rgrids(self, radii, labels=None, angle=None, fmt=None, **kwargs):
(8)                 """
(8)                 Set the radial gridlines on a polar plot.
(8)                 Parameters
(8)                 ----------
```

```
(8)                 radii : tuple with floats
(12)                    The radii for the radial gridlines
(8)                 labels : tuple with strings or None
(12)                    The labels to use at each radial gridline. The
(12)                    `matplotlib.ticker.ScalarFormatter` will be used if None.
(8)                 angle : float
(12)                    The angular position of the radius labels in degrees.
(8)                 fmt : str or None
(12)                    Format string used in `matplotlib.ticker.FormatStrFormatter`.
(12)                    For example '%f'.
(8)                 Returns
(8)                 -------
(8)                 lines : list of `.lines.Line2D`
(12)                    The radial gridlines.
(8)                 labels : list of `.text.Text`
(12)                    The tick labels.
(8)                 Other Parameters
(8)                 ----------------
(8)                 **kwargs
(12)                    *kwargs* are optional `.Text` properties for the labels.
(12)                    .. warning::
(16)                        This only sets the properties of the current ticks.
(16)                        Ticks are not guaranteed to be persistent. Various operations
(16)                        can create, delete and modify the Tick instances. There is an
(16)                        imminent risk that these settings can get lost if you work on
(16)                        the figure further (including also panning/zooming on a
(16)                        displayed figure).
(16)                        Use `.set_tick_params` instead if possible.
(8)                 See Also
(8)                 --------
(8)                 .PolarAxes.set_thetagrids
(8)                 .Axis.get_gridlines
(8)                 .Axis.get_ticklabels
(8)                 """
(8)                 radii = self.convert_xunits(radii)
(8)                 radii = np.asarray(radii)
(8)                 self.set_yticks(radii)
(8)                 if labels is not None:
(12)                    self.set_yticklabels(labels)
(8)                 elif fmt is not None:
(12)                    self.yaxis.set_major_formatter(mticker.FormatStrFormatter(fmt))
(8)                 if angle is None:
(12)                    angle = self.get_rlabel_position()
(8)                 self.set_rlabel_position(angle)
(8)                 for t in self.yaxis.get_ticklabels():
(12)                    t._internal_update(kwargs)
(8)                 return self.yaxis.get_gridlines(), self.yaxis.get_ticklabels()
(4)             def format_coord(self, theta, r):
(8)                 screen_xy = self.transData.transform((theta, r))
(8)                 screen_xys = screen_xy + np.stack(
(12)                    np.meshgrid([-1, 0, 1], [-1, 0, 1])).reshape((2, -1)).T
(8)                 ts, rs = self.transData.inverted().transform(screen_xys).T
(8)                 delta_t = abs((ts - theta + np.pi) % (2 * np.pi) - np.pi).max()
(8)                 delta_t_halfturns = delta_t / np.pi
(8)                 delta_t_degrees = delta_t_halfturns * 180
(8)                 delta_r = abs(rs - r).max()
(8)                 if theta < 0:
(12)                    theta += 2 * np.pi
(8)                 theta_halfturns = theta / np.pi
(8)                 theta_degrees = theta_halfturns * 180
(8)                 def format_sig(value, delta, opt, fmt):
(12)                    prec = (max(0, -math.floor(math.log10(delta))) if fmt == "f" else
(20)                            cbook._g_sig_digits(value, delta))
(12)                    return f"{value:-{opt}.{prec}{fmt}}"
(8)                 return ('\N{GREEK SMALL LETTER THETA}={}\N{GREEK SMALL LETTER PI} '
(16)                        '({}\N{DEGREE SIGN}), r={}').format(
(20)                            format_sig(theta_halfturns, delta_t_halfturns, "", "f"),
(20)                            format_sig(theta_degrees, delta_t_degrees, "", "f"),
(20)                            format_sig(r, delta_r, "#", "g"),
```

```
(16)                            )
(4)                 def get_data_ratio(self):
(8)                     """
(8)                     Return the aspect ratio of the data itself.  For a polar plot,
(8)                     this should always be 1.0
(8)                     """
(8)                     return 1.0
(4)                 def can_zoom(self):
(8)                     """
(8)                     Return whether this Axes supports the zoom box button functionality.
(8)                     A polar Axes does not support zoom boxes.
(8)                     """
(8)                     return False
(4)                 def can_pan(self):
(8)                     """
(8)                     Return whether this Axes supports the pan/zoom button functionality.
(8)                     For a polar Axes, this is slightly misleading. Both panning and
(8)                     zooming are performed by the same button. Panning is performed
(8)                     in azimuth while zooming is done along the radial.
(8)                     """
(8)                     return True
(4)                 def start_pan(self, x, y, button):
(8)                     angle = np.deg2rad(self.get_rlabel_position())
(8)                     mode = ''
(8)                     if button == 1:
(12)                        epsilon = np.pi / 45.0
(12)                        t, r = self.transData.inverted().transform((x, y))
(12)                        if angle - epsilon <= t <= angle + epsilon:
(16)                            mode = 'drag_r_labels'
(8)                     elif button == 3:
(12)                        mode = 'zoom'
(8)                     self._pan_start = types.SimpleNamespace(
(12)                        rmax=self.get_rmax(),
(12)                        trans=self.transData.frozen(),
(12)                        trans_inverse=self.transData.inverted().frozen(),
(12)                        r_label_angle=self.get_rlabel_position(),
(12)                        x=x,
(12)                        y=y,
(12)                        mode=mode)
(4)                 def end_pan(self):
(8)                     del self._pan_start
(4)                 def drag_pan(self, button, key, x, y):
(8)                     p = self._pan_start
(8)                     if p.mode == 'drag_r_labels':
(12)                        (startt, startr), (t, r) = p.trans_inverse.transform(
(16)                            [(p.x, p.y), (x, y)])
(12)                        dt = np.rad2deg(startt - t)
(12)                        self.set_rlabel_position(p.r_label_angle - dt)
(12)                        trans, vert1, horiz1 = self.get_yaxis_text1_transform(0.0)
(12)                        trans, vert2, horiz2 = self.get_yaxis_text2_transform(0.0)
(12)                        for t in self.yaxis.majorTicks + self.yaxis.minorTicks:
(16)                            t.label1.set_va(vert1)
(16)                            t.label1.set_ha(horiz1)
(16)                            t.label2.set_va(vert2)
(16)                            t.label2.set_ha(horiz2)
(8)                     elif p.mode == 'zoom':
(12)                        (startt, startr), (t, r) = p.trans_inverse.transform(
(16)                            [(p.x, p.y), (x, y)])
(12)                        scale = r / startr
(12)                        self.set_rmax(p.rmax / scale)
(0)             PolarAxes.PolarTransform = PolarTransform
(0)             PolarAxes.PolarAffine = PolarAffine
(0)             PolarAxes.InvertedPolarTransform = InvertedPolarTransform
(0)             PolarAxes.ThetaFormatter = ThetaFormatter
(0)             PolarAxes.RadialLocator = RadialLocator
(0)             PolarAxes.ThetaLocator = ThetaLocator
```

File 119 - __init__.py:

1: (0)

File 120 - figmpl_directive.py:

```
1: (0)              """
2: (0)              Add a ``figure-mpl`` directive that is a responsive version of ``figure``.
3: (0)              This implementation is very similar to ``.. figure::``, except it also allows
a
4: (0)              ``srcset=`` argument to be passed to the image tag, hence allowing responsive
5: (0)              resolution images.
6: (0)              There is no particular reason this could not be used standalone, but is meant
7: (0)              to be used with :doc:`/api/sphinxext_plot_directive_api`.
8: (0)              Note that the directory organization is a bit different than ``.. figure::``.
9: (0)              See the *FigureMpl* documentation below.
10: (0)             """
11: (0)             from docutils import nodes
12: (0)             from docutils.parsers.rst import directives
13: (0)             from docutils.parsers.rst.directives.images import Figure, Image
14: (0)             import os
15: (0)             from os.path import relpath
16: (0)             from pathlib import PurePath, Path
17: (0)             import shutil
18: (0)             from sphinx.errors import ExtensionError
19: (0)             import matplotlib
20: (0)             class figmplnode(nodes.General, nodes.Element):
21: (4)                 pass
22: (0)             class FigureMpl(Figure):
23: (4)                 """
24: (4)                 Implements a directive to allow an optional hidpi image.
25: (4)                 Meant to be used with the *plot_srcset* configuration option in conf.py,
26: (4)                 and gets set in the TEMPLATE of plot_directive.py
27: (4)                 e.g.::
28: (8)                     .. figure-mpl:: plot_directive/some_plots-1.png
29: (12)                        :alt: bar
30: (12)                        :srcset: plot_directive/some_plots-1.png,
31: (21)                                 plot_directive/some_plots-1.2x.png 2.00x
32: (12)                        :class: plot-directive
33: (4)                 The resulting html (at ``some_plots.html``) is::
34: (8)                     <img src="sphx_glr_bar_001_hidpi.png"
35: (12)                        srcset="_images/some_plot-1.png,
36: (20)                                _images/some_plots-1.2x.png 2.00x",
37: (12)                        alt="bar"
38: (12)                        class="plot_directive" />
39: (4)                 Note that the handling of subdirectories is different than that used by
the sphinx
40: (4)                 figure directive::
41: (8)                     .. figure-mpl:: plot_directive/nestedpage/index-1.png
42: (12)                        :alt: bar
43: (12)                        :srcset: plot_directive/nestedpage/index-1.png
44: (21)                                 plot_directive/nestedpage/index-1.2x.png 2.00x
45: (12)                        :class: plot_directive
46: (4)                 The resulting html (at ``nestedpage/index.html``)::
47: (8)                     <img src="../_images/nestedpage-index-1.png"
48: (12)                        srcset="../_images/nestedpage-index-1.png,
49: (20)                                ../_images/_images/nestedpage-index-1.2x.png 2.00x",
50: (12)                        alt="bar"
51: (12)                        class="sphx-glr-single-img" />
52: (4)                 where the subdirectory is included in the image name for uniqueness.
53: (4)                 """
54: (4)                 has_content = False
55: (4)                 required_arguments = 1
56: (4)                 optional_arguments = 2
57: (4)                 final_argument_whitespace = False
58: (4)                 option_spec = {
59: (8)                     'alt': directives.unchanged,
```

```
60: (8)                     'height': directives.length_or_unitless,
61: (8)                     'width': directives.length_or_percentage_or_unitless,
62: (8)                     'scale': directives.nonnegative_int,
63: (8)                     'align': Image.align,
64: (8)                     'class': directives.class_option,
65: (8)                     'caption': directives.unchanged,
66: (8)                     'srcset': directives.unchanged,
67: (4)                 }
68: (4)                 def run(self):
69: (8)                     image_node = figmplnode()
70: (8)                     imagenm = self.arguments[0]
71: (8)                     image_node['alt'] = self.options.get('alt', '')
72: (8)                     image_node['align'] = self.options.get('align', None)
73: (8)                     image_node['class'] = self.options.get('class', None)
74: (8)                     image_node['width'] = self.options.get('width', None)
75: (8)                     image_node['height'] = self.options.get('height', None)
76: (8)                     image_node['scale'] = self.options.get('scale', None)
77: (8)                     image_node['caption'] = self.options.get('caption', None)
78: (8)                     image_node['uri'] = imagenm
79: (8)                     image_node['srcset'] = self.options.get('srcset', None)
80: (8)                     return [image_node]
81: (0)             def _parse_srcsetNodes(st):
82: (4)                 """
83: (4)                 parse srcset...
84: (4)                 """
85: (4)                 entries = st.split(',')
86: (4)                 srcset = {}
87: (4)                 for entry in entries:
88: (8)                     spl = entry.strip().split(' ')
89: (8)                     if len(spl) == 1:
90: (12)                        srcset[0] = spl[0]
91: (8)                     elif len(spl) == 2:
92: (12)                        mult = spl[1][:-1]
93: (12)                        srcset[float(mult)] = spl[0]
94: (8)                     else:
95: (12)                        raise ExtensionError(f'srcset argument "{entry}" is invalid.')
96: (4)                 return srcset
97: (0)             def _copy_images_figmpl(self, node):
98: (4)                 if node['srcset']:
99: (8)                     srcset = _parse_srcsetNodes(node['srcset'])
100: (4)                else:
101: (8)                    srcset = None
102: (4)                docsource = PurePath(self.document['source']).parent
103: (4)                srctop = self.builder.srcdir
104: (4)                rel = relpath(docsource, srctop).replace('.', '').replace(os.sep, '-')
105: (4)                if len(rel):
106: (8)                    rel += '-'
107: (4)                imagedir = PurePath(self.builder.outdir, self.builder.imagedir)
108: (4)                Path(imagedir).mkdir(parents=True, exist_ok=True)
109: (4)                if srcset:
110: (8)                    for src in srcset.values():
111: (12)                       abspath = PurePath(docsource, src)
112: (12)                       name = rel + abspath.name
113: (12)                       shutil.copyfile(abspath, imagedir / name)
114: (4)                else:
115: (8)                    abspath = PurePath(docsource, node['uri'])
116: (8)                    name = rel + abspath.name
117: (8)                    shutil.copyfile(abspath, imagedir / name)
118: (4)                return imagedir, srcset, rel
119: (0)            def visit_figmpl_html(self, node):
120: (4)                imagedir, srcset, rel = _copy_images_figmpl(self, node)
121: (4)                docsource = PurePath(self.document['source'])
122: (4)                srctop = PurePath(self.builder.srcdir, '')
123: (4)                relsource = relpath(docsource, srctop)
124: (4)                desttop = PurePath(self.builder.outdir, '')
125: (4)                dest = desttop / relsource
126: (4)                imagerel = PurePath(relpath(imagedir, dest.parent)).as_posix()
127: (4)                if self.builder.name == "dirhtml":
128: (8)                    imagerel = f'..{imagerel}'
```

```
129: (4)                nm = PurePath(node['uri'][1:]).name
130: (4)                uri = f'{imagerel}/{rel}{nm}'
131: (4)                maxsrc = uri
132: (4)                srcsetst = ''
133: (4)                if srcset:
134: (8)                    maxmult = -1
135: (8)                    for mult, src in srcset.items():
136: (12)                       nm = PurePath(src[1:]).name
137: (12)                       path = f'{imagerel}/{rel}{nm}'
138: (12)                       srcsetst += path
139: (12)                       if mult == 0:
140: (16)                           srcsetst += ', '
141: (12)                       else:
142: (16)                           srcsetst += f' {mult:1.2f}x, '
143: (12)                       if mult > maxmult:
144: (16)                           maxmult = mult
145: (16)                           maxsrc = path
146: (8)                    srcsetst = srcsetst[:-2]
147: (4)                alt = node['alt']
148: (4)                if node['class'] is not None:
149: (8)                    classst = ' '.join(node['class'])
150: (8)                    classst = f'class="{classst}"'
151: (4)                else:
152: (8)                    classst = ''
153: (4)                stylers = ['width', 'height', 'scale']
154: (4)                stylest = ''
155: (4)                for style in stylers:
156: (8)                    if node[style]:
157: (12)                       stylest += f'{style}: {node[style]};'
158: (4)                figalign = node['align'] if node['align'] else 'center'
159: (4)                img_block = (f'<img src="{uri}" style="{stylest}" srcset="{srcsetst}" '
160: (17)                            f'alt="{alt}" {classst}/>')
161: (4)                html_block = f'<figure class="align-{figalign}">\n'
162: (4)                html_block += f'  <a class="reference internal image-reference" href="
{maxsrc}">\n'
163: (4)                html_block += f'    {img_block}\n  </a>\n'
164: (4)                if node['caption']:
165: (8)                    html_block += '  <figcaption>\n'
166: (8)                    html_block += f'   <p><span class="caption-text">{node["caption"]}
</span></p>\n'
167: (8)                    html_block += '  </figcaption>\n'
168: (4)                html_block += '</figure>\n'
169: (4)                self.body.append(html_block)
170: (0)            def visit_figmpl_latex(self, node):
171: (4)                if node['srcset'] is not None:
172: (8)                    imagedir, srcset = _copy_images_figmpl(self, node)
173: (8)                    maxmult = -1
174: (8)                    maxmult = max(srcset, default=-1)
175: (8)                    node['uri'] = PurePath(srcset[maxmult]).name
176: (4)                self.visit_figure(node)
177: (0)            def depart_figmpl_html(self, node):
178: (4)                pass
179: (0)            def depart_figmpl_latex(self, node):
180: (4)                self.depart_figure(node)
181: (0)            def figurempl_addnode(app):
182: (4)                app.add_node(figmplnode,
183: (17)                            html=(visit_figmpl_html, depart_figmpl_html),
184: (17)                            latex=(visit_figmpl_latex, depart_figmpl_latex))
185: (0)            def setup(app):
186: (4)                app.add_directive("figure-mpl", FigureMpl)
187: (4)                figurempl_addnode(app)
188: (4)                metadata = {'parallel_read_safe': True, 'parallel_write_safe': True,
189: (16)                           'version': matplotlib.__version__}
190: (4)                return metadata

----------------------------------------

File 121 - mathmpl.py:
```

```
1: (0)              r"""
2: (0)              A role and directive to display mathtext in Sphinx
3: (0)              ==================================================
4: (0)              The ``mathmpl`` Sphinx extension creates a mathtext image in Matplotlib and
5: (0)              shows it in html output. Thus, it is a true and faithful representation of
what
6: (0)              you will see if you pass a given LaTeX string to Matplotlib (see
7: (0)              :ref:`mathtext`).
8: (0)              .. warning::
9: (4)                  In most cases, you will likely want to use one of `Sphinx's builtin Math
10: (4)                 extensions
11: (4)                 <https://www.sphinx-doc.org/en/master/usage/extensions/math.html>`__
12: (4)                 instead of this one. The builtin Sphinx math directive uses MathJax to
13: (4)                 render mathematical expressions, and addresses accessibility concerns that
14: (4)                 ``mathmpl`` doesn't address.
15: (0)             Mathtext may be included in two ways:
16: (0)             1. Inline, using the role::
17: (5)                  This text uses inline math: :mathmpl:`\alpha > \beta`.
18: (3)                which produces:
19: (5)                  This text uses inline math: :mathmpl:`\alpha > \beta`.
20: (0)             2. Standalone, using the directive::
21: (5)                  Here is some standalone math:
22: (5)                  .. mathmpl::
23: (9)                      \alpha > \beta
24: (3)                which produces:
25: (5)                  Here is some standalone math:
26: (5)                  .. mathmpl::
27: (9)                      \alpha > \beta
28: (0)             Options
29: (0)             -------
30: (0)             The ``mathmpl`` role and directive both support the following options:
31: (0)             fontset : str, default: 'cm'
32: (4)                 The font set to use when displaying math. See :rc:`mathtext.fontset`.
33: (0)             fontsize : float
34: (4)                 The font size, in points. Defaults to the value from the extension
35: (4)                 configuration option defined below.
36: (0)             Configuration options
37: (0)             ---------------------
38: (0)             The mathtext extension has the following configuration options:
39: (0)             mathmpl_fontsize : float, default: 10.0
40: (4)                 Default font size, in points.
41: (0)             mathmpl_srcset : list of str, default: []
42: (4)                 Additional image sizes to generate when embedding in HTML, to support
43: (4)                 `responsive resolution images
44: (4)                 <https://developer.mozilla.org/en-
US/docs/Learn/HTML/Multimedia_and_embedding/Responsive_images>`__.
45: (4)                 The list should contain additional x-descriptors (``'1.5x'``, ``'2x'``,
46: (4)                 etc.) to generate (1x is the default and always included.)
47: (0)             """
48: (0)             import hashlib
49: (0)             from pathlib import Path
50: (0)             from docutils import nodes
51: (0)             from docutils.parsers.rst import Directive, directives
52: (0)             import sphinx
53: (0)             from sphinx.errors import ConfigError, ExtensionError
54: (0)             import matplotlib as mpl
55: (0)             from matplotlib import _api, mathtext
56: (0)             from matplotlib.rcsetup import validate_float_or_None
57: (0)             class latex_math(nodes.General, nodes.Element):
58: (4)                 pass
59: (0)             def fontset_choice(arg):
60: (4)                 return directives.choice(arg, mathtext.MathTextParser._font_type_mapping)
61: (0)             def math_role(role, rawtext, text, lineno, inliner,
62: (14)                          options={}, content=[]):
63: (4)                 i = rawtext.find('`')
64: (4)                 latex = rawtext[i+1:-1]
65: (4)                 node = latex_math(rawtext)
66: (4)                 node['latex'] = latex
67: (4)                 node['fontset'] = options.get('fontset', 'cm')
```

```
 68: (4)                 node['fontsize'] = options.get('fontsize',
 69: (35)                                                setup.app.config.mathmpl_fontsize)
 70: (4)                 return [node], []
 71: (0)             math_role.options = {'fontset': fontset_choice,
 72: (21)                                 'fontsize': validate_float_or_None}
 73: (0)             class MathDirective(Directive):
 74: (4)                 """
 75: (4)                 The ``.. mathmpl::`` directive, as documented in the module's docstring.
 76: (4)                 """
 77: (4)                 has_content = True
 78: (4)                 required_arguments = 0
 79: (4)                 optional_arguments = 0
 80: (4)                 final_argument_whitespace = False
 81: (4)                 option_spec = {'fontset': fontset_choice,
 82: (19)                               'fontsize': validate_float_or_None}
 83: (4)                 def run(self):
 84: (8)                     latex = ''.join(self.content)
 85: (8)                     node = latex_math(self.block_text)
 86: (8)                     node['latex'] = latex
 87: (8)                     node['fontset'] = self.options.get('fontset', 'cm')
 88: (8)                     node['fontsize'] = self.options.get('fontsize',
 89: (44)                                                    setup.app.config.mathmpl_fontsize)
 90: (8)                     return [node]
 91: (0)             def latex2png(latex, filename, fontset='cm', fontsize=10, dpi=100):
 92: (4)                 with mpl.rc_context({'mathtext.fontset': fontset, 'font.size': fontsize}):
 93: (8)                     try:
 94: (12)                        depth = mathtext.math_to_image(
 95: (16)                            f"${latex}$", filename, dpi=dpi, format="png")
 96: (8)                     except Exception:
 97: (12)                        _api.warn_external(f"Could not render math expression {latex}")
 98: (12)                        depth = 0
 99: (4)                 return depth
100: (0)             def latex2html(node, source):
101: (4)                 inline = isinstance(node.parent, nodes.TextElement)
102: (4)                 latex = node['latex']
103: (4)                 fontset = node['fontset']
104: (4)                 fontsize = node['fontsize']
105: (4)                 name = 'math-{}'.format(
106: (8)                     hashlib.md5(f'{latex}{fontset}{fontsize}'.encode()).hexdigest()[-10:])
107: (4)                 destdir = Path(setup.app.builder.outdir, '_images', 'mathmpl')
108: (4)                 destdir.mkdir(parents=True, exist_ok=True)
109: (4)                 dest = destdir / f'{name}.png'
110: (4)                 depth = latex2png(latex, dest, fontset, fontsize=fontsize)
111: (4)                 srcset = []
112: (4)                 for size in setup.app.config.mathmpl_srcset:
113: (8)                     filename = f'{name}-{size.replace(".", "_")}.png'
114: (8)                     latex2png(latex, destdir / filename, fontset, fontsize=fontsize,
115: (18)                              dpi=100 * float(size[:-1]))
116: (8)                     srcset.append(
117: (12)                        f'{setup.app.builder.imgpath}/mathmpl/{filename} {size}')
118: (4)                 if srcset:
119: (8)                     srcset = (f'srcset="{setup.app.builder.imgpath}/mathmpl/{name}.png, '
+
120: (18)                              ', '.join(srcset) + '" ')
121: (4)                 if inline:
122: (8)                     cls = ''
123: (4)                 else:
124: (8)                     cls = 'class="center" '
125: (4)                 if inline and depth != 0:
126: (8)                     style = 'style="position: relative; bottom: -%dpx"' % (depth + 1)
127: (4)                 else:
128: (8)                     style = ''
129: (4)                 return (f'<img src="{setup.app.builder.imgpath}/mathmpl/{name}.png"'
130: (12)                        f' {srcset}{cls}{style}/>')
131: (0)             def _config_inited(app, config):
132: (4)                 for i, size in enumerate(app.config.mathmpl_srcset):
133: (8)                     if size[-1] == 'x':  # "2x" = "2.0"
134: (12)                        try:
135: (16)                           float(size[:-1])
```

```
136: (12)                        except ValueError:
137: (16)                            raise ConfigError(
138: (20)                                f'Invalid value for mathmpl_srcset parameter: {size!r}. '
139: (20)                                'Must be a list of strings with the multiplicative '
140: (20)                                'factor followed by an "x".  e.g. ["2.0x", "1.5x"]')
141: (8)                    else:
142: (12)                        raise ConfigError(
143: (16)                            f'Invalid value for mathmpl_srcset parameter: {size!r}. '
144: (16)                            'Must be a list of strings with the multiplicative '
145: (16)                            'factor followed by an "x".  e.g. ["2.0x", "1.5x"]')
146: (0)            def setup(app):
147: (4)                setup.app = app
148: (4)                app.add_config_value('mathmpl_fontsize', 10.0, True)
149: (4)                app.add_config_value('mathmpl_srcset', [], True)
150: (4)                try:
151: (8)                    app.connect('config-inited', _config_inited)  # Sphinx 1.8+
152: (4)                except ExtensionError:
153: (8)                    app.connect('env-updated', lambda app, env: _config_inited(app, None))
154: (4)                def visit_latex_math_html(self, node):
155: (8)                    source = self.document.attributes['source']
156: (8)                    self.body.append(latex2html(node, source))
157: (4)                def depart_latex_math_html(self, node):
158: (8)                    pass
159: (4)                def visit_latex_math_latex(self, node):
160: (8)                    inline = isinstance(node.parent, nodes.TextElement)
161: (8)                    if inline:
162: (12)                       self.body.append('$%s$' % node['latex'])
163: (8)                    else:
164: (12)                       self.body.extend(['\\begin{equation}',
165: (30)                                         node['latex'],
166: (30)                                         '\\end{equation}'])
167: (4)                def depart_latex_math_latex(self, node):
168: (8)                    pass
169: (4)                app.add_node(latex_math,
170: (17)                            html=(visit_latex_math_html, depart_latex_math_html),
171: (17)                            latex=(visit_latex_math_latex, depart_latex_math_latex))
172: (4)                app.add_role('mathmpl', math_role)
173: (4)                app.add_directive('mathmpl', MathDirective)
174: (4)                if sphinx.version_info < (1, 8):
175: (8)                    app.add_role('math', math_role)
176: (8)                    app.add_directive('math', MathDirective)
177: (4)                metadata = {'parallel_read_safe': True, 'parallel_write_safe': True}
178: (4)                return metadata
```

File 122 - plot_directive.py:

```
1: (0)              """
2: (0)              A directive for including a Matplotlib plot in a Sphinx document
3: (0)              ================================================================
4: (0)              This is a Sphinx extension providing a reStructuredText directive
5: (0)              ``.. plot::`` for including a plot in a Sphinx document.
6: (0)              In HTML output, ``.. plot::`` will include a .png file with a link
7: (0)              to a high-res .png and .pdf.  In LaTeX output, it will include a .pdf.
8: (0)              The plot content may be defined in one of three ways:
9: (0)              1. **A path to a source file** as the argument to the directive::
10: (5)                  .. plot:: path/to/plot.py
11: (3)                 When a path to a source file is given, the content of the
12: (3)                 directive may optionally contain a caption for the plot::
13: (5)                  .. plot:: path/to/plot.py
14: (8)                     The plot caption.
15: (3)                 Additionally, one may specify the name of a function to call (with
16: (3)                 no arguments) immediately after importing the module::
17: (5)                  .. plot:: path/to/plot.py plot_function1
18: (0)             2. Included as **inline content** to the directive::
19: (5)                  .. plot::
20: (8)                     import matplotlib.pyplot as plt
21: (8)                     plt.plot([1, 2, 3], [4, 5, 6])
```

```
 22: (8)                    plt.title("A plotting exammple")
 23: (0)             3. Using **doctest** syntax::
 24: (5)                  .. plot::
 25: (8)                    A plotting example:
 26: (8)                    >>> import matplotlib.pyplot as plt
 27: (8)                    >>> plt.plot([1, 2, 3], [4, 5, 6])
 28: (0)             Options
 29: (0)             -------
 30: (0)             The ``.. plot::`` directive supports the following options:
 31: (0)             ``:format:`` : {'python', 'doctest'}
 32: (4)                 The format of the input.  If unset, the format is auto-detected.
 33: (0)             ``:include-source:`` : bool
 34: (4)                 Whether to display the source code. The default can be changed using
 35: (4)                 the ``plot_include_source`` variable in :file:`conf.py` (which itself
 36: (4)                 defaults to False).
 37: (0)             ``:show-source-link:`` : bool
 38: (4)                 Whether to show a link to the source in HTML. The default can be
 39: (4)                 changed using the ``plot_html_show_source_link`` variable in
 40: (4)                 :file:`conf.py` (which itself defaults to True).
 41: (0)             ``:context:`` : bool or str
 42: (4)                 If provided, the code will be run in the context of all previous plot
 43: (4)                 directives for which the ``:context:`` option was specified.  This only
 44: (4)                 applies to inline code plot directives, not those run from files. If
 45: (4)                 the ``:context: reset`` option is specified, the context is reset
 46: (4)                 for this and future plots, and previous figures are closed prior to
 47: (4)                 running the code. ``:context: close-figs`` keeps the context but closes
 48: (4)                 previous figures before running the code.
 49: (0)             ``:nofigs:`` : bool
 50: (4)                 If specified, the code block will be run, but no figures will be
 51: (4)                 inserted.  This is usually useful with the ``:context:`` option.
 52: (0)             ``:caption:`` : str
 53: (4)                 If specified, the option's argument will be used as a caption for the
 54: (4)                 figure. This overwrites the caption given in the content, when the plot
 55: (4)                 is generated from a file.
 56: (0)             Additionally, this directive supports all the options of the `image directive
 57: (0)             <https://docutils.sourceforge.io/docs/ref/rst/directives.html#image>`_,
 58: (0)             except for ``:target:`` (since plot will add its own target).  These include
 59: (0)             ``:alt:``, ``:height:``, ``:width:``, ``:scale:``, ``:align:`` and
``:class:``.
 60: (0)             Configuration options
 61: (0)             ---------------------
 62: (0)             The plot directive has the following configuration options:
 63: (0)             plot_include_source
 64: (4)                 Default value for the include-source option (default: False).
 65: (0)             plot_html_show_source_link
 66: (4)                 Whether to show a link to the source in HTML (default: True).
 67: (0)             plot_pre_code
 68: (4)                 Code that should be executed before each plot. If None (the default),
 69: (4)                 it will default to a string containing::
 70: (8)                     import numpy as np
 71: (8)                     from matplotlib import pyplot as plt
 72: (0)             plot_basedir
 73: (4)                 Base directory, to which ``plot::`` file names are relative to.
 74: (4)                 If None or empty (the default), file names are relative to the
 75: (4)                 directory where the file containing the directive is.
 76: (0)             plot_formats
 77: (4)                 File formats to generate (default: ['png', 'hires.png', 'pdf']).
 78: (4)                 List of tuples or strings::
 79: (8)                     [(suffix, dpi), suffix, ...]
 80: (4)                 that determine the file format and the DPI. For entries whose
 81: (4)                 DPI was omitted, sensible defaults are chosen. When passing from
 82: (4)                 the command line through sphinx_build the list should be passed as
 83: (4)                 suffix:dpi,suffix:dpi, ...
 84: (0)             plot_html_show_formats
 85: (4)                 Whether to show links to the files in HTML (default: True).
 86: (0)             plot_rcparams
 87: (4)                 A dictionary containing any non-standard rcParams that should
 88: (4)                 be applied before each plot (default: {}).
 89: (0)             plot_apply_rcparams
```

```
90: (4)                 By default, rcParams are applied when ``:context:`` option is not used
91: (4)                 in a plot directive.  If set, this configuration option overrides this
92: (4)                 behavior and applies rcParams before each plot.
93: (0)             plot_working_directory
94: (4)                 By default, the working directory will be changed to the directory of
95: (4)                 the example, so the code can get at its data files, if any.  Also its
96: (4)                 path will be added to `sys.path` so it can import any helper modules
97: (4)                 sitting beside it.  This configuration option can be used to specify
98: (4)                 a central directory (also added to `sys.path`) where data files and
99: (4)                 helper modules for all code are located.
100: (0)            plot_template
101: (4)                Provide a customized template for preparing restructured text.
102: (0)            plot_srcset
103: (4)                Allow the srcset image option for responsive image resolutions. List of
104: (4)                strings with the multiplicative factors followed by an "x".
105: (4)                e.g. ["2.0x", "1.5x"].  "2.0x" will create a png with the default "png"
106: (4)                resolution from plot_formats, multiplied by 2. If plot_srcset is
107: (4)                specified, the plot directive uses the
108: (4)                :doc:`/api/sphinxext_figmpl_directive_api` (instead of the usual figure
109: (4)                directive) in the intermediary rst file that is generated.
110: (4)                The plot_srcset option is incompatible with *singlehtml* builds, and an
111: (4)                error will be raised.
112: (0)            Notes on how it works
113: (0)            ---------------------
114: (0)            The plot directive runs the code it is given, either in the source file or the
115: (0)            code under the directive. The figure created (if any) is saved in the sphinx
116: (0)            build directory under a subdirectory named ``plot_directive``.  It then
creates
117: (0)            an intermediate rst file that calls a ``.. figure:`` directive (or
118: (0)            ``.. figmpl::`` directive if ``plot_srcset`` is being used) and has links to
119: (0)            the ``*.png`` files in the ``plot_directive`` directory.  These translations
can
120: (0)            be customized by changing the *plot_template*.  See the source of
121: (0)            :doc:`/api/sphinxext_plot_directive_api` for the templates defined in
*TEMPLATE*
122: (0)            and *TEMPLATE_SRCSET*.
123: (0)            """
124: (0)            import contextlib
125: (0)            import doctest
126: (0)            from io import StringIO
127: (0)            import itertools
128: (0)            import os
129: (0)            from os.path import relpath
130: (0)            from pathlib import Path
131: (0)            import re
132: (0)            import shutil
133: (0)            import sys
134: (0)            import textwrap
135: (0)            import traceback
136: (0)            from docutils.parsers.rst import directives, Directive
137: (0)            from docutils.parsers.rst.directives.images import Image
138: (0)            import jinja2  # Sphinx dependency.
139: (0)            from sphinx.errors import ExtensionError
140: (0)            import matplotlib
141: (0)            from matplotlib.backend_bases import FigureManagerBase
142: (0)            import matplotlib.pyplot as plt
143: (0)            from matplotlib import _pylab_helpers, cbook
144: (0)            matplotlib.use("agg")
145: (0)            __version__ = 2
146: (0)            def _option_boolean(arg):
147: (4)                if not arg or not arg.strip():
148: (8)                    return True
149: (4)                elif arg.strip().lower() in ('no', '0', 'false'):
150: (8)                    return False
151: (4)                elif arg.strip().lower() in ('yes', '1', 'true'):
152: (8)                    return True
153: (4)                else:
154: (8)                    raise ValueError(f'{arg!r} unknown boolean')
155: (0)            def _option_context(arg):
```

```
(4)                 if arg in [None, 'reset', 'close-figs']:
(8)                     return arg
(4)                 raise ValueError("Argument should be None or 'reset' or 'close-figs'")
(0)             def _option_format(arg):
(4)                 return directives.choice(arg, ('python', 'doctest'))
(0)             def mark_plot_labels(app, document):
(4)                 """
(4)                 To make plots referenceable, we need to move the reference from the
(4)                 "htmlonly" (or "latexonly") node to the actual figure node itself.
(4)                 """
(4)                 for name, explicit in document.nametypes.items():
(8)                     if not explicit:
(12)                        continue
(8)                     labelid = document.nameids[name]
(8)                     if labelid is None:
(12)                        continue
(8)                     node = document.ids[labelid]
(8)                     if node.tagname in ('html_only', 'latex_only'):
(12)                        for n in node:
(16)                            if n.tagname == 'figure':
(20)                                sectname = name
(20)                                for c in n:
(24)                                    if c.tagname == 'caption':
(28)                                        sectname = c.astext()
(28)                                        break
(20)                                node['ids'].remove(labelid)
(20)                                node['names'].remove(name)
(20)                                n['ids'].append(labelid)
(20)                                n['names'].append(name)
(20)                                document.settings.env.labels[name] = \
(24)                                    document.settings.env.docname, labelid, sectname
(20)                                break
(0)             class PlotDirective(Directive):
(4)                 """The ``.. plot::`` directive, as documented in the module's
docstring."""
(4)                 has_content = True
(4)                 required_arguments = 0
(4)                 optional_arguments = 2
(4)                 final_argument_whitespace = False
(4)                 option_spec = {
(8)                     'alt': directives.unchanged,
(8)                     'height': directives.length_or_unitless,
(8)                     'width': directives.length_or_percentage_or_unitless,
(8)                     'scale': directives.nonnegative_int,
(8)                     'align': Image.align,
(8)                     'class': directives.class_option,
(8)                     'include-source': _option_boolean,
(8)                     'show-source-link': _option_boolean,
(8)                     'format': _option_format,
(8)                     'context': _option_context,
(8)                     'nofigs': directives.flag,
(8)                     'caption': directives.unchanged,
(8)                     }
(4)                 def run(self):
(8)                     """Run the plot directive."""
(8)                     try:
(12)                        return run(self.arguments, self.content, self.options,
(23)                                   self.state_machine, self.state, self.lineno)
(8)                     except Exception as e:
(12)                        raise self.error(str(e))
(0)             def _copy_css_file(app, exc):
(4)                 if exc is None and app.builder.format == 'html':
(8)                     src = cbook._get_data_path('plot_directive/plot_directive.css')
(8)                     dst = app.outdir / Path('_static')
(8)                     dst.mkdir(exist_ok=True)
(8)                     shutil.copyfile(src, dst / Path('plot_directive.css'))
(0)             def setup(app):
(4)                 setup.app = app
(4)                 setup.config = app.config
```

```
224: (4)                setup.confdir = app.confdir
225: (4)                app.add_directive('plot', PlotDirective)
226: (4)                app.add_config_value('plot_pre_code', None, True)
227: (4)                app.add_config_value('plot_include_source', False, True)
228: (4)                app.add_config_value('plot_html_show_source_link', True, True)
229: (4)                app.add_config_value('plot_formats', ['png', 'hires.png', 'pdf'], True)
230: (4)                app.add_config_value('plot_basedir', None, True)
231: (4)                app.add_config_value('plot_html_show_formats', True, True)
232: (4)                app.add_config_value('plot_rcparams', {}, True)
233: (4)                app.add_config_value('plot_apply_rcparams', False, True)
234: (4)                app.add_config_value('plot_working_directory', None, True)
235: (4)                app.add_config_value('plot_template', None, True)
236: (4)                app.add_config_value('plot_srcset', [], True)
237: (4)                app.connect('doctree-read', mark_plot_labels)
238: (4)                app.add_css_file('plot_directive.css')
239: (4)                app.connect('build-finished', _copy_css_file)
240: (4)                metadata = {'parallel_read_safe': True, 'parallel_write_safe': True,
241: (16)                           'version': matplotlib.__version__}
242: (4)                return metadata
243: (0)            def contains_doctest(text):
244: (4)                try:
245: (8)                    compile(text, '<string>', 'exec')
246: (8)                    return False
247: (4)                except SyntaxError:
248: (8)                    pass
249: (4)                r = re.compile(r'^\s*>>>', re.M)
250: (4)                m = r.search(text)
251: (4)                return bool(m)
252: (0)            def _split_code_at_show(text, function_name):
253: (4)                """Split code at plt.show()."""
254: (4)                is_doctest = contains_doctest(text)
255: (4)                if function_name is None:
256: (8)                    parts = []
257: (8)                    part = []
258: (8)                    for line in text.split("\n"):
259: (12)                       if ((not is_doctest and line.startswith('plt.show(')) or
260: (19)                              (is_doctest and line.strip() == '>>> plt.show()')):
261: (16)                           part.append(line)
262: (16)                           parts.append("\n".join(part))
263: (16)                           part = []
264: (12)                       else:
265: (16)                           part.append(line)
266: (8)                    if "\n".join(part).strip():
267: (12)                       parts.append("\n".join(part))
268: (4)                else:
269: (8)                    parts = [text]
270: (4)                return is_doctest, parts
271: (0)            _SOURCECODE = """
272: (0)            {{ source_code }}
273: (0)            .. only:: html
274: (3)               {% if src_name or (html_show_formats and not multi_image) %}
275: (3)               (
276: (3)               {%- if src_name -%}
277: (3)               :download:`Source code <{{ build_dir }}/{{ src_name }}>`
278: (3)               {%- endif -%}
279: (3)               {%- if html_show_formats and not multi_image -%}
280: (5)                 {%- for img in images -%}
281: (7)                   {%- for fmt in img.formats -%}
282: (9)                     {%- if src_name or not loop.first -%}, {% endif -%}
283: (9)                     :download:`{{ fmt }} <{{ build_dir }}/{{ img.basename }}.{{ fmt }}>`
284: (7)                   {%- endfor -%}
285: (5)                 {%- endfor -%}
286: (3)               {%- endif -%}
287: (3)               )
288: (3)               {% endif %}
289: (0)            """
290: (0)            TEMPLATE_SRCSET = _SOURCECODE + """
291: (3)               {% for img in images %}
292: (3)               .. figure-mpl:: {{ build_dir }}/{{ img.basename }}.{{ default_fmt }}
```

```
293: (6)                 {% for option in options -%}
294: (6)                 {{ option }}
295: (6)                 {% endfor %}
296: (6)                 {%- if caption -%}
297: (6)                 {{ caption }}  {# appropriate leading whitespace added beforehand #}
298: (6)                 {% endif -%}
299: (6)                 {%- if srcset -%}
300: (8)                   :srcset: {{ build_dir }}/{{ img.basename }}.{{ default_fmt }}
301: (8)                   {%- for sr in srcset -%}
302: (12)                      , {{ build_dir }}/{{ img.basename }}.{{ sr }}.{{ default_fmt }}
{{sr}}
303: (8)                   {%- endfor -%}
304: (6)                 {% endif %}
305: (3)              {% if html_show_formats and multi_image %}
306: (3)              (
307: (4)               {%- for fmt in img.formats -%}
308: (4)               {%- if not loop.first -%}, {% endif -%}
309: (4)               :download:`{{ fmt }} <{{ build_dir }}/{{ img.basename }}.{{ fmt }}>`
310: (4)               {%- endfor -%}
311: (3)              )
312: (3)              {% endif %}
313: (3)              {% endfor %}
314: (0)           .. only:: not html
315: (3)              {% for img in images %}
316: (3)              .. figure-mpl:: {{ build_dir }}/{{ img.basename }}.*
317: (6)                 {% for option in options -%}
318: (6)                 {{ option }}
319: (6)                 {% endfor -%}
320: (6)                 {{ caption }}  {# appropriate leading whitespace added beforehand #}
321: (3)              {% endfor %}
322: (0)           """
323: (0)           TEMPLATE = _SOURCECODE + """
324: (3)              {% for img in images %}
325: (3)              .. figure:: {{ build_dir }}/{{ img.basename }}.{{ default_fmt }}
326: (6)                 {% for option in options -%}
327: (6)                 {{ option }}
328: (6)                 {% endfor %}
329: (6)                 {% if html_show_formats and multi_image -%}
330: (8)                   (
331: (8)                   {%- for fmt in img.formats -%}
332: (8)                   {%- if not loop.first -%}, {% endif -%}
333: (8)                   :download:`{{ fmt }} <{{ build_dir }}/{{ img.basename }}.{{ fmt }}>`
334: (8)                   {%- endfor -%}
335: (8)                   )
336: (6)                 {%- endif -%}
337: (6)                 {{ caption }}  {# appropriate leading whitespace added beforehand #}
338: (3)              {% endfor %}
339: (0)           .. only:: not html
340: (3)              {% for img in images %}
341: (3)              .. figure:: {{ build_dir }}/{{ img.basename }}.*
342: (6)                 {% for option in options -%}
343: (6)                 {{ option }}
344: (6)                 {% endfor -%}
345: (6)                 {{ caption }}  {# appropriate leading whitespace added beforehand #}
346: (3)              {% endfor %}
347: (0)           """
348: (0)           exception_template = """
349: (0)           .. only:: html
350: (3)              [`source code <%(linkdir)s/%(basename)s.py>`__]
351: (0)           Exception occurred rendering plot.
352: (0)           """
353: (0)           plot_context = dict()
354: (0)           class ImageFile:
355: (4)               def __init__(self, basename, dirname):
356: (8)                   self.basename = basename
357: (8)                   self.dirname = dirname
358: (8)                   self.formats = []
359: (4)               def filename(self, format):
360: (8)                   return os.path.join(self.dirname, f"{self.basename}.{format}")
```

```
(4)     def filenames(self):
(8)         return [self.filename(fmt) for fmt in self.formats]
(0) def out_of_date(original, derived, includes=None):
(4)     """
(4)     Return whether *derived* is out-of-date relative to *original* or any of
(4)     the RST files included in it using the RST include directive (*includes*).
(4)     *derived* and *original* are full paths, and *includes* is optionally a
(4)     list of full paths which may have been included in the *original*.
(4)     """
(4)     if not os.path.exists(derived):
(8)         return True
(4)     if includes is None:
(8)         includes = []
(4)     files_to_check = [original, *includes]
(4)     def out_of_date_one(original, derived_mtime):
(8)         return (os.path.exists(original) and
(16)                 derived_mtime < os.stat(original).st_mtime)
(4)     derived_mtime = os.stat(derived).st_mtime
(4)     return any(out_of_date_one(f, derived_mtime) for f in files_to_check)
(0) class PlotError(RuntimeError):
(4)     pass
(0) def _run_code(code, code_path, ns=None, function_name=None):
(4)     """
(4)     Import a Python module from a path, and run the function given by
(4)     name, if function_name is not None.
(4)     """
(4)     pwd = os.getcwd()
(4)     if setup.config.plot_working_directory is not None:
(8)         try:
(12)            os.chdir(setup.config.plot_working_directory)
(8)         except OSError as err:
(12)            raise OSError(f'{err}\n`plot_working_directory` option in '
(26)                          f'Sphinx configuration file must be a valid '
(26)                          f'directory path') from err
(8)         except TypeError as err:
(12)            raise TypeError(f'{err}\n`plot_working_directory` option in '
(28)                            f'Sphinx configuration file must be a string or '
(28)                            f'None') from err
(4)     elif code_path is not None:
(8)         dirname = os.path.abspath(os.path.dirname(code_path))
(8)         os.chdir(dirname)
(4)     with cbook._setattr_cm(
(12)            sys, argv=[code_path], path=[os.getcwd(), *sys.path]), \
(12)            contextlib.redirect_stdout(StringIO()):
(8)         try:
(12)            if ns is None:
(16)                ns = {}
(12)            if not ns:
(16)                if setup.config.plot_pre_code is None:
(20)                    exec('import numpy as np\n'
(25)                         'from matplotlib import pyplot as plt\n', ns)
(16)                else:
(20)                    exec(str(setup.config.plot_pre_code), ns)
(12)            if "__main__" in code:
(16)                ns['__name__'] = '__main__'
(12)            with cbook._setattr_cm(FigureManagerBase, show=lambda self: None):
(16)                exec(code, ns)
(16)                if function_name is not None:
(20)                    exec(function_name + "()", ns)
(8)         except (Exception, SystemExit) as err:
(12)            raise PlotError(traceback.format_exc()) from err
(8)         finally:
(12)            os.chdir(pwd)
(4)     return ns
(0) def clear_state(plot_rcparams, close=True):
(4)     if close:
(8)         plt.close('all')
(4)     matplotlib.rc_file_defaults()
(4)     matplotlib.rcParams.update(plot_rcparams)
```

```
(0)             def get_plot_formats(config):
(4)                 default_dpi = {'png': 80, 'hires.png': 200, 'pdf': 200}
(4)                 formats = []
(4)                 plot_formats = config.plot_formats
(4)                 for fmt in plot_formats:
(8)                     if isinstance(fmt, str):
(12)                        if ':' in fmt:
(16)                            suffix, dpi = fmt.split(':')
(16)                            formats.append((str(suffix), int(dpi)))
(12)                        else:
(16)                            formats.append((fmt, default_dpi.get(fmt, 80)))
(8)                     elif isinstance(fmt, (tuple, list)) and len(fmt) == 2:
(12)                        formats.append((str(fmt[0]), int(fmt[1])))
(8)                     else:
(12)                        raise PlotError('invalid image format "%r" in plot_formats' % fmt)
(4)                 return formats
(0)             def _parse_srcset(entries):
(4)                 """
(4)                 Parse srcset for multiples...
(4)                 """
(4)                 srcset = {}
(4)                 for entry in entries:
(8)                     entry = entry.strip()
(8)                     if len(entry) >= 2:
(12)                        mult = entry[:-1]
(12)                        srcset[float(mult)] = entry
(8)                     else:
(12)                        raise ExtensionError(f'srcset argument {entry!r} is invalid.')
(4)                 return srcset
(0)             def render_figures(code, code_path, output_dir, output_base, context,
(19)                               function_name, config, context_reset=False,
(19)                               close_figs=False,
(19)                               code_includes=None):
(4)                 """
(4)                 Run a pyplot script and save the images in *output_dir*.
(4)                 Save the images under *output_dir* with file names derived from
(4)                 *output_base*
(4)                 """
(4)                 if function_name is not None:
(8)                     output_base = f'{output_base}_{function_name}'
(4)                 formats = get_plot_formats(config)
(4)                 is_doctest, code_pieces = _split_code_at_show(code, function_name)
(4)                 img = ImageFile(output_base, output_dir)
(4)                 for format, dpi in formats:
(8)                     if context or out_of_date(code_path, img.filename(format),
(34)                                              includes=code_includes):
(12)                        all_exists = False
(12)                        break
(8)                     img.formats.append(format)
(4)                 else:
(8)                     all_exists = True
(4)                 if all_exists:
(8)                     return [(code, [img])]
(4)                 results = []
(4)                 for i, code_piece in enumerate(code_pieces):
(8)                     images = []
(8)                     for j in itertools.count():
(12)                        if len(code_pieces) > 1:
(16)                            img = ImageFile('%s_%02d_%02d' % (output_base, i, j),
(32)                                            output_dir)
(12)                        else:
(16)                            img = ImageFile('%s_%02d' % (output_base, j), output_dir)
(12)                        for fmt, dpi in formats:
(16)                            if context or out_of_date(code_path, img.filename(fmt),
(42)                                                      includes=code_includes):
(20)                                all_exists = False
(20)                                break
(16)                            img.formats.append(fmt)
(12)                        if not all_exists:
```

```
(16)                            all_exists = (j > 0)
(16)                            break
(12)                        images.append(img)
(8)                     if not all_exists:
(12)                        break
(8)                     results.append((code_piece, images))
(4)                 else:
(8)                     all_exists = True
(4)                 if all_exists:
(8)                     return results
(4)                 results = []
(4)                 ns = plot_context if context else {}
(4)                 if context_reset:
(8)                     clear_state(config.plot_rcparams)
(8)                     plot_context.clear()
(4)                 close_figs = not context or close_figs
(4)                 for i, code_piece in enumerate(code_pieces):
(8)                     if not context or config.plot_apply_rcparams:
(12)                        clear_state(config.plot_rcparams, close_figs)
(8)                     elif close_figs:
(12)                        plt.close('all')
(8)                     _run_code(doctest.script_from_examples(code_piece) if is_doctest
(18)                              else code_piece,
(18)                              code_path, ns, function_name)
(8)                     images = []
(8)                     fig_managers = _pylab_helpers.Gcf.get_all_fig_managers()
(8)                     for j, figman in enumerate(fig_managers):
(12)                        if len(fig_managers) == 1 and len(code_pieces) == 1:
(16)                            img = ImageFile(output_base, output_dir)
(12)                        elif len(code_pieces) == 1:
(16)                            img = ImageFile("%s_%02d" % (output_base, j), output_dir)
(12)                        else:
(16)                            img = ImageFile("%s_%02d_%02d" % (output_base, i, j),
(32)                                            output_dir)
(12)                        images.append(img)
(12)                        for fmt, dpi in formats:
(16)                            try:
(20)                                figman.canvas.figure.savefig(img.filename(fmt), dpi=dpi)
(20)                                if fmt == formats[0][0] and config.plot_srcset:
(24)                                    srcset = _parse_srcset(config.plot_srcset)
(24)                                    for mult, suffix in srcset.items():
(28)                                        fm = f'{suffix}.{fmt}'
(28)                                        img.formats.append(fm)
(28)                                        figman.canvas.figure.savefig(img.filename(fm),
(57)                                                                     dpi=int(dpi * mult))
(16)                            except Exception as err:
(20)                                raise PlotError(traceback.format_exc()) from err
(16)                            img.formats.append(fmt)
(8)                     results.append((code_piece, images))
(4)                 if not context or config.plot_apply_rcparams:
(8)                     clear_state(config.plot_rcparams, close=not context)
(4)                 return results
(0)             def run(arguments, content, options, state_machine, state, lineno):
(4)                 document = state_machine.document
(4)                 config = document.settings.env.config
(4)                 nofigs = 'nofigs' in options
(4)                 if config.plot_srcset and setup.app.builder.name == 'singlehtml':
(8)                     raise ExtensionError(
(12)                        'plot_srcset option not compatible with single HTML writer')
(4)                 formats = get_plot_formats(config)
(4)                 default_fmt = formats[0][0]
(4)                 options.setdefault('include-source', config.plot_include_source)
(4)                 options.setdefault('show-source-link', config.plot_html_show_source_link)
(4)                 if 'class' in options:
(8)                     options['class'] = ['plot-directive'] + options['class']
(4)                 else:
(8)                     options.setdefault('class', ['plot-directive'])
(4)                 keep_context = 'context' in options
(4)                 context_opt = None if not keep_context else options['context']
```

```
(4)                 rst_file = document.attributes['source']
(4)                 rst_dir = os.path.dirname(rst_file)
(4)                 if len(arguments):
(8)                     if not config.plot_basedir:
(12)                        source_file_name = os.path.join(setup.app.builder.srcdir,
(44)                                                        directives.uri(arguments[0]))
(8)                     else:
(12)                        source_file_name = os.path.join(setup.confdir,
config.plot_basedir,
(44)                                                        directives.uri(arguments[0]))
(8)                     caption = '\n'.join(content)
(8)                     if "caption" in options:
(12)                        if caption:
(16)                            raise ValueError(
(20)                                'Caption specified in both content and options.'
(20)                                ' Please remove ambiguity.'
(16)                            )
(12)                        caption = options["caption"]
(8)                     if len(arguments) == 2:
(12)                        function_name = arguments[1]
(8)                     else:
(12)                        function_name = None
(8)                     code = Path(source_file_name).read_text(encoding='utf-8')
(8)                     output_base = os.path.basename(source_file_name)
(4)                 else:
(8)                     source_file_name = rst_file
(8)                     code = textwrap.dedent("\n".join(map(str, content)))
(8)                     counter = document.attributes.get('_plot_counter', 0) + 1
(8)                     document.attributes['_plot_counter'] = counter
(8)                     base, ext = os.path.splitext(os.path.basename(source_file_name))
(8)                     output_base = '%s-%d.py' % (base, counter)
(8)                     function_name = None
(8)                     caption = options.get('caption', '')
(4)                 base, source_ext = os.path.splitext(output_base)
(4)                 if source_ext in ('.py', '.rst', '.txt'):
(8)                     output_base = base
(4)                 else:
(8)                     source_ext = ''
(4)                 output_base = output_base.replace('.', '-')
(4)                 is_doctest = contains_doctest(code)
(4)                 if 'format' in options:
(8)                     if options['format'] == 'python':
(12)                        is_doctest = False
(8)                     else:
(12)                        is_doctest = True
(4)                 source_rel_name = relpath(source_file_name, setup.confdir)
(4)                 source_rel_dir = os.path.dirname(source_rel_name).lstrip(os.path.sep)
(4)                 build_dir = os.path.join(os.path.dirname(setup.app.doctreedir),
(29)                                         'plot_directive',
(29)                                         source_rel_dir)
(4)                 build_dir = os.path.normpath(build_dir)
(4)                 os.makedirs(build_dir, exist_ok=True)
(4)                 try:
(8)                     build_dir_link = relpath(build_dir, rst_dir).replace(os.path.sep, '/')
(4)                 except ValueError:
(8)                     build_dir_link = build_dir
(4)                 try:
(8)                     source_file_includes = [os.path.join(os.getcwd(), t[0])
(32)                                            for t in state.document.include_log]
(4)                 except AttributeError:
(8)                     possible_sources = {os.path.join(setup.confdir, t[0])
(28)                                        for t in state_machine.input_lines.items}
(8)                     source_file_includes = [f for f in possible_sources
(32)                                            if os.path.isfile(f)]
(4)                 try:
(8)                     source_file_includes.remove(source_file_name)
(4)                 except ValueError:
(8)                     pass
(4)                 if options['show-source-link']:
```

```
(8)                     Path(build_dir, output_base + source_ext).write_text(
(12)                        doctest.script_from_examples(code)
(12)                        if source_file_name == rst_file and is_doctest
(12)                        else code,
(12)                        encoding='utf-8')
(4)                 try:
(8)                     results = render_figures(code=code,
(33)                                             code_path=source_file_name,
(33)                                             output_dir=build_dir,
(33)                                             output_base=output_base,
(33)                                             context=keep_context,
(33)                                             function_name=function_name,
(33)                                             config=config,
(33)                                             context_reset=context_opt == 'reset',
(33)                                             close_figs=context_opt == 'close-figs',
(33)                                             code_includes=source_file_includes)
(8)                     errors = []
(4)                 except PlotError as err:
(8)                     reporter = state.memo.reporter
(8)                     sm = reporter.system_message(
(12)                        2, "Exception occurred in plotting {}\n from {}:\n{}".format(
(16)                            output_base, source_file_name, err),
(12)                        line=lineno)
(8)                     results = [(code, [])]
(8)                     errors = [sm]
(4)                 if caption and config.plot_srcset:
(8)                     caption = f':caption: {caption}'
(4)                 elif caption:
(8)                     caption = '\n' + '\n'.join('      ' + line.strip()
(35)                                               for line in caption.split('\n'))
(4)                 total_lines = []
(4)                 for j, (code_piece, images) in enumerate(results):
(8)                     if options['include-source']:
(12)                        if is_doctest:
(16)                            lines = ['', *code_piece.splitlines()]
(12)                        else:
(16)                            lines = ['.. code-block:: python', '',
(25)                                     *textwrap.indent(code_piece, '    ').splitlines()]
(12)                        source_code = "\n".join(lines)
(8)                     else:
(12)                        source_code = ""
(8)                     if nofigs:
(12)                        images = []
(8)                     opts = [
(12)                        f':{key}: {val}' for key, val in options.items()
(12)                        if key in ('alt', 'height', 'width', 'scale', 'align', 'class')]
(8)                     if j == 0 and options['show-source-link']:
(12)                        src_name = output_base + source_ext
(8)                     else:
(12)                        src_name = None
(8)                     if config.plot_srcset:
(12)                        srcset = [*_parse_srcset(config.plot_srcset).values()]
(12)                        template = TEMPLATE_SRCSET
(8)                     else:
(12)                        srcset = None
(12)                        template = TEMPLATE
(8)                     result = jinja2.Template(config.plot_template or template).render(
(12)                        default_fmt=default_fmt,
(12)                        build_dir=build_dir_link,
(12)                        src_name=src_name,
(12)                        multi_image=len(images) > 1,
(12)                        options=opts,
(12)                        srcset=srcset,
(12)                        images=images,
(12)                        source_code=source_code,
(12)                        html_show_formats=config.plot_html_show_formats and len(images),
(12)                        caption=caption)
(8)                     total_lines.extend(result.split("\n"))
(8)                     total_lines.extend("\n")
```

```
705: (4)                if total_lines:
706: (8)                    state_machine.insert_input(total_lines, source=source_file_name)
707: (4)                return errors
```

File 123 - roles.py:

```
1: (0)              """
2: (0)              Custom roles for the Matplotlib documentation.
3: (0)              .. warning::
4: (4)                  These roles are considered semi-public. They are only intended to be used
in
5: (4)                  the Matplotlib documentation.
6: (0)              However, it can happen that downstream packages end up pulling these roles
into
7: (0)              their documentation, which will result in documentation build errors. The
following
8: (0)              describes the exact mechanism and how to fix the errors.
9: (0)              There are two ways, Matplotlib docstrings can end up in downstream
documentation.
10: (0)             You have to subclass a Matplotlib class and either use the ``:inherited-
members:``
11: (0)             option in your autodoc configuration, or you have to override a method without
12: (0)             specifying a new docstring; the new method will inherit the original docstring
and
13: (0)             still render in your autodoc. If the docstring contains one of the custom
sphinx
14: (0)             roles, you'll see one of the following error messages:
15: (0)             .. code-block:: none
16: (4)                 Unknown interpreted text role "mpltype".
17: (4)                 Unknown interpreted text role "rc".
18: (0)             To fix this, you can add this module as extension to your sphinx
:file:`conf.py`::
19: (4)                 extensions = [
20: (8)                     'matplotlib.sphinxext.roles',
21: (4)                 ]
22: (0)             .. warning::
23: (4)                 Direct use of these roles in other packages is not officially supported.
We
24: (4)                 reserve the right to modify or remove these roles without prior
notification.
25: (0)             """
26: (0)             from urllib.parse import urlsplit, urlunsplit
27: (0)             from docutils import nodes
28: (0)             import matplotlib
29: (0)             from matplotlib import rcParamsDefault
30: (0)             class _QueryReference(nodes.Inline, nodes.TextElement):
31: (4)                 """
32: (4)                 Wraps a reference or pending reference to add a query string.
33: (4)                 The query string is generated from the attributes added to this node.
34: (4)                 Also equivalent to a `~docutils.nodes.literal` node.
35: (4)                 """
36: (4)                 def to_query_string(self):
37: (8)                     """Generate query string from node attributes."""
38: (8)                     return '&'.join(f'{name}={value}' for name, value in self.attlist())
39: (0)             def _visit_query_reference_node(self, node):
40: (4)                 """
41: (4)                 Resolve *node* into query strings on its ``reference`` children.
42: (4)                 Then act as if this is a `~docutils.nodes.literal`.
43: (4)                 """
44: (4)                 query = node.to_query_string()
45: (4)                 for refnode in node.findall(nodes.reference):
46: (8)                     uri = urlsplit(refnode['refuri'])._replace(query=query)
47: (8)                     refnode['refuri'] = urlunsplit(uri)
48: (4)                 self.visit_literal(node)
49: (0)             def _depart_query_reference_node(self, node):
50: (4)                 """
51: (4)                 Act as if this is a `~docutils.nodes.literal`.
```

```
52: (4)                 """
53: (4)                 self.depart_literal(node)
54: (0)             def _rcparam_role(name, rawtext, text, lineno, inliner, options=None,
content=None):
55: (4)                 """
56: (4)                 Sphinx role ``:rc:`` to highlight and link ``rcParams`` entries.
57: (4)                 Usage: Give the desired ``rcParams`` key as parameter.
58: (4)                 :code:`:rc:`figure.dpi`` will render as: :rc:`figure.dpi`
59: (4)                 """
60: (4)                 title = f'rcParams["{text}"]'
61: (4)                 target = 'matplotlibrc-sample'
62: (4)                 ref_nodes, messages = inliner.interpreted(title, f'{title} <{target}>',
63: (46)                                                          'ref', lineno)
64: (4)                 qr = _QueryReference(rawtext, highlight=text)
65: (4)                 qr += ref_nodes
66: (4)                 node_list = [qr]
67: (4)                 if text in rcParamsDefault and text != "backend":
68: (8)                     node_list.extend([
69: (12)                        nodes.Text(' (default: '),
70: (12)                        nodes.literal('', repr(rcParamsDefault[text])),
71: (12)                        nodes.Text(')'),
72: (12)                        ])
73: (4)                 return node_list, messages
74: (0)             def _mpltype_role(name, rawtext, text, lineno, inliner, options=None,
content=None):
75: (4)                 """
76: (4)                 Sphinx role ``:mpltype:`` for custom matplotlib types.
77: (4)                 In Matplotlib, there are a number of type-like concepts that do not have a
78: (4)                 direct type representation; example: color. This role allows to properly
79: (4)                 highlight them in the docs and link to their definition.
80: (4)                 Currently supported values:
81: (4)                 - :code:`:mpltype:`color`` will render as: :mpltype:`color`
82: (4)                 """
83: (4)                 mpltype = text
84: (4)                 type_to_link_target = {
85: (8)                     'color': 'colors_def',
86: (4)                 }
87: (4)                 if mpltype not in type_to_link_target:
88: (8)                     raise ValueError(f"Unknown mpltype: {mpltype!r}")
89: (4)                 node_list, messages = inliner.interpreted(
90: (8)                     mpltype, f'{mpltype} <{type_to_link_target[mpltype]}>', 'ref', lineno)
91: (4)                 return node_list, messages
92: (0)             def setup(app):
93: (4)                 app.add_role("rc", _rcparam_role)
94: (4)                 app.add_role("mpltype", _mpltype_role)
95: (4)                 app.add_node(
96: (8)                     _QueryReference,
97: (8)                     html=(_visit_query_reference_node, _depart_query_reference_node),
98: (8)                     latex=(_visit_query_reference_node, _depart_query_reference_node),
99: (8)                     text=(_visit_query_reference_node, _depart_query_reference_node),
100: (4)                )
101: (4)                return {"version": matplotlib.__version__,
102: (12)                       "parallel_read_safe": True, "parallel_write_safe": True}

----------------------------------------

File 124 - __init__.py:

1: (0)              from .core import available, context, library, reload_library, use
2: (0)              __all__ = ["available", "context", "library", "reload_library", "use"]

----------------------------------------

File 125 - core.py:

1: (0)              """
2: (0)              Core functions and attributes for the matplotlib style library:
3: (0)              ``use``
4: (4)                  Select style sheet to override the current matplotlib settings.
```

```
5: (0)              ``context``
6: (4)                  Context manager to use a style sheet temporarily.
7: (0)              ``available``
8: (4)                  List available style sheets.
9: (0)              ``library``
10: (4)                 A dictionary of style names and matplotlib settings.
11: (0)             """
12: (0)             import contextlib
13: (0)             import logging
14: (0)             import os
15: (0)             from pathlib import Path
16: (0)             import sys
17: (0)             import warnings
18: (0)             if sys.version_info >= (3, 10):
19: (4)                 import importlib.resources as importlib_resources
20: (0)             else:
21: (4)                 import importlib_resources
22: (0)             import matplotlib as mpl
23: (0)             from matplotlib import _api, _docstring, _rc_params_in_file, rcParamsDefault
24: (0)             _log = logging.getLogger(__name__)
25: (0)             __all__ = ['use', 'context', 'available', 'library', 'reload_library']
26: (0)             BASE_LIBRARY_PATH = os.path.join(mpl.get_data_path(), 'stylelib')
27: (0)             USER_LIBRARY_PATHS = [os.path.join(mpl.get_configdir(), 'stylelib')]
28: (0)             STYLE_EXTENSION = 'mplstyle'
29: (0)             STYLE_BLACKLIST = {
30: (4)                 'interactive', 'backend', 'webagg.port', 'webagg.address',
31: (4)                 'webagg.port_retries', 'webagg.open_in_browser', 'backend_fallback',
32: (4)                 'toolbar', 'timezone', 'figure.max_open_warning',
33: (4)                 'figure.raise_window', 'savefig.directory', 'tk.window_focus',
34: (4)                 'docstring.hardcopy', 'date.epoch'}
35: (0)             @_docstring.Substitution(
36: (4)                 "\n".join(map("- {}".format, sorted(STYLE_BLACKLIST, key=str.lower)))
37: (0)             )
38: (0)             def use(style):
39: (4)                 """
40: (4)                 Use Matplotlib style settings from a style specification.
41: (4)                 The style name of 'default' is reserved for reverting back to
42: (4)                 the default style settings.
43: (4)                 .. note::
44: (7)                    This updates the `.rcParams` with the settings from the style.
45: (7)                    `.rcParams` not defined in the style are kept.
46: (4)                 Parameters
47: (4)                 ----------
48: (4)                 style : str, dict, Path or list
49: (8)                     A style specification. Valid options are:
50: (8)                     str
51: (12)                        - One of the style names in `.style.available` (a builtin style or
52: (14)                          a style installed in the user library path).
53: (12)                        - A dotted name of the form "package.style_name"; in that case,
54: (14)                          "package" should be an importable Python package name, e.g. at
55: (14)                          ``/path/to/package/__init__.py``; the loaded style file is
56: (14)                          ``/path/to/package/style_name.mplstyle``.  (Style files in
57: (14)                          subpackages are likewise supported.)
58: (12)                        - The path or URL to a style file, which gets loaded by
59: (14)                          `.rc_params_from_file`.
60: (8)                     dict
61: (12)                        A mapping of key/value pairs for `matplotlib.rcParams`.
62: (8)                     Path
63: (12)                        The path to a style file, which gets loaded by
64: (12)                        `.rc_params_from_file`.
65: (8)                     list
66: (12)                        A list of style specifiers (str, Path or dict), which are applied
67: (12)                        from first to last in the list.
68: (4)                 Notes
69: (4)                 -----
70: (4)                 The following `.rcParams` are not related to style and will be ignored if
71: (4)                 found in a style specification:
72: (4)                 %s
73: (4)                 """
```

```
(4)                 if isinstance(style, (str, Path)) or hasattr(style, 'keys'):
(8)                     styles = [style]
(4)                 else:
(8)                     styles = style
(4)                 style_alias = {'mpl20': 'default', 'mpl15': 'classic'}
(4)                 for style in styles:
(8)                     if isinstance(style, str):
(12)                        style = style_alias.get(style, style)
(12)                        if style == "default":
(16)                            with _api.suppress_matplotlib_deprecation_warning():
(20)                                style = {k: rcParamsDefault[k] for k in rcParamsDefault
(29)                                         if k not in STYLE_BLACKLIST}
(12)                        elif style in library:
(16)                            style = library[style]
(12)                        elif "." in style:
(16)                            pkg, _, name = style.rpartition(".")
(16)                            try:
(20)                                path = (importlib_resources.files(pkg)
(28)                                        / f"{name}.{STYLE_EXTENSION}")
(20)                                style = _rc_params_in_file(path)
(16)                            except (ModuleNotFoundError, OSError, TypeError) as exc:
(20)                                pass
(8)                     if isinstance(style, (str, Path)):
(12)                        try:
(16)                            style = _rc_params_in_file(style)
(12)                        except OSError as err:
(16)                            raise OSError(
(20)                                f"{style!r} is not a valid package style, path of style "
(20)                                f"file, URL of style file, or library style name (library
"
(20)                                f"styles are listed in `style.available`)") from err
(8)                     filtered = {}
(8)                     for k in style:  # don't trigger RcParams.__getitem__('backend')
(12)                        if k in STYLE_BLACKLIST:
(16)                            _api.warn_external(
(20)                                f"Style includes a parameter, {k!r}, that is not "
(20)                                f"related to style.  Ignoring this parameter.")
(12)                        else:
(16)                            filtered[k] = style[k]
(8)                     mpl.rcParams.update(filtered)
(0)             @contextlib.contextmanager
(0)             def context(style, after_reset=False):
(4)                 """
(4)                 Context manager for using style settings temporarily.
(4)                 Parameters
(4)                 ----------
(4)                 style : str, dict, Path or list
(8)                     A style specification. Valid options are:
(8)                     str
(12)                        - One of the style names in `.style.available` (a builtin style or
(14)                          a style installed in the user library path).
(12)                        - A dotted name of the form "package.style_name"; in that case,
(14)                          "package" should be an importable Python package name, e.g. at
(14)                          ``/path/to/package/__init__.py``; the loaded style file is
(14)                          ``/path/to/package/style_name.mplstyle``.  (Style files in
(14)                          subpackages are likewise supported.)
(12)                        - The path or URL to a style file, which gets loaded by
(14)                          `.rc_params_from_file`.
(8)                     dict
(12)                        A mapping of key/value pairs for `matplotlib.rcParams`.
(8)                     Path
(12)                        The path to a style file, which gets loaded by
(12)                        `.rc_params_from_file`.
(8)                     list
(12)                        A list of style specifiers (str, Path or dict), which are applied
(12)                        from first to last in the list.
(4)                 after_reset : bool
(8)                     If True, apply style after resetting settings to their defaults;
(8)                     otherwise, apply style on top of the current settings.
```

```
142: (4)                """
143: (4)                with mpl.rc_context():
144: (8)                    if after_reset:
145: (12)                       mpl.rcdefaults()
146: (8)                    use(style)
147: (8)                    yield
148: (0)            def update_user_library(library):
149: (4)                """Update style library with user-defined rc files."""
150: (4)                for stylelib_path in map(os.path.expanduser, USER_LIBRARY_PATHS):
151: (8)                    styles = read_style_directory(stylelib_path)
152: (8)                    update_nested_dict(library, styles)
153: (4)                return library
154: (0)            def read_style_directory(style_dir):
155: (4)                """Return dictionary of styles defined in *style_dir*."""
156: (4)                styles = dict()
157: (4)                for path in Path(style_dir).glob(f"*.{STYLE_EXTENSION}"):
158: (8)                    with warnings.catch_warnings(record=True) as warns:
159: (12)                       styles[path.stem] = _rc_params_in_file(path)
160: (8)                    for w in warns:
161: (12)                       _log.warning('In %s: %s', path, w.message)
162: (4)                return styles
163: (0)            def update_nested_dict(main_dict, new_dict):
164: (4)                """
165: (4)                Update nested dict (only level of nesting) with new values.
166: (4)                Unlike `dict.update`, this assumes that the values of the parent dict are
167: (4)                dicts (or dict-like), so you shouldn't replace the nested dict if it
168: (4)                already exists. Instead you should update the sub-dict.
169: (4)                """
170: (4)                for name, rc_dict in new_dict.items():
171: (8)                    main_dict.setdefault(name, {}).update(rc_dict)
172: (4)                return main_dict
173: (0)            _base_library = read_style_directory(BASE_LIBRARY_PATH)
174: (0)            library = {}
175: (0)            available = []
176: (0)            def reload_library():
177: (4)                """Reload the style library."""
178: (4)                library.clear()
179: (4)                library.update(update_user_library(_base_library))
180: (4)                available[:] = sorted(library.keys())
181: (0)            reload_library()
```

File 126 - __init__.py:

```
1: (0)              """
2: (0)              Helper functions for testing.
3: (0)              """
4: (0)              from pathlib import Path
5: (0)              from tempfile import TemporaryDirectory
6: (0)              import locale
7: (0)              import logging
8: (0)              import os
9: (0)              import subprocess
10: (0)             import sys
11: (0)             import matplotlib as mpl
12: (0)             from matplotlib import _api
13: (0)             _log = logging.getLogger(__name__)
14: (0)             def set_font_settings_for_testing():
15: (4)                 mpl.rcParams['font.family'] = 'DejaVu Sans'
16: (4)                 mpl.rcParams['text.hinting'] = 'none'
17: (4)                 mpl.rcParams['text.hinting_factor'] = 8
18: (0)             def set_reproducibility_for_testing():
19: (4)                 mpl.rcParams['svg.hashsalt'] = 'matplotlib'
20: (0)             def setup():
21: (4)                 try:
22: (8)                     locale.setlocale(locale.LC_ALL, 'en_US.UTF-8')
23: (4)                 except locale.Error:
24: (8)                     try:
```

```
 25: (12)                        locale.setlocale(locale.LC_ALL, 'English_United States.1252')
 26: (8)                     except locale.Error:
 27: (12)                        _log.warning(
 28: (16)                            "Could not set locale to English/United States. "
 29: (16)                            "Some date-related tests may fail.")
 30: (4)                 mpl.use('Agg')
 31: (4)                 with _api.suppress_matplotlib_deprecation_warning():
 32: (8)                     mpl.rcdefaults()  # Start with all defaults
 33: (4)                 set_font_settings_for_testing()
 34: (4)                 set_reproducibility_for_testing()
 35: (0)             def subprocess_run_for_testing(command, env=None, timeout=60, stdout=None,
 36: (31)                                           stderr=None, check=False, text=True,
 37: (31)                                           capture_output=False):
 38: (4)                 """
 39: (4)                 Create and run a subprocess.
 40: (4)                 Thin wrapper around `subprocess.run`, intended for testing.  Will
 41: (4)                 mark fork() failures on Cygwin as expected failures: not a
 42: (4)                 success, but not indicating a problem with the code either.
 43: (4)                 Parameters
 44: (4)                 ----------
 45: (4)                 args : list of str
 46: (4)                 env : dict[str, str]
 47: (4)                 timeout : float
 48: (4)                 stdout, stderr
 49: (4)                 check : bool
 50: (4)                 text : bool
 51: (8)                     Also called ``universal_newlines`` in subprocess.  I chose this
 52: (8)                     name since the main effect is returning bytes (`False`) vs. str
 53: (8)                     (`True`), though it also tries to normalize newlines across
 54: (8)                     platforms.
 55: (4)                 capture_output : bool
 56: (8)                     Set stdout and stderr to subprocess.PIPE
 57: (4)                 Returns
 58: (4)                 -------
 59: (4)                 proc : subprocess.Popen
 60: (4)                 See Also
 61: (4)                 --------
 62: (4)                 subprocess.run
 63: (4)                 Raises
 64: (4)                 ------
 65: (4)                 pytest.xfail
 66: (8)                     If platform is Cygwin and subprocess reports a fork() failure.
 67: (4)                 """
 68: (4)                 if capture_output:
 69: (8)                     stdout = stderr = subprocess.PIPE
 70: (4)                 try:
 71: (8)                     proc = subprocess.run(
 72: (12)                        command, env=env,
 73: (12)                        timeout=timeout, check=check,
 74: (12)                        stdout=stdout, stderr=stderr,
 75: (12)                        text=text
 76: (8)                     )
 77: (4)                 except BlockingIOError:
 78: (8)                     if sys.platform == "cygwin":
 79: (12)                        import pytest
 80: (12)                        pytest.xfail("Fork failure")
 81: (8)                     raise
 82: (4)                 return proc
 83: (0)             def subprocess_run_helper(func, *args, timeout, extra_env=None):
 84: (4)                 """
 85: (4)                 Run a function in a sub-process.
 86: (4)                 Parameters
 87: (4)                 ----------
 88: (4)                 func : function
 89: (8)                     The function to be run.  It must be in a module that is importable.
 90: (4)                 *args : str
 91: (8)                     Any additional command line arguments to be passed in
 92: (8)                     the first argument to ``subprocess.run``.
 93: (4)                 extra_env : dict[str, str]
```

```
 94: (8)                     Any additional environment variables to be set for the subprocess.
 95: (4)                 """
 96: (4)                 target = func.__name__
 97: (4)                 module = func.__module__
 98: (4)                 file = func.__code__.co_filename
 99: (4)                 proc = subprocess_run_for_testing(
100: (8)                     [
101: (12)                        sys.executable,
102: (12)                        "-c",
103: (12)                        f"import importlib.util;"
104: (12)                        f"_spec = importlib.util.spec_from_file_location({module!r},
{file!r});"
105: (12)                        f"_module = importlib.util.module_from_spec(_spec);"
106: (12)                        f"_spec.loader.exec_module(_module);"
107: (12)                        f"_module.{target}()",
108: (12)                        *args
109: (8)                     ],
110: (8)                     env={**os.environ, "SOURCE_DATE_EPOCH": "0", **(extra_env or {})},
111: (8)                     timeout=timeout, check=True,
112: (8)                     stdout=subprocess.PIPE,
113: (8)                     stderr=subprocess.PIPE,
114: (8)                     text=True
115: (4)                 )
116: (4)                 return proc
117: (0)             def _check_for_pgf(texsystem):
118: (4)                 """
119: (4)                 Check if a given TeX system + pgf is available
120: (4)                 Parameters
121: (4)                 ----------
122: (4)                 texsystem : str
123: (8)                     The executable name to check
124: (4)                 """
125: (4)                 with TemporaryDirectory() as tmpdir:
126: (8)                     tex_path = Path(tmpdir, "test.tex")
127: (8)                     tex_path.write_text(r"""
128: (12)                        \documentclass{article}
129: (12)                        \usepackage{pgf}
130: (12)                        \begin{document}
131: (12)                        \typeout{pgfversion=\pgfversion}
132: (12)                        \makeatletter
133: (12)                        \@@end
134: (8)                     """, encoding="utf-8")
135: (8)                     try:
136: (12)                        subprocess.check_call(
137: (16)                            [texsystem, "-halt-on-error", str(tex_path)], cwd=tmpdir,
138: (16)                            stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
139: (8)                     except (OSError, subprocess.CalledProcessError):
140: (12)                        return False
141: (8)                     return True
142: (0)             def _has_tex_package(package):
143: (4)                 try:
144: (8)                     mpl.dviread.find_tex_file(f"{package}.sty")
145: (8)                     return True
146: (4)                 except FileNotFoundError:
147: (8)                     return False
148: (0)             def ipython_in_subprocess(requested_backend_or_gui_framework,
all_expected_backends):
149: (4)                 import pytest
150: (4)                 IPython = pytest.importorskip("IPython")
151: (4)                 if sys.platform == "win32":
152: (8)                     pytest.skip("Cannot change backend running IPython in subprocess on
Windows")
153: (4)                 if (IPython.version_info[:3] == (8, 24, 0) and
154: (12)                        requested_backend_or_gui_framework == "osx"):
155: (8)                     pytest.skip("Bug using macosx backend in IPython 8.24.0 fixed in
8.24.1")
156: (4)                 for min_version, backend in all_expected_backends.items():
157: (8)                     if IPython.version_info[:2] >= min_version:
158: (12)                        expected_backend = backend
```

```
159: (12)                        break
160: (4)                code = ("import matplotlib as mpl, matplotlib.pyplot as plt;"
161: (12)                        "fig, ax=plt.subplots(); ax.plot([1, 3, 2]); mpl.get_backend()")
162: (4)                proc = subprocess_run_for_testing(
163: (8)                    [
164: (12)                       "ipython",
165: (12)                       "--no-simple-prompt",
166: (12)                       f"--matplotlib={requested_backend_or_gui_framework}",
167: (12)                       "-c", code,
168: (8)                    ],
169: (8)                    check=True,
170: (8)                    capture_output=True,
171: (4)                )
172: (4)                assert proc.stdout.strip().endswith(f"'{expected_backend}'")
173: (0)            def is_ci_environment():
174: (4)                ci_environment_variables = [
175: (8)                    'CI',        # Generic CI environment variable
176: (8)                    'CONTINUOUS_INTEGRATION',  # Generic CI environment variable
177: (8)                    'TRAVIS',    # Travis CI
178: (8)                    'CIRCLECI',  # CircleCI
179: (8)                    'JENKINS',   # Jenkins
180: (8)                    'GITLAB_CI',  # GitLab CI
181: (8)                    'GITHUB_ACTIONS',  # GitHub Actions
182: (8)                    'TEAMCITY_VERSION'  # TeamCity
183: (4)                ]
184: (4)                for env_var in ci_environment_variables:
185: (8)                    if os.getenv(env_var):
186: (12)                       return True
187: (4)                return False
```

File 127 - compare.py:

```
1: (0)              """
2: (0)              Utilities for comparing image results.
3: (0)              """
4: (0)              import atexit
5: (0)              import functools
6: (0)              import hashlib
7: (0)              import logging
8: (0)              import os
9: (0)              from pathlib import Path
10: (0)             import shutil
11: (0)             import subprocess
12: (0)             import sys
13: (0)             from tempfile import TemporaryDirectory, TemporaryFile
14: (0)             import weakref
15: (0)             import numpy as np
16: (0)             from PIL import Image
17: (0)             import matplotlib as mpl
18: (0)             from matplotlib import cbook
19: (0)             from matplotlib.testing.exceptions import ImageComparisonFailure
20: (0)             _log = logging.getLogger(__name__)
21: (0)             __all__ = ['calculate_rms', 'comparable_formats', 'compare_images']
22: (0)             def make_test_filename(fname, purpose):
23: (4)                 """
24: (4)                 Make a new filename by inserting *purpose* before the file's extension.
25: (4)                 """
26: (4)                 base, ext = os.path.splitext(fname)
27: (4)                 return f'{base}-{purpose}{ext}'
28: (0)             def _get_cache_path():
29: (4)                 cache_dir = Path(mpl.get_cachedir(), 'test_cache')
30: (4)                 cache_dir.mkdir(parents=True, exist_ok=True)
31: (4)                 return cache_dir
32: (0)             def get_cache_dir():
33: (4)                 return str(_get_cache_path())
34: (0)             def get_file_hash(path, block_size=2 ** 20):
35: (4)                 md5 = hashlib.md5()
```

```
 36: (4)                 with open(path, 'rb') as fd:
 37: (8)                     while True:
 38: (12)                        data = fd.read(block_size)
 39: (12)                        if not data:
 40: (16)                            break
 41: (12)                        md5.update(data)
 42: (4)                 if Path(path).suffix == '.pdf':
 43: (8)                     md5.update(str(mpl._get_executable_info("gs").version)
 44: (19)                                .encode('utf-8'))
 45: (4)                 elif Path(path).suffix == '.svg':
 46: (8)                     md5.update(str(mpl._get_executable_info("inkscape").version)
 47: (19)                                .encode('utf-8'))
 48: (4)                 return md5.hexdigest()
 49: (0)             class _ConverterError(Exception):
 50: (4)                 pass
 51: (0)             class _Converter:
 52: (4)                 def __init__(self):
 53: (8)                     self._proc = None
 54: (8)                     atexit.register(self.__del__)
 55: (4)                 def __del__(self):
 56: (8)                     if self._proc:
 57: (12)                        self._proc.kill()
 58: (12)                        self._proc.wait()
 59: (12)                        for stream in filter(None, [self._proc.stdin,
 60: (40)                                                    self._proc.stdout,
 61: (40)                                                    self._proc.stderr]):
 62: (16)                            stream.close()
 63: (12)                        self._proc = None
 64: (4)                 def _read_until(self, terminator):
 65: (8)                     """Read until the prompt is reached."""
 66: (8)                     buf = bytearray()
 67: (8)                     while True:
 68: (12)                        c = self._proc.stdout.read(1)
 69: (12)                        if not c:
 70: (16)                            raise _ConverterError(os.fsdecode(bytes(buf)))
 71: (12)                        buf.extend(c)
 72: (12)                        if buf.endswith(terminator):
 73: (16)                            return bytes(buf)
 74: (0)             class _GSConverter(_Converter):
 75: (4)                 def __call__(self, orig, dest):
 76: (8)                     if not self._proc:
 77: (12)                        self._proc = subprocess.Popen(
 78: (16)                            [mpl._get_executable_info("gs").executable,
 79: (17)                             "-dNOSAFER", "-dNOPAUSE", "-dEPSCrop", "-sDEVICE=png16m"],
 80: (16)                            stdin=subprocess.PIPE, stdout=subprocess.PIPE)
 81: (12)                        try:
 82: (16)                            self._read_until(b"\nGS")
 83: (12)                        except _ConverterError as e:
 84: (16)                            raise OSError(f"Failed to start Ghostscript:\n\n{e.args[0]}")
 from None
 85: (8)                     def encode_and_escape(name):
 86: (12)                        return (os.fsencode(name)
 87: (20)                                .replace(b"\\", b"\\\\")
 88: (20)                                .replace(b"(", br"\(")
 89: (20)                                .replace(b")", br"\)"))
 90: (8)                     self._proc.stdin.write(
 91: (12)                        b"<< /OutputFile ("
 92: (12)                        + encode_and_escape(dest)
 93: (12)                        + b") >> setpagedevice ("
 94: (12)                        + encode_and_escape(orig)
 95: (12)                        + b") run flush\n")
 96: (8)                     self._proc.stdin.flush()
 97: (8)                     err = self._read_until((b"GS<", b"GS>"))
 98: (8)                     stack = self._read_until(b">") if err.endswith(b"GS<") else b""
 99: (8)                     if stack or not os.path.exists(dest):
 100: (12)                       stack_size = int(stack[:-1]) if stack else 0
 101: (12)                       self._proc.stdin.write(b"pop\n" * stack_size)
 102: (12)                       raise ImageComparisonFailure(
 103: (16)                           (err + stack).decode(sys.getfilesystemencoding(), "replace"))
```

```
(0)  class _SVGConverter(_Converter):
(4)      def __call__(self, orig, dest):
(8)          old_inkscape = mpl._get_executable_info("inkscape").version.major < 1
(8)          terminator = b"\n>" if old_inkscape else b"> "
(8)          if not hasattr(self, "_tmpdir"):
(12)             self._tmpdir = TemporaryDirectory()
(12)             weakref.finalize(self._tmpdir, self.__del__)
(8)          if (not self._proc  # First run.
(16)                 or self._proc.poll() is not None):  # Inkscape terminated.
(12)             if self._proc is not None and self._proc.poll() is not None:
(16)                 for stream in filter(None, [self._proc.stdin,
(44)                                             self._proc.stdout,
(44)                                             self._proc.stderr]):
(20)                     stream.close()
(12)             env = {
(16)                 **os.environ,
(16)                 "DISPLAY": "",
(16)                 "INKSCAPE_PROFILE_DIR": self._tmpdir.name,
(12)             }
(12)             stderr = TemporaryFile()
(12)             self._proc = subprocess.Popen(
(16)                 ["inkscape", "--without-gui", "--shell"] if old_inkscape else
(16)                 ["inkscape", "--shell"],
(16)                 stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=stderr,
(16)                 env=env, cwd=self._tmpdir.name)
(12)             self._proc.stderr = stderr
(12)             try:
(16)                 self._read_until(terminator)
(12)             except _ConverterError as err:
(16)                 raise OSError(
(20)                     "Failed to start Inkscape in interactive mode:\n\n"
(20)                     + err.args[0]) from err
(8)          inkscape_orig = Path(self._tmpdir.name, os.fsdecode(b"f.svg"))
(8)          inkscape_dest = Path(self._tmpdir.name, os.fsdecode(b"f.png"))
(8)          try:
(12)             inkscape_orig.symlink_to(Path(orig).resolve())
(8)          except OSError:
(12)             shutil.copyfile(orig, inkscape_orig)
(8)          self._proc.stdin.write(
(12)             b"f.svg --export-png=f.png\n" if old_inkscape else
(12)             b"file-open:f.svg;export-filename:f.png;export-do;file-close\n")
(8)          self._proc.stdin.flush()
(8)          try:
(12)             self._read_until(terminator)
(8)          except _ConverterError as err:
(12)             self._proc.stderr.seek(0)
(12)             raise ImageComparisonFailure(
(16)                 self._proc.stderr.read().decode(
(20)                     sys.getfilesystemencoding(), "replace")) from err
(8)          os.remove(inkscape_orig)
(8)          shutil.move(inkscape_dest, dest)
(4)      def __del__(self):
(8)          super().__del__()
(8)          if hasattr(self, "_tmpdir"):
(12)             self._tmpdir.cleanup()
(0)  class _SVGWithMatplotlibFontsConverter(_SVGConverter):
(4)      """
(4)      A SVG converter which explicitly adds the fonts shipped by Matplotlib to
(4)      Inkspace's font search path, to better support `svg.fonttype = "none"`
(4)      (which is in particular used by certain mathtext tests).
(4)      """
(4)      def __call__(self, orig, dest):
(8)          if not hasattr(self, "_tmpdir"):
(12)             self._tmpdir = TemporaryDirectory()
(12)             shutil.copytree(cbook._get_data_path("fonts/ttf"),
(28)                             Path(self._tmpdir.name, "fonts"))
(8)          return super().__call__(orig, dest)
(0)  def _update_converter():
(4)      try:
```

```
173: (8)                    mpl._get_executable_info("gs")
174: (4)                except mpl.ExecutableNotFoundError:
175: (8)                    pass
176: (4)                else:
177: (8)                    converter['pdf'] = converter['eps'] = _GSConverter()
178: (4)                try:
179: (8)                    mpl._get_executable_info("inkscape")
180: (4)                except mpl.ExecutableNotFoundError:
181: (8)                    pass
182: (4)                else:
183: (8)                    converter['svg'] = _SVGConverter()
184: (0)            converter = {}
185: (0)            _update_converter()
186: (0)            _svg_with_matplotlib_fonts_converter = _SVGWithMatplotlibFontsConverter()
187: (0)            def comparable_formats():
188: (4)                """
189: (4)                Return the list of file formats that `.compare_images` can compare
190: (4)                on this system.
191: (4)                Returns
192: (4)                -------
193: (4)                list of str
194: (8)                    E.g. ``['png', 'pdf', 'svg', 'eps']``.
195: (4)                """
196: (4)                return ['png', *converter]
197: (0)            def convert(filename, cache):
198: (4)                """
199: (4)                Convert the named file to png; return the name of the created file.
200: (4)                If *cache* is True, the result of the conversion is cached in
201: (4)                `matplotlib.get_cachedir() + '/test_cache/'`.  The caching is based on a
202: (4)                hash of the exact contents of the input file.  Old cache entries are
203: (4)                automatically deleted as needed to keep the size of the cache capped to
204: (4)                twice the size of all baseline images.
205: (4)                """
206: (4)                path = Path(filename)
207: (4)                if not path.exists():
208: (8)                    raise OSError(f"{path} does not exist")
209: (4)                if path.suffix[1:] not in converter:
210: (8)                    import pytest
211: (8)                    pytest.skip(f"Don't know how to convert {path.suffix} files to png")
212: (4)                newpath = path.parent / f"{path.stem}_{path.suffix[1:]}.png"
213: (4)                if not newpath.exists() or newpath.stat().st_mtime < path.stat().st_mtime:
214: (8)                    cache_dir = _get_cache_path() if cache else None
215: (8)                    if cache_dir is not None:
216: (12)                       _register_conversion_cache_cleaner_once()
217: (12)                       hash_value = get_file_hash(path)
218: (12)                       cached_path = cache_dir / (hash_value + newpath.suffix)
219: (12)                       if cached_path.exists():
220: (16)                           _log.debug("For %s: reusing cached conversion.", filename)
221: (16)                           shutil.copyfile(cached_path, newpath)
222: (16)                           return str(newpath)
223: (8)                    _log.debug("For %s: converting to png.", filename)
224: (8)                    convert = converter[path.suffix[1:]]
225: (8)                    if path.suffix == ".svg":
226: (12)                       contents = path.read_text()
227: (12)                       if 'style="font:' in contents:
228: (16)                           convert = _svg_with_matplotlib_fonts_converter
229: (8)                    convert(path, newpath)
230: (8)                    if cache_dir is not None:
231: (12)                       _log.debug("For %s: caching conversion result.", filename)
232: (12)                       shutil.copyfile(newpath, cached_path)
233: (4)                return str(newpath)
234: (0)            def _clean_conversion_cache():
235: (4)                baseline_images_size = sum(
236: (8)                    path.stat().st_size
237: (8)                    for path in Path(mpl.__file__).parent.glob("**/baseline_images/**/*"))
238: (4)                max_cache_size = 2 * baseline_images_size
239: (4)                with cbook._lock_path(_get_cache_path()):
240: (8)                    cache_stat = {
241: (12)                       path: path.stat() for path in _get_cache_path().glob("*")}
```

```
242: (8)                    cache_size = sum(stat.st_size for stat in cache_stat.values())
243: (8)                    paths_by_atime = sorted(  # Oldest at the end.
244: (12)                       cache_stat, key=lambda path: cache_stat[path].st_atime,
245: (12)                       reverse=True)
246: (8)                    while cache_size > max_cache_size:
247: (12)                       path = paths_by_atime.pop()
248: (12)                       cache_size -= cache_stat[path].st_size
249: (12)                       path.unlink()
250: (0)            @functools.cache  # Ensure this is only registered once.
251: (0)            def _register_conversion_cache_cleaner_once():
252: (4)                atexit.register(_clean_conversion_cache)
253: (0)            def crop_to_same(actual_path, actual_image, expected_path, expected_image):
254: (4)                if actual_path[-7:-4] == 'eps' and expected_path[-7:-4] == 'pdf':
255: (8)                    aw, ah, ad = actual_image.shape
256: (8)                    ew, eh, ed = expected_image.shape
257: (8)                    actual_image = actual_image[int(aw / 2 - ew / 2):int(
258: (12)                       aw / 2 + ew / 2), int(ah / 2 - eh / 2):int(ah / 2 + eh / 2)]
259: (4)                return actual_image, expected_image
260: (0)            def calculate_rms(expected_image, actual_image):
261: (4)                """
262: (4)                Calculate the per-pixel errors, then compute the root mean square error.
263: (4)                """
264: (4)                if expected_image.shape != actual_image.shape:
265: (8)                    raise ImageComparisonFailure(
266: (12)                       f"Image sizes do not match expected size: {expected_image.shape} "
267: (12)                       f"actual size {actual_image.shape}")
268: (4)                return np.sqrt(((expected_image - actual_image).astype(float) **
2).mean())
269: (0)            def _load_image(path):
270: (4)                img = Image.open(path)
271: (4)                if img.mode != "RGBA" or img.getextrema()[3][0] == 255:
272: (8)                    img = img.convert("RGB")
273: (4)                return np.asarray(img)
274: (0)            def compare_images(expected, actual, tol, in_decorator=False):
275: (4)                """
276: (4)                Compare two "image" files checking differences within a tolerance.
277: (4)                The two given filenames may point to files which are convertible to
278: (4)                PNG via the `.converter` dictionary. The underlying RMS is calculated
279: (4)                with the `.calculate_rms` function.
280: (4)                Parameters
281: (4)                ----------
282: (4)                expected : str
283: (8)                    The filename of the expected image.
284: (4)                actual : str
285: (8)                    The filename of the actual image.
286: (4)                tol : float
287: (8)                    The tolerance (a color value difference, where 255 is the
288: (8)                    maximal difference).  The test fails if the average pixel
289: (8)                    difference is greater than this value.
290: (4)                in_decorator : bool
291: (8)                    Determines the output format. If called from image_comparison
292: (8)                    decorator, this should be True. (default=False)
293: (4)                Returns
294: (4)                -------
295: (4)                None or dict or str
296: (8)                    Return *None* if the images are equal within the given tolerance.
297: (8)                    If the images differ, the return value depends on  *in_decorator*.
298: (8)                    If *in_decorator* is true, a dict with the following entries is
299: (8)                    returned:
300: (8)                    - *rms*: The RMS of the image difference.
301: (8)                    - *expected*: The filename of the expected image.
302: (8)                    - *actual*: The filename of the actual image.
303: (8)                    - *diff_image*: The filename of the difference image.
304: (8)                    - *tol*: The comparison tolerance.
305: (8)                    Otherwise, a human-readable multi-line string representation of this
306: (8)                    information is returned.
307: (4)                Examples
308: (4)                --------
309: (4)                ::
```

```
310: (8)                    img1 = "./baseline/plot.png"
311: (8)                    img2 = "./output/plot.png"
312: (8)                    compare_images(img1, img2, 0.001)
313: (4)                """
314: (4)                actual = os.fspath(actual)
315: (4)                if not os.path.exists(actual):
316: (8)                    raise Exception(f"Output image {actual} does not exist.")
317: (4)                if os.stat(actual).st_size == 0:
318: (8)                    raise Exception(f"Output image file {actual} is empty.")
319: (4)                expected = os.fspath(expected)
320: (4)                if not os.path.exists(expected):
321: (8)                    raise OSError(f'Baseline image {expected!r} does not exist.')
322: (4)                extension = expected.split('.')[-1]
323: (4)                if extension != 'png':
324: (8)                    actual = convert(actual, cache=True)
325: (8)                    expected = convert(expected, cache=True)
326: (4)                expected_image = _load_image(expected)
327: (4)                actual_image = _load_image(actual)
328: (4)                actual_image, expected_image = crop_to_same(
329: (8)                    actual, actual_image, expected, expected_image)
330: (4)                diff_image = make_test_filename(actual, 'failed-diff')
331: (4)                if tol <= 0:
332: (8)                    if np.array_equal(expected_image, actual_image):
333: (12)                       return None
334: (4)                expected_image = expected_image.astype(np.int16)
335: (4)                actual_image = actual_image.astype(np.int16)
336: (4)                rms = calculate_rms(expected_image, actual_image)
337: (4)                if rms <= tol:
338: (8)                    return None
339: (4)                save_diff_image(expected, actual, diff_image)
340: (4)                results = dict(rms=rms, expected=str(expected),
341: (19)                              actual=str(actual), diff=str(diff_image), tol=tol)
342: (4)                if not in_decorator:
343: (8)                    template = ['Error: Image files did not match.',
344: (20)                               'RMS Value: {rms}',
345: (20)                               'Expected:  \n    {expected}',
346: (20)                               'Actual:    \n    {actual}',
347: (20)                               'Difference:\n    {diff}',
348: (20)                               'Tolerance: \n    {tol}', ]
349: (8)                    results = '\n  '.join([line.format(**results) for line in template])
350: (4)                return results
351: (0)            def save_diff_image(expected, actual, output):
352: (4)                """
353: (4)                Parameters
354: (4)                ----------
355: (4)                expected : str
356: (8)                    File path of expected image.
357: (4)                actual : str
358: (8)                    File path of actual image.
359: (4)                output : str
360: (8)                    File path to save difference image to.
361: (4)                """
362: (4)                expected_image = _load_image(expected)
363: (4)                actual_image = _load_image(actual)
364: (4)                actual_image, expected_image = crop_to_same(
365: (8)                    actual, actual_image, expected, expected_image)
366: (4)                expected_image = np.array(expected_image, float)
367: (4)                actual_image = np.array(actual_image, float)
368: (4)                if expected_image.shape != actual_image.shape:
369: (8)                    raise ImageComparisonFailure(
370: (12)                       f"Image sizes do not match expected size: {expected_image.shape} "
371: (12)                       f"actual size {actual_image.shape}")
372: (4)                abs_diff = np.abs(expected_image - actual_image)
373: (4)                abs_diff *= 10
374: (4)                abs_diff = np.clip(abs_diff, 0, 255).astype(np.uint8)
375: (4)                if abs_diff.shape[2] == 4:  # Hard-code the alpha channel to fully solid
376: (8)                    abs_diff[:, :, 3] = 255
377: (4)                Image.fromarray(abs_diff).save(output, format="png")
```

```
----------------------------------------

File 128 - _markers.py:

1: (0)              """
2: (0)              pytest markers for the internal Matplotlib test suite.
3: (0)              """
4: (0)              import logging
5: (0)              import shutil
6: (0)              import pytest
7: (0)              import matplotlib.testing
8: (0)              import matplotlib.testing.compare
9: (0)              from matplotlib import _get_executable_info, ExecutableNotFoundError
10: (0)             _log = logging.getLogger(__name__)
11: (0)             def _checkdep_usetex() -> bool:
12: (4)                 if not shutil.which("tex"):
13: (8)                     _log.warning("usetex mode requires TeX.")
14: (8)                     return False
15: (4)                 try:
16: (8)                     _get_executable_info("dvipng")
17: (4)                 except ExecutableNotFoundError:
18: (8)                     _log.warning("usetex mode requires dvipng.")
19: (8)                     return False
20: (4)                 try:
21: (8)                     _get_executable_info("gs")
22: (4)                 except ExecutableNotFoundError:
23: (8)                     _log.warning("usetex mode requires ghostscript.")
24: (8)                     return False
25: (4)                 return True
26: (0)             needs_ghostscript = pytest.mark.skipif(
27: (4)                 "eps" not in matplotlib.testing.compare.converter,
28: (4)                 reason="This test needs a ghostscript installation")
29: (0)             needs_pgf_lualatex = pytest.mark.skipif(
30: (4)                 not matplotlib.testing._check_for_pgf('lualatex'),
31: (4)                 reason='lualatex + pgf is required')
32: (0)             needs_pgf_pdflatex = pytest.mark.skipif(
33: (4)                 not matplotlib.testing._check_for_pgf('pdflatex'),
34: (4)                 reason='pdflatex + pgf is required')
35: (0)             needs_pgf_xelatex = pytest.mark.skipif(
36: (4)                 not matplotlib.testing._check_for_pgf('xelatex'),
37: (4)                 reason='xelatex + pgf is required')
38: (0)             needs_usetex = pytest.mark.skipif(
39: (4)                 not _checkdep_usetex(),
40: (4)                 reason="This test needs a TeX installation")

----------------------------------------

File 129 - conftest.py:

1: (0)              import pytest
2: (0)              import sys
3: (0)              import matplotlib
4: (0)              from matplotlib import _api
5: (0)              def pytest_configure(config):
6: (4)                  for key, value in [
7: (8)                      ("markers", "flaky: (Provided by pytest-rerunfailures.)"),
8: (8)                      ("markers", "timeout: (Provided by pytest-timeout.)"),
9: (8)                      ("markers", "backend: Set alternate Matplotlib backend temporarily."),
10: (8)                     ("markers", "baseline_images: Compare output against references."),
11: (8)                     ("markers", "pytz: Tests that require pytz to be installed."),
12: (8)                     ("filterwarnings", "error"),
13: (8)                     ("filterwarnings",
14: (9)                      "ignore:.*The py23 module has been deprecated:DeprecationWarning"),
15: (8)                     ("filterwarnings",
16: (9)                      r"ignore:DynamicImporter.find_spec\(\) not found; "
17: (9)                      r"falling back to find_module\(\):ImportWarning"),
18: (4)                 ]:
19: (8)                     config.addinivalue_line(key, value)
20: (4)                 matplotlib.use('agg', force=True)
```

```
21: (4)                 matplotlib._called_from_pytest = True
22: (4)                 matplotlib._init_tests()
23: (0)             def pytest_unconfigure(config):
24: (4)                 matplotlib._called_from_pytest = False
25: (0)             @pytest.fixture(autouse=True)
26: (0)             def mpl_test_settings(request):
27: (4)                 from matplotlib.testing.decorators import _cleanup_cm
28: (4)                 with _cleanup_cm():
29: (8)                     backend = None
30: (8)                     backend_marker = request.node.get_closest_marker('backend')
31: (8)                     prev_backend = matplotlib.get_backend()
32: (8)                     if backend_marker is not None:
33: (12)                        assert len(backend_marker.args) == 1, \
34: (16)                            "Marker 'backend' must specify 1 backend."
35: (12)                        backend, = backend_marker.args
36: (12)                        skip_on_importerror = backend_marker.kwargs.get(
37: (16)                            'skip_on_importerror', False)
38: (12)                        if backend.lower().startswith('qt5'):
39: (16)                            if any(sys.modules.get(k) for k in ('PyQt4', 'PySide')):
40: (20)                                pytest.skip('Qt4 binding already imported')
41: (8)                     matplotlib.testing.setup()
42: (8)                     with _api.suppress_matplotlib_deprecation_warning():
43: (12)                        if backend is not None:
44: (16)                            import matplotlib.pyplot as plt
45: (16)                            try:
46: (20)                                plt.switch_backend(backend)
47: (16)                            except ImportError as exc:
48: (20)                                if 'cairo' in backend.lower() or skip_on_importerror:
49: (24)                                    pytest.skip("Failed to switch to backend "
50: (36)                                                f"{backend} ({exc}).")
51: (20)                                else:
52: (24)                                    raise
53: (12)                        matplotlib.style.use(["classic", "_classic_test_patch"])
54: (8)                     try:
55: (12)                        yield
56: (8)                     finally:
57: (12)                        if backend is not None:
58: (16)                            plt.close("all")
59: (16)                            matplotlib.use(prev_backend)
60: (0)             @pytest.fixture
61: (0)             def pd():
62: (4)                 """Fixture to import and configure pandas."""
63: (4)                 pd = pytest.importorskip('pandas')
64: (4)                 try:
65: (8)                     from pandas.plotting import (
66: (12)                        deregister_matplotlib_converters as deregister)
67: (8)                     deregister()
68: (4)                 except ImportError:
69: (8)                     pass
70: (4)                 return pd
71: (0)             @pytest.fixture
72: (0)             def xr():
73: (4)                 """Fixture to import xarray."""
74: (4)                 xr = pytest.importorskip('xarray')
75: (4)                 return xr
```

File 130 - decorators.py:

```
1: (0)              import contextlib
2: (0)              import functools
3: (0)              import inspect
4: (0)              import os
5: (0)              from platform import uname
6: (0)              from pathlib import Path
7: (0)              import shutil
8: (0)              import string
9: (0)              import sys
```

```
(0)  import warnings
(0)  from packaging.version import parse as parse_version
(0)  import matplotlib.style
(0)  import matplotlib.units
(0)  import matplotlib.testing
(0)  from matplotlib import _pylab_helpers, cbook, ft2font, pyplot as plt, ticker
(0)  from .compare import comparable_formats, compare_images, make_test_filename
(0)  from .exceptions import ImageComparisonFailure
(0)  @contextlib.contextmanager
(0)  def _cleanup_cm():
(4)      orig_units_registry = matplotlib.units.registry.copy()
(4)      try:
(8)          with warnings.catch_warnings(), matplotlib.rc_context():
(12)             yield
(4)      finally:
(8)          matplotlib.units.registry.clear()
(8)          matplotlib.units.registry.update(orig_units_registry)
(8)          plt.close("all")
(0)  def _check_freetype_version(ver):
(4)      if ver is None:
(8)          return True
(4)      if isinstance(ver, str):
(8)          ver = (ver, ver)
(4)      ver = [parse_version(x) for x in ver]
(4)      found = parse_version(ft2font.__freetype_version__)
(4)      return ver[0] <= found <= ver[1]
(0)  def _checked_on_freetype_version(required_freetype_version):
(4)      import pytest
(4)      return pytest.mark.xfail(
(8)          not _check_freetype_version(required_freetype_version),
(8)          reason=f"Mismatched version of freetype. "
(15)                 f"Test requires '{required_freetype_version}', "
(15)                 f"you have '{ft2font.__freetype_version__}'",
(8)          raises=ImageComparisonFailure, strict=False)
(0)  def remove_ticks_and_titles(figure):
(4)      figure.suptitle("")
(4)      null_formatter = ticker.NullFormatter()
(4)      def remove_ticks(ax):
(8)          """Remove ticks in *ax* and all its child Axes."""
(8)          ax.set_title("")
(8)          ax.xaxis.set_major_formatter(null_formatter)
(8)          ax.xaxis.set_minor_formatter(null_formatter)
(8)          ax.yaxis.set_major_formatter(null_formatter)
(8)          ax.yaxis.set_minor_formatter(null_formatter)
(8)          try:
(12)             ax.zaxis.set_major_formatter(null_formatter)
(12)             ax.zaxis.set_minor_formatter(null_formatter)
(8)          except AttributeError:
(12)             pass
(8)          for child in ax.child_axes:
(12)             remove_ticks(child)
(4)      for ax in figure.get_axes():
(8)          remove_ticks(ax)
(0)  @contextlib.contextmanager
(0)  def _collect_new_figures():
(4)      """
(4)      After::
(8)          with _collect_new_figures() as figs:
(12)             some_code()
(4)      the list *figs* contains the figures that have been created during the
(4)      execution of ``some_code``, sorted by figure number.
(4)      """
(4)      managers = _pylab_helpers.Gcf.figs
(4)      preexisting = [manager for manager in managers.values()]
(4)      new_figs = []
(4)      try:
(8)          yield new_figs
(4)      finally:
(8)          new_managers = sorted([manager for manager in managers.values()
```

```
79: (31)                                         if manager not in preexisting],
80: (30)                                        key=lambda manager: manager.num)
81: (8)                     new_figs[:] = [manager.canvas.figure for manager in new_managers]
82: (0)             def _raise_on_image_difference(expected, actual, tol):
83: (4)                 __tracebackhide__ = True
84: (4)                 err = compare_images(expected, actual, tol, in_decorator=True)
85: (4)                 if err:
86: (8)                     for key in ["actual", "expected", "diff"]:
87: (12)                        err[key] = os.path.relpath(err[key])
88: (8)                     raise ImageComparisonFailure(
89: (12)                        ('images not close (RMS %(rms).3f):'
90: (16)                            '\n\t%(actual)s\n\t%(expected)s\n\t%(diff)s') % err)
91: (0)             class _ImageComparisonBase:
92: (4)                 """
93: (4)                 Image comparison base class
94: (4)                 This class provides *just* the comparison-related functionality and avoids
95: (4)                 any code that would be specific to any testing framework.
96: (4)                 """
97: (4)                 def __init__(self, func, tol, remove_text, savefig_kwargs):
98: (8)                     self.func = func
99: (8)                     self.baseline_dir, self.result_dir = _image_directories(func)
100: (8)                    self.tol = tol
101: (8)                    self.remove_text = remove_text
102: (8)                    self.savefig_kwargs = savefig_kwargs
103: (4)                def copy_baseline(self, baseline, extension):
104: (8)                    baseline_path = self.baseline_dir / baseline
105: (8)                    orig_expected_path = baseline_path.with_suffix(f'.{extension}')
106: (8)                    if extension == 'eps' and not orig_expected_path.exists():
107: (12)                       orig_expected_path = orig_expected_path.with_suffix('.pdf')
108: (8)                    expected_fname = make_test_filename(
109: (12)                       self.result_dir / orig_expected_path.name, 'expected')
110: (8)                    try:
111: (12)                       with contextlib.suppress(OSError):
112: (16)                           os.remove(expected_fname)
113: (12)                       try:
114: (16)                           if 'microsoft' in uname().release.lower():
115: (20)                               raise OSError  # On WSL, symlink breaks silently
116: (16)                           os.symlink(orig_expected_path, expected_fname)
117: (12)                       except OSError:  # On Windows, symlink *may* be unavailable.
118: (16)                           shutil.copyfile(orig_expected_path, expected_fname)
119: (8)                    except OSError as err:
120: (12)                       raise ImageComparisonFailure(
121: (16)                           f"Missing baseline image {expected_fname} because the "
122: (16)                           f"following file cannot be accessed: "
123: (16)                           f"{orig_expected_path}") from err
124: (8)                    return expected_fname
125: (4)                def compare(self, fig, baseline, extension, *, _lock=False):
126: (8)                    __tracebackhide__ = True
127: (8)                    if self.remove_text:
128: (12)                       remove_ticks_and_titles(fig)
129: (8)                    actual_path = (self.result_dir / baseline).with_suffix(f'.
{extension}')
130: (8)                    kwargs = self.savefig_kwargs.copy()
131: (8)                    if extension == 'pdf':
132: (12)                       kwargs.setdefault('metadata',
133: (30)                                        {'Creator': None, 'Producer': None,
134: (31)                                         'CreationDate': None})
135: (8)                    lock = (cbook._lock_path(actual_path)
136: (16)                           if _lock else contextlib.nullcontext())
137: (8)                    with lock:
138: (12)                       try:
139: (16)                           fig.savefig(actual_path, **kwargs)
140: (12)                       finally:
141: (16)                           plt.close(fig)
142: (12)                       expected_path = self.copy_baseline(baseline, extension)
143: (12)                       _raise_on_image_difference(expected_path, actual_path, self.tol)
144: (0)            def _pytest_image_comparison(baseline_images, extensions, tol,
145: (29)                                        freetype_version, remove_text, savefig_kwargs,
146: (29)                                        style):
```

```
147: (4)                """
148: (4)                Decorate function with image comparison for pytest.
149: (4)                This function creates a decorator that wraps a figure-generating function
150: (4)                with image comparison code.
151: (4)                """
152: (4)                import pytest
153: (4)                KEYWORD_ONLY = inspect.Parameter.KEYWORD_ONLY
154: (4)                def decorator(func):
155: (8)                    old_sig = inspect.signature(func)
156: (8)                    @functools.wraps(func)
157: (8)                    @pytest.mark.parametrize('extension', extensions)
158: (8)                    @matplotlib.style.context(style)
159: (8)                    @_checked_on_freetype_version(freetype_version)
160: (8)                    @functools.wraps(func)
161: (8)                    def wrapper(*args, extension, request, **kwargs):
162: (12)                       __tracebackhide__ = True
163: (12)                       if 'extension' in old_sig.parameters:
164: (16)                           kwargs['extension'] = extension
165: (12)                       if 'request' in old_sig.parameters:
166: (16)                           kwargs['request'] = request
167: (12)                       if extension not in comparable_formats():
168: (16)                           reason = {
169: (20)                               'pdf': 'because Ghostscript is not installed',
170: (20)                               'eps': 'because Ghostscript is not installed',
171: (20)                               'svg': 'because Inkscape is not installed',
172: (16)                           }.get(extension, 'on this system')
173: (16)                           pytest.skip(f"Cannot compare {extension} files {reason}")
174: (12)                       img = _ImageComparisonBase(func, tol=tol, remove_text=remove_text,
175: (39)                                                  savefig_kwargs=savefig_kwargs)
176: (12)                       matplotlib.testing.set_font_settings_for_testing()
177: (12)                       with _collect_new_figures() as figs:
178: (16)                           func(*args, **kwargs)
179: (12)                       needs_lock = any(
180: (16)                           marker.args[0] != 'extension'
181: (16)                           for marker in request.node.iter_markers('parametrize'))
182: (12)                       if baseline_images is not None:
183: (16)                           our_baseline_images = baseline_images
184: (12)                       else:
185: (16)                           our_baseline_images = request.getfixturevalue(
186: (20)                               'baseline_images')
187: (12)                       assert len(figs) == len(our_baseline_images), (
188: (16)                           f"Test generated {len(figs)} images but there are "
189: (16)                           f"{len(our_baseline_images)} baseline images")
190: (12)                       for fig, baseline in zip(figs, our_baseline_images):
191: (16)                           img.compare(fig, baseline, extension, _lock=needs_lock)
192: (8)                    parameters = list(old_sig.parameters.values())
193: (8)                    if 'extension' not in old_sig.parameters:
194: (12)                       parameters += [inspect.Parameter('extension', KEYWORD_ONLY)]
195: (8)                    if 'request' not in old_sig.parameters:
196: (12)                       parameters += [inspect.Parameter("request", KEYWORD_ONLY)]
197: (8)                    new_sig = old_sig.replace(parameters=parameters)
198: (8)                    wrapper.__signature__ = new_sig
199: (8)                    new_marks = getattr(func, 'pytestmark', []) + wrapper.pytestmark
200: (8)                    wrapper.pytestmark = new_marks
201: (8)                    return wrapper
202: (4)                return decorator
203: (0)            def image_comparison(baseline_images, extensions=None, tol=0,
204: (21)                                freetype_version=None, remove_text=False,
205: (21)                                savefig_kwarg=None,
206: (21)                                style=("classic", "_classic_test_patch")):
207: (4)                """
208: (4)                Compare images generated by the test with those specified in
209: (4)                *baseline_images*, which must correspond, else an `ImageComparisonFailure`
210: (4)                exception will be raised.
211: (4)                Parameters
212: (4)                ----------
213: (4)                baseline_images : list or None
214: (8)                    A list of strings specifying the names of the images generated by
215: (8)                    calls to `.Figure.savefig`.
```

```
(8)                 If *None*, the test function must use the ``baseline_images`` fixture,
(8)                 either as a parameter or with `pytest.mark.usefixtures`. This value is
(8)                 only allowed when using pytest.
(4)             extensions : None or list of str
(8)                 The list of extensions to test, e.g. ``['png', 'pdf']``.
(8)                 If *None*, defaults to all supported extensions: png, pdf, and svg.
(8)                 When testing a single extension, it can be directly included in the
(8)                 names passed to *baseline_images*.  In that case, *extensions* must
not
(8)                 be set.
(8)                 In order to keep the size of the test suite from ballooning, we only
(8)                 include the ``svg`` or ``pdf`` outputs if the test is explicitly
(8)                 exercising a feature dependent on that backend (see also the
(8)                 `check_figures_equal` decorator for that purpose).
(4)             tol : float, default: 0
(8)                 The RMS threshold above which the test is considered failed.
(8)                 Due to expected small differences in floating-point calculations, on
(8)                 32-bit systems an additional 0.06 is added to this threshold.
(4)             freetype_version : str or tuple
(8)                 The expected freetype version or range of versions for this test to
(8)                 pass.
(4)             remove_text : bool
(8)                 Remove the title and tick text from the figure before comparison.
This
(8)                 is useful to make the baseline images independent of variations in
text
(8)                 rendering between different versions of FreeType.
(8)                 This does not remove other, more deliberate, text, such as legends and
(8)                 annotations.
(4)             savefig_kwarg : dict
(8)                 Optional arguments that are passed to the savefig method.
(4)             style : str, dict, or list
(8)                 The optional style(s) to apply to the image test. The test itself
(8)                 can also apply additional styles if desired. Defaults to ``["classic",
(8)                 "_classic_test_patch"]``.
(4)             """
(4)             if baseline_images is not None:
(8)                 baseline_exts = [*filter(None, {Path(baseline).suffix[1:]
(40)                                             for baseline in baseline_images})]
(8)                 if baseline_exts:
(12)                    if extensions is not None:
(16)                        raise ValueError(
(20)                            "When including extensions directly in 'baseline_images',
"
(20)                            "'extensions' cannot be set as well")
(12)                    if len(baseline_exts) > 1:
(16)                        raise ValueError(
(20)                            "When including extensions directly in 'baseline_images',
"
(20)                            "all baselines must share the same suffix")
(12)                    extensions = baseline_exts
(12)                    baseline_images = [  # Chop suffix out from baseline_images.
(16)                        Path(baseline).stem for baseline in baseline_images]
(4)             if extensions is None:
(8)                 extensions = ['png', 'pdf', 'svg']
(4)             if savefig_kwarg is None:
(8)                 savefig_kwarg = dict()  # default no kwargs to savefig
(4)             if sys.maxsize <= 2**32:
(8)                 tol += 0.06
(4)             return _pytest_image_comparison(
(8)                 baseline_images=baseline_images, extensions=extensions, tol=tol,
(8)                 freetype_version=freetype_version, remove_text=remove_text,
(8)                 savefig_kwargs=savefig_kwarg, style=style)
(0)         def check_figures_equal(*, extensions=("png", "pdf", "svg"), tol=0):
(4)             """
(4)             Decorator for test cases that generate and compare two figures.
(4)             The decorated function must take two keyword arguments, *fig_test*
(4)             and *fig_ref*, and draw the test and reference images on them.
(4)             After the function returns, the figures are saved and compared.
```

```
(4)                 This decorator should be preferred over `image_comparison` when possible in
(4)                 order to keep the size of the test suite from ballooning.
(4)                 Parameters
(4)                 ----------
(4)                 extensions : list, default: ["png", "pdf", "svg"]
(8)                     The extensions to test.
(4)                 tol : float
(8)                     The RMS threshold above which the test is considered failed.
(4)                 Raises
(4)                 ------
(4)                 RuntimeError
(8)                     If any new figures are created (and not subsequently closed) inside
(8)                     the test function.
(4)                 Examples
(4)                 --------
(4)                 Check that calling `.Axes.plot` with a single argument plots it against
(4)                 ``[0, 1, 2, ...]``::
(8)                     @check_figures_equal()
(8)                     def test_plot(fig_test, fig_ref):
(12)                        fig_test.subplots().plot([1, 3, 5])
(12)                        fig_ref.subplots().plot([0, 1, 2], [1, 3, 5])
(4)                 """
(4)                 ALLOWED_CHARS = set(string.digits + string.ascii_letters + '_-[]()')
(4)                 KEYWORD_ONLY = inspect.Parameter.KEYWORD_ONLY
(4)                 def decorator(func):
(8)                     import pytest
(8)                     _, result_dir = _image_directories(func)
(8)                     old_sig = inspect.signature(func)
(8)                     if not {"fig_test", "fig_ref"}.issubset(old_sig.parameters):
(12)                        raise ValueError("The decorated function must have at least the "
(29)                                         "parameters 'fig_test' and 'fig_ref', but your "
(29)                                         f"function has the signature {old_sig}")
(8)                     @pytest.mark.parametrize("ext", extensions)
(8)                     def wrapper(*args, ext, request, **kwargs):
(12)                        if 'ext' in old_sig.parameters:
(16)                            kwargs['ext'] = ext
(12)                        if 'request' in old_sig.parameters:
(16)                            kwargs['request'] = request
(12)                        file_name = "".join(c for c in request.node.name
(32)                                            if c in ALLOWED_CHARS)
(12)                        try:
(16)                            fig_test = plt.figure("test")
(16)                            fig_ref = plt.figure("reference")
(16)                            with _collect_new_figures() as figs:
(20)                                func(*args, fig_test=fig_test, fig_ref=fig_ref, **kwargs)
(16)                            if figs:
(20)                                raise RuntimeError('Number of open figures changed during '
(39)                                                   'test. Make sure you are plotting to '
(39)                                                   'fig_test or fig_ref, or if this is '
(39)                                                   'deliberate explicitly close the '
(39)                                                   'new figure(s) inside the test.')
(16)                            test_image_path = result_dir / (file_name + "." + ext)
(16)                            ref_image_path = result_dir / (file_name + "-expected." + ext)
(16)                            fig_test.savefig(test_image_path)
(16)                            fig_ref.savefig(ref_image_path)
(16)                            _raise_on_image_difference(
(20)                                ref_image_path, test_image_path, tol=tol
(16)                            )
(12)                        finally:
(16)                            plt.close(fig_test)
(16)                            plt.close(fig_ref)
(8)                     parameters = [
(12)                        param
(12)                        for param in old_sig.parameters.values()
(12)                        if param.name not in {"fig_test", "fig_ref"}
(8)                     ]
(8)                     if 'ext' not in old_sig.parameters:
```

```
347: (12)                        parameters += [inspect.Parameter("ext", KEYWORD_ONLY)]
348: (8)                     if 'request' not in old_sig.parameters:
349: (12)                        parameters += [inspect.Parameter("request", KEYWORD_ONLY)]
350: (8)                     new_sig = old_sig.replace(parameters=parameters)
351: (8)                     wrapper.__signature__ = new_sig
352: (8)                     new_marks = getattr(func, "pytestmark", []) + wrapper.pytestmark
353: (8)                     wrapper.pytestmark = new_marks
354: (8)                     return wrapper
355: (4)                 return decorator
356: (0)             def _image_directories(func):
357: (4)                 """
358: (4)                 Compute the baseline and result image directories for testing *func*.
359: (4)                 For test module ``foo.bar.test_baz``, the baseline directory is at
360: (4)                 ``foo/bar/baseline_images/test_baz`` and the result directory at
361: (4)                 ``$(pwd)/result_images/test_baz``.  The result directory is created if it
362: (4)                 doesn't exist.
363: (4)                 """
364: (4)                 module_path = Path(inspect.getfile(func))
365: (4)                 baseline_dir = module_path.parent / "baseline_images" / module_path.stem
366: (4)                 result_dir = Path().resolve() / "result_images" / module_path.stem
367: (4)                 result_dir.mkdir(parents=True, exist_ok=True)
368: (4)                 return baseline_dir, result_dir
```

File 131 - exceptions.py:

```
1: (0)              class ImageComparisonFailure(AssertionError):
2: (4)                  """
3: (4)                  Raise this exception to mark a test as a comparison between two images.
4: (4)                  """
```

File 132 - widgets.py:

```
1: (0)              """
2: (0)              ========================
3: (0)              Widget testing utilities
4: (0)              ========================
5: (0)              See also :mod:`matplotlib.tests.test_widgets`.
6: (0)              """
7: (0)              from unittest import mock
8: (0)              import matplotlib.pyplot as plt
9: (0)              def get_ax():
10: (4)                 """Create a plot and return its Axes."""
11: (4)                 fig, ax = plt.subplots(1, 1)
12: (4)                 ax.plot([0, 200], [0, 200])
13: (4)                 ax.set_aspect(1.0)
14: (4)                 ax.figure.canvas.draw()
15: (4)                 return ax
16: (0)             def noop(*args, **kwargs):
17: (4)                 pass
18: (0)             def mock_event(ax, button=1, xdata=0, ydata=0, key=None, step=1):
19: (4)                 r"""
20: (4)                 Create a mock event that can stand in for `.Event` and its subclasses.
21: (4)                 This event is intended to be used in tests where it can be passed into
22: (4)                 event handling functions.
23: (4)                 Parameters
24: (4)                 ----------
25: (4)                 ax : `~matplotlib.axes.Axes`
26: (8)                     The Axes the event will be in.
27: (4)                 xdata : float
28: (8)                     x coord of mouse in data coords.
29: (4)                 ydata : float
30: (8)                     y coord of mouse in data coords.
31: (4)                 button : None or `MouseButton` or {'up', 'down'}
32: (8)                     The mouse button pressed in this event (see also `.MouseEvent`).
33: (4)                 key : None or str
```

```
 34: (8)                    The key pressed when the mouse event triggered (see also `.KeyEvent`).
 35: (4)                step : int
 36: (8)                    Number of scroll steps (positive for 'up', negative for 'down').
 37: (4)                Returns
 38: (4)                -------
 39: (4)                event
 40: (8)                    A `.Event`\-like Mock instance.
 41: (4)                """
 42: (4)                event = mock.Mock()
 43: (4)                event.button = button
 44: (4)                event.x, event.y = ax.transData.transform([(xdata, ydata),
 45: (47)                                                          (xdata, ydata)])[0]
 46: (4)                event.xdata, event.ydata = xdata, ydata
 47: (4)                event.inaxes = ax
 48: (4)                event.canvas = ax.figure.canvas
 49: (4)                event.key = key
 50: (4)                event.step = step
 51: (4)                event.guiEvent = None
 52: (4)                event.name = 'Custom'
 53: (4)                return event
 54: (0)            def do_event(tool, etype, button=1, xdata=0, ydata=0, key=None, step=1):
 55: (4)                """
 56: (4)                Trigger an event on the given tool.
 57: (4)                Parameters
 58: (4)                ----------
 59: (4)                tool : matplotlib.widgets.AxesWidget
 60: (4)                etype : str
 61: (8)                    The event to trigger.
 62: (4)                xdata : float
 63: (8)                    x coord of mouse in data coords.
 64: (4)                ydata : float
 65: (8)                    y coord of mouse in data coords.
 66: (4)                button : None or `MouseButton` or {'up', 'down'}
 67: (8)                    The mouse button pressed in this event (see also `.MouseEvent`).
 68: (4)                key : None or str
 69: (8)                    The key pressed when the mouse event triggered (see also `.KeyEvent`).
 70: (4)                step : int
 71: (8)                    Number of scroll steps (positive for 'up', negative for 'down').
 72: (4)                """
 73: (4)                event = mock_event(tool.ax, button, xdata, ydata, key, step)
 74: (4)                func = getattr(tool, etype)
 75: (4)                func(event)
 76: (0)            def click_and_drag(tool, start, end, key=None):
 77: (4)                """
 78: (4)                Helper to simulate a mouse drag operation.
 79: (4)                Parameters
 80: (4)                ----------
 81: (4)                tool : `~matplotlib.widgets.Widget`
 82: (4)                start : [float, float]
 83: (8)                    Starting point in data coordinates.
 84: (4)                end : [float, float]
 85: (8)                    End point in data coordinates.
 86: (4)                key : None or str
 87: (9)                     An optional key that is pressed during the whole operation
 88: (9)                     (see also `.KeyEvent`).
 89: (4)                """
 90: (4)                if key is not None:
 91: (8)                    do_event(tool, 'on_key_press', xdata=start[0], ydata=start[1],
 92: (17)                            button=1, key=key)
 93: (4)                do_event(tool, 'press', xdata=start[0], ydata=start[1], button=1)
 94: (4)                do_event(tool, 'onmove', xdata=end[0], ydata=end[1], button=1)
 95: (4)                do_event(tool, 'release', xdata=end[0], ydata=end[1], button=1)
 96: (4)                if key is not None:
 97: (8)                    do_event(tool, 'on_key_release', xdata=end[0], ydata=end[1],
 98: (17)                            button=1, key=key)

----------------------------------------

File 133 - __init__.py:
```

```
1: (0)              """
2: (0)              A sample set of units for use with testing unit conversion
3: (0)              of Matplotlib routines.  These are used because they use very strict
4: (0)              enforcement of unitized data which will test the entire spectrum of how
5: (0)              unitized data might be used (it is not always meaningful to convert to
6: (0)              a float without specific units given).
7: (0)              UnitDbl is essentially a unitized floating point number.  It has a
8: (0)              minimal set of supported units (enough for testing purposes).  All
9: (0)              of the mathematical operation are provided to fully test any behaviour
10: (0)             that might occur with unitized data.  Remember that unitized data has
11: (0)             rules as to how it can be applied to one another (a value of distance
12: (0)             cannot be added to a value of time).  Thus we need to guard against any
13: (0)             accidental "default" conversion that will strip away the meaning of the
14: (0)             data and render it neutered.
15: (0)             Epoch is different than a UnitDbl of time.  Time is something that can be
16: (0)             measured where an Epoch is a specific moment in time.  Epochs are typically
17: (0)             referenced as an offset from some predetermined epoch.
18: (0)             A difference of two epochs is a Duration.  The distinction between a Duration
19: (0)             and a UnitDbl of time is made because an Epoch can have different frames (or
20: (0)             units).  In the case of our test Epoch class the two allowed frames are 'UTC'
21: (0)             and 'ET' (Note that these are rough estimates provided for testing purposes
22: (0)             and should not be used in production code where accuracy of time frames is
23: (0)             desired).  As such a Duration also has a frame of reference and therefore
needs
24: (0)             to be called out as different that a simple measurement of time since a delta-
t
25: (0)             in one frame may not be the same in another.
26: (0)             """
27: (0)             from .Duration import Duration
28: (0)             from .Epoch import Epoch
29: (0)             from .UnitDbl import UnitDbl
30: (0)             from .StrConverter import StrConverter
31: (0)             from .EpochConverter import EpochConverter
32: (0)             from .UnitDblConverter import UnitDblConverter
33: (0)             from .UnitDblFormatter import UnitDblFormatter
34: (0)             __version__ = "1.0"
35: (0)             __all__ = [
36: (12)                        'register',
37: (12)                        'Duration',
38: (12)                        'Epoch',
39: (12)                        'UnitDbl',
40: (12)                        'UnitDblFormatter',
41: (10)                    ]
42: (0)             def register():
43: (4)                 """Register the unit conversion classes with matplotlib."""
44: (4)                 import matplotlib.units as mplU
45: (4)                 mplU.registry[str] = StrConverter()
46: (4)                 mplU.registry[Epoch] = EpochConverter()
47: (4)                 mplU.registry[Duration] = EpochConverter()
48: (4)                 mplU.registry[UnitDbl] = UnitDblConverter()
49: (0)             m = UnitDbl(1.0, "m")
50: (0)             km = UnitDbl(1.0, "km")
51: (0)             mile = UnitDbl(1.0, "mile")
52: (0)             deg = UnitDbl(1.0, "deg")
53: (0)             rad = UnitDbl(1.0, "rad")
54: (0)             sec = UnitDbl(1.0, "sec")
55: (0)             min = UnitDbl(1.0, "min")
56: (0)             hr = UnitDbl(1.0, "hour")
57: (0)             day = UnitDbl(24.0, "hour")
58: (0)             sec = UnitDbl(1.0, "sec")
```

File 134 - Duration.py:

```
1: (0)              """Duration module."""
2: (0)              import functools
3: (0)              import operator
```

```
4: (0)              from matplotlib import _api
5: (0)              class Duration:
6: (4)                  """Class Duration in development."""
7: (4)                  allowed = ["ET", "UTC"]
8: (4)                  def __init__(self, frame, seconds):
9: (8)                      """
10: (8)                     Create a new Duration object.
11: (8)                     = ERROR CONDITIONS
12: (8)                     - If the input frame is not in the allowed list, an error is thrown.
13: (8)                     = INPUT VARIABLES
14: (8)                     - frame     The frame of the duration.  Must be 'ET' or 'UTC'
15: (8)                     - seconds  The number of seconds in the Duration.
16: (8)                     """
17: (8)                     _api.check_in_list(self.allowed, frame=frame)
18: (8)                     self._frame = frame
19: (8)                     self._seconds = seconds
20: (4)                 def frame(self):
21: (8)                     """Return the frame the duration is in."""
22: (8)                     return self._frame
23: (4)                 def __abs__(self):
24: (8)                     """Return the absolute value of the duration."""
25: (8)                     return Duration(self._frame, abs(self._seconds))
26: (4)                 def __neg__(self):
27: (8)                     """Return the negative value of this Duration."""
28: (8)                     return Duration(self._frame, -self._seconds)
29: (4)                 def seconds(self):
30: (8)                     """Return the number of seconds in the Duration."""
31: (8)                     return self._seconds
32: (4)                 def __bool__(self):
33: (8)                     return self._seconds != 0
34: (4)                 def _cmp(self, op, rhs):
35: (8)                     """
36: (8)                     Check that *self* and *rhs* share frames; compare them using *op*.
37: (8)                     """
38: (8)                     self.checkSameFrame(rhs, "compare")
39: (8)                     return op(self._seconds, rhs._seconds)
40: (4)                 __eq__ = functools.partialmethod(_cmp, operator.eq)
41: (4)                 __ne__ = functools.partialmethod(_cmp, operator.ne)
42: (4)                 __lt__ = functools.partialmethod(_cmp, operator.lt)
43: (4)                 __le__ = functools.partialmethod(_cmp, operator.le)
44: (4)                 __gt__ = functools.partialmethod(_cmp, operator.gt)
45: (4)                 __ge__ = functools.partialmethod(_cmp, operator.ge)
46: (4)                 def __add__(self, rhs):
47: (8)                     """
48: (8)                     Add two Durations.
49: (8)                     = ERROR CONDITIONS
50: (8)                     - If the input rhs is not in the same frame, an error is thrown.
51: (8)                     = INPUT VARIABLES
52: (8)                     - rhs     The Duration to add.
53: (8)                     = RETURN VALUE
54: (8)                     - Returns the sum of ourselves and the input Duration.
55: (8)                     """
56: (8)                     import matplotlib.testing.jpl_units as U
57: (8)                     if isinstance(rhs, U.Epoch):
58: (12)                        return rhs + self
59: (8)                     self.checkSameFrame(rhs, "add")
60: (8)                     return Duration(self._frame, self._seconds + rhs._seconds)
61: (4)                 def __sub__(self, rhs):
62: (8)                     """
63: (8)                     Subtract two Durations.
64: (8)                     = ERROR CONDITIONS
65: (8)                     - If the input rhs is not in the same frame, an error is thrown.
66: (8)                     = INPUT VARIABLES
67: (8)                     - rhs     The Duration to subtract.
68: (8)                     = RETURN VALUE
69: (8)                     - Returns the difference of ourselves and the input Duration.
70: (8)                     """
71: (8)                     self.checkSameFrame(rhs, "sub")
72: (8)                     return Duration(self._frame, self._seconds - rhs._seconds)
```

```
73: (4)                 def __mul__(self, rhs):
74: (8)                     """
75: (8)                     Scale a UnitDbl by a value.
76: (8)                     = INPUT VARIABLES
77: (8)                     - rhs     The scalar to multiply by.
78: (8)                     = RETURN VALUE
79: (8)                     - Returns the scaled Duration.
80: (8)                     """
81: (8)                     return Duration(self._frame, self._seconds * float(rhs))
82: (4)                 __rmul__ = __mul__
83: (4)                 def __str__(self):
84: (8)                     """Print the Duration."""
85: (8)                     return f"{self._seconds:g} {self._frame}"
86: (4)                 def __repr__(self):
87: (8)                     """Print the Duration."""
88: (8)                     return f"Duration('{self._frame}', {self._seconds:g})"
89: (4)                 def checkSameFrame(self, rhs, func):
90: (8)                     """
91: (8)                     Check to see if frames are the same.
92: (8)                     = ERROR CONDITIONS
93: (8)                     - If the frame of the rhs Duration is not the same as our frame,
94: (10)                      an error is thrown.
95: (8)                     = INPUT VARIABLES
96: (8)                     - rhs     The Duration to check for the same frame
97: (8)                     - func    The name of the function doing the check.
98: (8)                     """
99: (8)                     if self._frame != rhs._frame:
100: (12)                       raise ValueError(
101: (16)                           f"Cannot {func} Durations with different frames.\n"
102: (16)                           f"LHS: {self._frame}\n"
103: (16)                           f"RHS: {rhs._frame}")
```

File 135 - EpochConverter.py:

```
1: (0)              """EpochConverter module containing class EpochConverter."""
2: (0)              from matplotlib import cbook, units
3: (0)              import matplotlib.dates as date_ticker
4: (0)              __all__ = ['EpochConverter']
5: (0)              class EpochConverter(units.ConversionInterface):
6: (4)                  """
7: (4)                  Provides Matplotlib conversion functionality for Monte Epoch and Duration
8: (4)                  classes.
9: (4)                  """
10: (4)                 jdRef = 1721425.5
11: (4)                 @staticmethod
12: (4)                 def axisinfo(unit, axis):
13: (8)                     majloc = date_ticker.AutoDateLocator()
14: (8)                     majfmt = date_ticker.AutoDateFormatter(majloc)
15: (8)                     return units.AxisInfo(majloc=majloc, majfmt=majfmt, label=unit)
16: (4)                 @staticmethod
17: (4)                 def float2epoch(value, unit):
18: (8)                     """
19: (8)                     Convert a Matplotlib floating-point date into an Epoch of the
specified
20: (8)                     units.
21: (8)                     = INPUT VARIABLES
22: (8)                     - value     The Matplotlib floating-point date.
23: (8)                     - unit      The unit system to use for the Epoch.
24: (8)                     = RETURN VALUE
25: (8)                     - Returns the value converted to an Epoch in the specified time
system.
26: (8)                     """
27: (8)                     import matplotlib.testing.jpl_units as U
28: (8)                     secPastRef = value * 86400.0 * U.UnitDbl(1.0, 'sec')
29: (8)                     return U.Epoch(unit, secPastRef, EpochConverter.jdRef)
30: (4)                 @staticmethod
31: (4)                 def epoch2float(value, unit):
```

```
32: (8)                     """
33: (8)                     Convert an Epoch value to a float suitable for plotting as a python
34: (8)                     datetime object.
35: (8)                     = INPUT VARIABLES
36: (8)                     - value    An Epoch or list of Epochs that need to be converted.
37: (8)                     - unit     The units to use for an axis with Epoch data.
38: (8)                     = RETURN VALUE
39: (8)                     - Returns the value parameter converted to floats.
40: (8)                     """
41: (8)                     return value.julianDate(unit) - EpochConverter.jdRef
42: (4)                 @staticmethod
43: (4)                 def duration2float(value):
44: (8)                     """
45: (8)                     Convert a Duration value to a float suitable for plotting as a python
46: (8)                     datetime object.
47: (8)                     = INPUT VARIABLES
48: (8)                     - value    A Duration or list of Durations that need to be converted.
49: (8)                     = RETURN VALUE
50: (8)                     - Returns the value parameter converted to floats.
51: (8)                     """
52: (8)                     return value.seconds() / 86400.0
53: (4)                 @staticmethod
54: (4)                 def convert(value, unit, axis):
55: (8)                     import matplotlib.testing.jpl_units as U
56: (8)                     if not cbook.is_scalar_or_string(value):
57: (12)                        return [EpochConverter.convert(x, unit, axis) for x in value]
58: (8)                     if unit is None:
59: (12)                        unit = EpochConverter.default_units(value, axis)
60: (8)                     if isinstance(value, U.Duration):
61: (12)                        return EpochConverter.duration2float(value)
62: (8)                     else:
63: (12)                        return EpochConverter.epoch2float(value, unit)
64: (4)                 @staticmethod
65: (4)                 def default_units(value, axis):
66: (8)                     if cbook.is_scalar_or_string(value):
67: (12)                        return value.frame()
68: (8)                     else:
69: (12)                        return EpochConverter.default_units(value[0], axis)
```

File 136 - Epoch.py:

```
1: (0)              """Epoch module."""
2: (0)              import functools
3: (0)              import operator
4: (0)              import math
5: (0)              import datetime as DT
6: (0)              from matplotlib import _api
7: (0)              from matplotlib.dates import date2num
8: (0)              class Epoch:
9: (4)                  allowed = {
10: (8)                     "ET": {
11: (12)                        "UTC": +64.1839,
12: (12)                        },
13: (8)                     "UTC": {
14: (12)                        "ET": -64.1839,
15: (12)                        },
16: (8)                     }
17: (4)                 def __init__(self, frame, sec=None, jd=None, daynum=None, dt=None):
18: (8)                     """
19: (8)                     Create a new Epoch object.
20: (8)                     Build an epoch 1 of 2 ways:
21: (8)                     Using seconds past a Julian date:
22: (8)                     or using a matplotlib day number
23: (8)                     = ERROR CONDITIONS
24: (8)                     - If the input units are not in the allowed list, an error is thrown.
25: (8)                     = INPUT VARIABLES
26: (8)                     - frame     The frame of the epoch.  Must be 'ET' or 'UTC'
```

```
(8)                     - sec        The number of seconds past the input JD.
(8)                     - jd         The Julian date of the epoch.
(8)                     - daynum    The matplotlib day number of the epoch.
(8)                     - dt         A python datetime instance.
(8)                     """
(8)                     if ((sec is None and jd is not None) or
(16)                            (sec is not None and jd is None) or
(16)                            (daynum is not None and
(17)                             (sec is not None or jd is not None)) or
(16)                            (daynum is None and dt is None and
(17)                             (sec is None or jd is None)) or
(16)                            (daynum is not None and dt is not None) or
(16)                            (dt is not None and (sec is not None or jd is not None)) or
(16)                            ((dt is not None) and not isinstance(dt, DT.datetime))):
(12)                        raise ValueError(
(16)                            "Invalid inputs.  Must enter sec and jd together, "
(16)                            "daynum by itself, or dt (must be a python datetime).\n"
(16)                            "Sec = %s\n"
(16)                            "JD  = %s\n"
(16)                            "dnum= %s\n"
(16)                            "dt  = %s" % (sec, jd, daynum, dt))
(8)                     _api.check_in_list(self.allowed, frame=frame)
(8)                     self._frame = frame
(8)                     if dt is not None:
(12)                        daynum = date2num(dt)
(8)                     if daynum is not None:
(12)                        jd = float(daynum) + 1721425.5
(12)                        self._jd = math.floor(jd)
(12)                        self._seconds = (jd - self._jd) * 86400.0
(8)                     else:
(12)                        self._seconds = float(sec)
(12)                        self._jd = float(jd)
(12)                        deltaDays = math.floor(self._seconds / 86400)
(12)                        self._jd += deltaDays
(12)                        self._seconds -= deltaDays * 86400.0
(4)                 def convert(self, frame):
(8)                     if self._frame == frame:
(12)                        return self
(8)                     offset = self.allowed[self._frame][frame]
(8)                     return Epoch(frame, self._seconds + offset, self._jd)
(4)                 def frame(self):
(8)                     return self._frame
(4)                 def julianDate(self, frame):
(8)                     t = self
(8)                     if frame != self._frame:
(12)                        t = self.convert(frame)
(8)                     return t._jd + t._seconds / 86400.0
(4)                 def secondsPast(self, frame, jd):
(8)                     t = self
(8)                     if frame != self._frame:
(12)                        t = self.convert(frame)
(8)                     delta = t._jd - jd
(8)                     return t._seconds + delta * 86400
(4)                 def _cmp(self, op, rhs):
(8)                     """Compare Epochs *self* and *rhs* using operator *op*."""
(8)                     t = self
(8)                     if self._frame != rhs._frame:
(12)                        t = self.convert(rhs._frame)
(8)                     if t._jd != rhs._jd:
(12)                        return op(t._jd, rhs._jd)
(8)                     return op(t._seconds, rhs._seconds)
(4)                 __eq__ = functools.partialmethod(_cmp, operator.eq)
(4)                 __ne__ = functools.partialmethod(_cmp, operator.ne)
(4)                 __lt__ = functools.partialmethod(_cmp, operator.lt)
(4)                 __le__ = functools.partialmethod(_cmp, operator.le)
(4)                 __gt__ = functools.partialmethod(_cmp, operator.gt)
(4)                 __ge__ = functools.partialmethod(_cmp, operator.ge)
(4)                 def __add__(self, rhs):
(8)                     """
```

```
96: (8)                     Add a duration to an Epoch.
97: (8)                     = INPUT VARIABLES
98: (8)                     - rhs     The Epoch to subtract.
99: (8)                     = RETURN VALUE
100: (8)                    - Returns the difference of ourselves and the input Epoch.
101: (8)                    """
102: (8)                    t = self
103: (8)                    if self._frame != rhs.frame():
104: (12)                       t = self.convert(rhs._frame)
105: (8)                    sec = t._seconds + rhs.seconds()
106: (8)                    return Epoch(t._frame, sec, t._jd)
107: (4)                def __sub__(self, rhs):
108: (8)                    """
109: (8)                    Subtract two Epoch's or a Duration from an Epoch.
110: (8)                    Valid:
111: (8)                    Duration = Epoch - Epoch
112: (8)                    Epoch = Epoch - Duration
113: (8)                    = INPUT VARIABLES
114: (8)                    - rhs     The Epoch to subtract.
115: (8)                    = RETURN VALUE
116: (8)                    - Returns either the duration between to Epoch's or the a new
117: (10)                     Epoch that is the result of subtracting a duration from an epoch.
118: (8)                    """
119: (8)                    import matplotlib.testing.jpl_units as U
120: (8)                    if isinstance(rhs, U.Duration):
121: (12)                       return self + -rhs
122: (8)                    t = self
123: (8)                    if self._frame != rhs._frame:
124: (12)                       t = self.convert(rhs._frame)
125: (8)                    days = t._jd - rhs._jd
126: (8)                    sec = t._seconds - rhs._seconds
127: (8)                    return U.Duration(rhs._frame, days*86400 + sec)
128: (4)                def __str__(self):
129: (8)                    """Print the Epoch."""
130: (8)                    return f"{self.julianDate(self._frame):22.15e} {self._frame}"
131: (4)                def __repr__(self):
132: (8)                    """Print the Epoch."""
133: (8)                    return str(self)
134: (4)                @staticmethod
135: (4)                def range(start, stop, step):
136: (8)                    """
137: (8)                    Generate a range of Epoch objects.
138: (8)                    Similar to the Python range() method.  Returns the range [
139: (8)                    start, stop) at the requested step.  Each element will be a
140: (8)                    Epoch object.
141: (8)                    = INPUT VARIABLES
142: (8)                    - start     The starting value of the range.
143: (8)                    - stop      The stop value of the range.
144: (8)                    - step      Step to use.
145: (8)                    = RETURN VALUE
146: (8)                    - Returns a list containing the requested Epoch values.
147: (8)                    """
148: (8)                    elems = []
149: (8)                    i = 0
150: (8)                    while True:
151: (12)                       d = start + i * step
152: (12)                       if d >= stop:
153: (16)                           break
154: (12)                       elems.append(d)
155: (12)                       i += 1
156: (8)                    return elems

----------------------------------------

File 137 - StrConverter.py:

1: (0)              """StrConverter module containing class StrConverter."""
2: (0)              import numpy as np
3: (0)              import matplotlib.units as units
```

```
4: (0)              __all__ = ['StrConverter']
5: (0)              class StrConverter(units.ConversionInterface):
6: (4)                  """
7: (4)                  A Matplotlib converter class for string data values.
8: (4)                  Valid units for string are:
9: (4)                  - 'indexed' : Values are indexed as they are specified for plotting.
10: (4)                 - 'sorted'  : Values are sorted alphanumerically.
11: (4)                 - 'inverted' : Values are inverted so that the first value is on top.
12: (4)                 - 'sorted-inverted' :  A combination of 'sorted' and 'inverted'
13: (4)                 """
14: (4)                 @staticmethod
15: (4)                 def axisinfo(unit, axis):
16: (8)                     return None
17: (4)                 @staticmethod
18: (4)                 def convert(value, unit, axis):
19: (8)                     if value == []:
20: (12)                        return []
21: (8)                     ax = axis.axes
22: (8)                     if axis is ax.xaxis:
23: (12)                        isXAxis = True
24: (8)                     else:
25: (12)                        isXAxis = False
26: (8)                     axis.get_major_ticks()
27: (8)                     ticks = axis.get_ticklocs()
28: (8)                     labels = axis.get_ticklabels()
29: (8)                     labels = [l.get_text() for l in labels if l.get_text()]
30: (8)                     if not labels:
31: (12)                        ticks = []
32: (12)                        labels = []
33: (8)                     if not np.iterable(value):
34: (12)                        value = [value]
35: (8)                     newValues = []
36: (8)                     for v in value:
37: (12)                        if v not in labels and v not in newValues:
38: (16)                            newValues.append(v)
39: (8)                     labels.extend(newValues)
40: (8)                     labels = ['] + labels + ['']
41: (8)                     ticks = list(range(len(labels)))
42: (8)                     ticks[0] = 0.5
43: (8)                     ticks[-1] = ticks[-1] - 0.5
44: (8)                     axis.set_ticks(ticks)
45: (8)                     axis.set_ticklabels(labels)
46: (8)                     loc = axis.get_major_locator()
47: (8)                     loc.set_bounds(ticks[0], ticks[-1])
48: (8)                     if isXAxis:
49: (12)                        ax.set_xlim(ticks[0], ticks[-1])
50: (8)                     else:
51: (12)                        ax.set_ylim(ticks[0], ticks[-1])
52: (8)                     result = [ticks[labels.index(v)] for v in value]
53: (8)                     ax.viewLim.ignore(-1)
54: (8)                     return result
55: (4)                 @staticmethod
56: (4)                 def default_units(value, axis):
57: (8)                     return "indexed"
```

File 138 - UnitDblConverter.py:

```
1: (0)              """UnitDblConverter module containing class UnitDblConverter."""
2: (0)              import numpy as np
3: (0)              from matplotlib import cbook, units
4: (0)              import matplotlib.projections.polar as polar
5: (0)              __all__ = ['UnitDblConverter']
6: (0)              def rad_fn(x, pos=None):
7: (4)                  """Radian function formatter."""
8: (4)                  n = int((x / np.pi) * 2.0 + 0.25)
9: (4)                  if n == 0:
10: (8)                     return str(x)
```

```
11: (4)                 elif n == 1:
12: (8)                     return r'$\pi/2$'
13: (4)                 elif n == 2:
14: (8)                     return r'$\pi$'
15: (4)                 elif n % 2 == 0:
16: (8)                     return fr'${n//2}\pi$'
17: (4)                 else:
18: (8)                     return fr'${n}\pi/2$'
19: (0)             class UnitDblConverter(units.ConversionInterface):
20: (4)                 """
21: (4)                 Provides Matplotlib conversion functionality for the Monte UnitDbl class.
22: (4)                 """
23: (4)                 defaults = {
24: (7)                    "distance": 'km',
25: (7)                    "angle": 'deg',
26: (7)                    "time": 'sec',
27: (7)                    }
28: (4)                 @staticmethod
29: (4)                 def axisinfo(unit, axis):
30: (8)                     import matplotlib.testing.jpl_units as U
31: (8)                     if unit:
32: (12)                        label = unit if isinstance(unit, str) else unit.label()
33: (8)                     else:
34: (12)                        label = None
35: (8)                     if label == "deg" and isinstance(axis.axes, polar.PolarAxes):
36: (12)                        majfmt = polar.PolarAxes.ThetaFormatter()
37: (8)                     else:
38: (12)                        majfmt = U.UnitDblFormatter(useOffset=False)
39: (8)                     return units.AxisInfo(majfmt=majfmt, label=label)
40: (4)                 @staticmethod
41: (4)                 def convert(value, unit, axis):
42: (8)                     if not cbook.is_scalar_or_string(value):
43: (12)                        return [UnitDblConverter.convert(x, unit, axis) for x in value]
44: (8)                     if unit is None:
45: (12)                        unit = UnitDblConverter.default_units(value, axis)
46: (8)                     if isinstance(axis.axes, polar.PolarAxes) and value.type() == "angle":
47: (12)                        return value.convert("rad")
48: (8)                     return value.convert(unit)
49: (4)                 @staticmethod
50: (4)                 def default_units(value, axis):
51: (8)                     if cbook.is_scalar_or_string(value):
52: (12)                        return UnitDblConverter.defaults[value.type()]
53: (8)                     else:
54: (12)                        return UnitDblConverter.default_units(value[0], axis)

----------------------------------------

File 139 - UnitDblFormatter.py:

1: (0)              """UnitDblFormatter module containing class UnitDblFormatter."""
2: (0)              import matplotlib.ticker as ticker
3: (0)              __all__ = ['UnitDblFormatter']
4: (0)              class UnitDblFormatter(ticker.ScalarFormatter):
5: (4)                  """
6: (4)                  The formatter for UnitDbl data types.
7: (4)                  This allows for formatting with the unit string.
8: (4)                  """
9: (4)                  def __call__(self, x, pos=None):
10: (8)                     if len(self.locs) == 0:
11: (12)                        return ''
12: (8)                     else:
13: (12)                        return f'{x:.12}'
14: (4)                 def format_data_short(self, value):
15: (8)                     return f'{value:.12}'
16: (4)                 def format_data(self, value):
17: (8)                     return f'{value:.12}'

----------------------------------------
```

File 140 - UnitDbl.py:

```
1: (0)              """UnitDbl module."""
2: (0)              import functools
3: (0)              import operator
4: (0)              from matplotlib import _api
5: (0)              class UnitDbl:
6: (4)                  """Class UnitDbl in development."""
7: (4)                  allowed = {
8: (8)                      "m": (0.001, "km"),
9: (8)                      "km": (1, "km"),
10: (8)                     "mile": (1.609344, "km"),
11: (8)                     "rad": (1, "rad"),
12: (8)                     "deg": (1.745329251994330e-02, "rad"),
13: (8)                     "sec": (1, "sec"),
14: (8)                     "min": (60.0, "sec"),
15: (8)                     "hour": (3600, "sec"),
16: (8)                     }
17: (4)                 _types = {
18: (8)                     "km": "distance",
19: (8)                     "rad": "angle",
20: (8)                     "sec": "time",
21: (8)                     }
22: (4)                 def __init__(self, value, units):
23: (8)                     """
24: (8)                     Create a new UnitDbl object.
25: (8)                     Units are internally converted to km, rad, and sec.  The only
26: (8)                     valid inputs for units are [m, km, mile, rad, deg, sec, min, hour].
27: (8)                     The field UnitDbl.value will contain the converted value.  Use
28: (8)                     the convert() method to get a specific type of units back.
29: (8)                     = ERROR CONDITIONS
30: (8)                     - If the input units are not in the allowed list, an error is thrown.
31: (8)                     = INPUT VARIABLES
32: (8)                     - value     The numeric value of the UnitDbl.
33: (8)                     - units     The string name of the units the value is in.
34: (8)                     """
35: (8)                     data = _api.check_getitem(self.allowed, units=units)
36: (8)                     self._value = float(value * data[0])
37: (8)                     self._units = data[1]
38: (4)                 def convert(self, units):
39: (8)                     """
40: (8)                     Convert the UnitDbl to a specific set of units.
41: (8)                     = ERROR CONDITIONS
42: (8)                     - If the input units are not in the allowed list, an error is thrown.
43: (8)                     = INPUT VARIABLES
44: (8)                     - units     The string name of the units to convert to.
45: (8)                     = RETURN VALUE
46: (8)                     - Returns the value of the UnitDbl in the requested units as a
floating
47: (10)                      point number.
48: (8)                     """
49: (8)                     if self._units == units:
50: (12)                        return self._value
51: (8)                     data = _api.check_getitem(self.allowed, units=units)
52: (8)                     if self._units != data[1]:
53: (12)                        raise ValueError(f"Error trying to convert to different units.\n"
54: (29)                                         f"    Invalid conversion requested.\n"
55: (29)                                         f"    UnitDbl: {self}\n"
56: (29)                                         f"    Units:   {units}\n")
57: (8)                     return self._value / data[0]
58: (4)                 def __abs__(self):
59: (8)                     """Return the absolute value of this UnitDbl."""
60: (8)                     return UnitDbl(abs(self._value), self._units)
61: (4)                 def __neg__(self):
62: (8)                     """Return the negative value of this UnitDbl."""
63: (8)                     return UnitDbl(-self._value, self._units)
64: (4)                 def __bool__(self):
65: (8)                     """Return the truth value of a UnitDbl."""
66: (8)                     return bool(self._value)
```

```
(4)     def _cmp(self, op, rhs):
(8)         """Check that *self* and *rhs* share units; compare them using
*op*."""
(8)         self.checkSameUnits(rhs, "compare")
(8)         return op(self._value, rhs._value)
(4)     __eq__ = functools.partialmethod(_cmp, operator.eq)
(4)     __ne__ = functools.partialmethod(_cmp, operator.ne)
(4)     __lt__ = functools.partialmethod(_cmp, operator.lt)
(4)     __le__ = functools.partialmethod(_cmp, operator.le)
(4)     __gt__ = functools.partialmethod(_cmp, operator.gt)
(4)     __ge__ = functools.partialmethod(_cmp, operator.ge)
(4)     def _binop_unit_unit(self, op, rhs):
(8)         """Check that *self* and *rhs* share units; combine them using
*op*."""
(8)         self.checkSameUnits(rhs, op.__name__)
(8)         return UnitDbl(op(self._value, rhs._value), self._units)
(4)     __add__ = functools.partialmethod(_binop_unit_unit, operator.add)
(4)     __sub__ = functools.partialmethod(_binop_unit_unit, operator.sub)
(4)     def _binop_unit_scalar(self, op, scalar):
(8)         """Combine *self* and *scalar* using *op*."""
(8)         return UnitDbl(op(self._value, scalar), self._units)
(4)     __mul__ = functools.partialmethod(_binop_unit_scalar, operator.mul)
(4)     __rmul__ = functools.partialmethod(_binop_unit_scalar, operator.mul)
(4)     def __str__(self):
(8)         """Print the UnitDbl."""
(8)         return f"{self._value:g} *{self._units}"
(4)     def __repr__(self):
(8)         """Print the UnitDbl."""
(8)         return f"UnitDbl({self._value:g}, '{self._units}')"
(4)     def type(self):
(8)         """Return the type of UnitDbl data."""
(8)         return self._types[self._units]
(4)     @staticmethod
(4)     def range(start, stop, step=None):
(8)         """
(8)         Generate a range of UnitDbl objects.
(8)         Similar to the Python range() method.  Returns the range [
(8)         start, stop) at the requested step.  Each element will be a
(8)         UnitDbl object.
(8)         = INPUT VARIABLES
(8)         - start     The starting value of the range.
(8)         - stop      The stop value of the range.
(8)         - step      Optional step to use.  If set to None, then a UnitDbl of
(22)                      value 1 w/ the units of the start is used.
(8)         = RETURN VALUE
(8)         - Returns a list containing the requested UnitDbl values.
(8)         """
(8)         if step is None:
(12)            step = UnitDbl(1, start._units)
(8)         elems = []
(8)         i = 0
(8)         while True:
(12)            d = start + i * step
(12)            if d >= stop:
(16)                break
(12)            elems.append(d)
(12)            i += 1
(8)         return elems
(4)     def checkSameUnits(self, rhs, func):
(8)         """
(8)         Check to see if units are the same.
(8)         = ERROR CONDITIONS
(8)         - If the units of the rhs UnitDbl are not the same as our units,
(10)          an error is thrown.
(8)         = INPUT VARIABLES
(8)         - rhs     The UnitDbl to check for the same units
(8)         - func    The name of the function doing the check.
(8)         """
(8)         if self._units != rhs._units:
```

```
134: (12)                       raise ValueError(f"Cannot {func} units of different types.\n"
135: (29)                                        f"LHS: {self._units}\n"
136: (29)                                        f"RHS: {rhs._units}")
```

File 141 - __init__.py:

```
1: (0)              from pathlib import Path
2: (0)              if not (Path(__file__).parent / 'baseline_images').exists():
3: (4)                  raise OSError(
4: (8)                      'The baseline image directory does not exist. '
5: (8)                      'This is most likely because the test data is not installed. '
6: (8)                      'You may need to install matplotlib from source to get the '
7: (8)                      'test data.')
```

File 142 - conftest.py:

```
1: (0)              from matplotlib.testing.conftest import (  # noqa
2: (4)                  mpl_test_settings, pytest_configure, pytest_unconfigure, pd, xr)
```

File 143 - test_afm.py:

```
1: (0)              from io import BytesIO
2: (0)              import pytest
3: (0)              import logging
4: (0)              from matplotlib import _afm
5: (0)              from matplotlib import font_manager as fm
6: (0)              AFM_TEST_DATA = b"""StartFontMetrics 2.0
7: (0)              Comment Comments are ignored.
8: (0)              Comment Creation Date:Mon Nov 13 12:34:11 GMT 2017
9: (0)              FontName MyFont-Bold
10: (0)             EncodingScheme FontSpecific
11: (0)             FullName My Font Bold
12: (0)             FamilyName Test Fonts
13: (0)             Weight Bold
14: (0)             ItalicAngle 0.0
15: (0)             IsFixedPitch false
16: (0)             UnderlinePosition -100
17: (0)             UnderlineThickness 56,789
18: (0)             Version 001.000
19: (0)             Notice Copyright \xa9 2017 No one.
20: (0)             FontBBox 0 -321 1234 369
21: (0)             StartCharMetrics 3
22: (0)             C 0 ; WX 250 ; N space ; B 0 0 0 0 ;
23: (0)             C 42 ; WX 1141 ; N foo ; B 40 60 800 360 ;
24: (0)             C 99 ; WX 583 ; N bar ; B 40 -10 543 210 ;
25: (0)             EndCharMetrics
26: (0)             EndFontMetrics
27: (0)             """
28: (0)             def test_nonascii_str():
29: (4)                 inp_str = "привет"
30: (4)                 byte_str = inp_str.encode("utf8")
31: (4)                 ret = _afm._to_str(byte_str)
32: (4)                 assert ret == inp_str
33: (0)             def test_parse_header():
34: (4)                 fh = BytesIO(AFM_TEST_DATA)
35: (4)                 header = _afm._parse_header(fh)
36: (4)                 assert header == {
37: (8)                     b'StartFontMetrics': 2.0,
38: (8)                     b'FontName': 'MyFont-Bold',
39: (8)                     b'EncodingScheme': 'FontSpecific',
40: (8)                     b'FullName': 'My Font Bold',
41: (8)                     b'FamilyName': 'Test Fonts',
42: (8)                     b'Weight': 'Bold',
```

```
(8)                     b'ItalicAngle': 0.0,
(8)                     b'IsFixedPitch': False,
(8)                     b'UnderlinePosition': -100,
(8)                     b'UnderlineThickness': 56.789,
(8)                     b'Version': '001.000',
(8)                     b'Notice': b'Copyright \xa9 2017 No one.',
(8)                     b'FontBBox': [0, -321, 1234, 369],
(8)                     b'StartCharMetrics': 3,
(4)                 }
(0)             def test_parse_char_metrics():
(4)                 fh = BytesIO(AFM_TEST_DATA)
(4)                 _afm._parse_header(fh)  # position
(4)                 metrics = _afm._parse_char_metrics(fh)
(4)                 assert metrics == (
(8)                     {0: (250.0, 'space', [0, 0, 0, 0]),
(9)                      42: (1141.0, 'foo', [40, 60, 800, 360]),
(9)                      99: (583.0, 'bar', [40, -10, 543, 210]),
(9)                      },
(8)                     {'space': (250.0, 'space', [0, 0, 0, 0]),
(9)                      'foo': (1141.0, 'foo', [40, 60, 800, 360]),
(9)                      'bar': (583.0, 'bar', [40, -10, 543, 210]),
(9)                      })
(0)             def test_get_familyname_guessed():
(4)                 fh = BytesIO(AFM_TEST_DATA)
(4)                 font = _afm.AFM(fh)
(4)                 del font._header[b'FamilyName']  # remove FamilyName, so we have to guess
(4)                 assert font.get_familyname() == 'My Font'
(0)             def test_font_manager_weight_normalization():
(4)                 font = _afm.AFM(BytesIO(
(8)                     AFM_TEST_DATA.replace(b"Weight Bold\n", b"Weight Custom\n")))
(4)                 assert fm.afmFontProperty("", font).weight == "normal"
(0)             @pytest.mark.parametrize(
(4)                 "afm_data",
(4)                 [
(8)                     b"""nope
(0)             really nope""",
(8)                     b"""StartFontMetrics 2.0
(0)             Comment Comments are ignored.
(0)             Comment Creation Date:Mon Nov 13 12:34:11 GMT 2017
(0)             FontName MyFont-Bold
(0)             EncodingScheme FontSpecific""",
(4)                 ],
(0)             )
(0)             def test_bad_afm(afm_data):
(4)                 fh = BytesIO(afm_data)
(4)                 with pytest.raises(RuntimeError):
(8)                     _afm._parse_header(fh)
(0)             @pytest.mark.parametrize(
(4)                 "afm_data",
(4)                 [
(8)                     b"""StartFontMetrics 2.0
(0)             Comment Comments are ignored.
(0)             Comment Creation Date:Mon Nov 13 12:34:11 GMT 2017
(0)             Aardvark bob
(0)             FontName MyFont-Bold
(0)             EncodingScheme FontSpecific
(0)             StartCharMetrics 3""",
(8)                     b"""StartFontMetrics 2.0
(0)             Comment Comments are ignored.
(0)             Comment Creation Date:Mon Nov 13 12:34:11 GMT 2017
(0)             ItalicAngle zero degrees
(0)             FontName MyFont-Bold
(0)             EncodingScheme FontSpecific
(0)             StartCharMetrics 3""",
(4)                 ],
(0)             )
(0)             def test_malformed_header(afm_data, caplog):
(4)                 fh = BytesIO(afm_data)
(4)                 with caplog.at_level(logging.ERROR):
```

```
112: (8)                    _afm._parse_header(fh)
113: (4)                assert len(caplog.records) == 1

----------------------------------------

File 144 - test_agg_filter.py:

1: (0)              import numpy as np
2: (0)              import matplotlib.pyplot as plt
3: (0)              from matplotlib.testing.decorators import image_comparison
4: (0)              @image_comparison(baseline_images=['agg_filter_alpha'],
5: (18)                                  extensions=['png', 'pdf'])
6: (0)              def test_agg_filter_alpha():
7: (4)                  plt.rcParams['pcolormesh.snap'] = False
8: (4)                  ax = plt.axes()
9: (4)                  x, y = np.mgrid[0:7, 0:8]
10: (4)                 data = x**2 - y**2
11: (4)                 mesh = ax.pcolormesh(data, cmap='Reds', zorder=5)
12: (4)                 def manual_alpha(im, dpi):
13: (8)                     im[:, :, 3] *= 0.6
14: (8)                     print('CALLED')
15: (8)                     return im, 0, 0
16: (4)                 mesh.set_agg_filter(manual_alpha)
17: (4)                 mesh.set_rasterized(True)
18: (4)                 ax.plot([0, 4, 7], [1, 3, 8])

----------------------------------------

File 145 - test_agg.py:

1: (0)              import io
2: (0)              import numpy as np
3: (0)              from numpy.testing import assert_array_almost_equal
4: (0)              from PIL import Image, TiffTags
5: (0)              import pytest
6: (0)              from matplotlib import (
7: (4)                  collections, patheffects, pyplot as plt, transforms as mtransforms,
8: (4)                  rcParams, rc_context)
9: (0)              from matplotlib.backends.backend_agg import RendererAgg
10: (0)             from matplotlib.figure import Figure
11: (0)             from matplotlib.image import imread
12: (0)             from matplotlib.path import Path
13: (0)             from matplotlib.testing.decorators import image_comparison
14: (0)             from matplotlib.transforms import IdentityTransform
15: (0)             def test_repeated_save_with_alpha():
16: (4)                 fig = Figure([1, 0.4])
17: (4)                 fig.set_facecolor((0, 1, 0.4))
18: (4)                 fig.patch.set_alpha(0.25)
19: (4)                 buf = io.BytesIO()
20: (4)                 fig.savefig(buf,
21: (16)                            facecolor=fig.get_facecolor(),
22: (16)                            edgecolor='none')
23: (4)                 buf.seek(0)
24: (4)                 fig.savefig(buf,
25: (16)                            facecolor=fig.get_facecolor(),
26: (16)                            edgecolor='none')
27: (4)                 buf.seek(0)
28: (4)                 assert_array_almost_equal(tuple(imread(buf)[0, 0]),
29: (30)                                          (0.0, 1.0, 0.4, 0.250),
30: (30)                                          decimal=3)
31: (0)             def test_large_single_path_collection():
32: (4)                 buff = io.BytesIO()
33: (4)                 f, ax = plt.subplots()
34: (4)                 collection = collections.PathCollection(
35: (8)                     [Path([[-10, 5], [10, 5], [10, -5], [-10, -5], [-10, 5]])])
36: (4)                 ax.add_artist(collection)
37: (4)                 ax.set_xlim(10**-3, 1)
38: (4)                 plt.savefig(buff)
39: (0)             def test_marker_with_nan():
```

```
 40: (4)                 fig, ax = plt.subplots(1)
 41: (4)                 steps = 1000
 42: (4)                 data = np.arange(steps)
 43: (4)                 ax.semilogx(data)
 44: (4)                 ax.fill_between(data, data*0.8, data*1.2)
 45: (4)                 buf = io.BytesIO()
 46: (4)                 fig.savefig(buf, format='png')
 47: (0)             def test_long_path():
 48: (4)                 buff = io.BytesIO()
 49: (4)                 fig = Figure()
 50: (4)                 ax = fig.subplots()
 51: (4)                 points = np.ones(100_000)
 52: (4)                 points[::2] *= -1
 53: (4)                 ax.plot(points)
 54: (4)                 fig.savefig(buff, format='png')
 55: (0)             @image_comparison(['agg_filter.png'], remove_text=True)
 56: (0)             def test_agg_filter():
 57: (4)                 def smooth1d(x, window_len):
 58: (8)                     s = np.r_[
 59: (12)                        2*x[0] - x[window_len:1:-1], x, 2*x[-1] - x[-1:-window_len:-1]]
 60: (8)                     w = np.hanning(window_len)
 61: (8)                     y = np.convolve(w/w.sum(), s, mode='same')
 62: (8)                     return y[window_len-1:-window_len+1]
 63: (4)                 def smooth2d(A, sigma=3):
 64: (8)                     window_len = max(int(sigma), 3) * 2 + 1
 65: (8)                     A = np.apply_along_axis(smooth1d, 0, A, window_len)
 66: (8)                     A = np.apply_along_axis(smooth1d, 1, A, window_len)
 67: (8)                     return A
 68: (4)                 class BaseFilter:
 69: (8)                     def get_pad(self, dpi):
 70: (12)                        return 0
 71: (8)                     def process_image(self, padded_src, dpi):
 72: (12)                        raise NotImplementedError("Should be overridden by subclasses")
 73: (8)                     def __call__(self, im, dpi):
 74: (12)                        pad = self.get_pad(dpi)
 75: (12)                        padded_src = np.pad(im, [(pad, pad), (pad, pad), (0, 0)],
 76: (32)                                            "constant")
 77: (12)                        tgt_image = self.process_image(padded_src, dpi)
 78: (12)                        return tgt_image, -pad, -pad
 79: (4)                 class OffsetFilter(BaseFilter):
 80: (8)                     def __init__(self, offsets=(0, 0)):
 81: (12)                        self.offsets = offsets
 82: (8)                     def get_pad(self, dpi):
 83: (12)                        return int(max(self.offsets) / 72 * dpi)
 84: (8)                     def process_image(self, padded_src, dpi):
 85: (12)                        ox, oy = self.offsets
 86: (12)                        a1 = np.roll(padded_src, int(ox / 72 * dpi), axis=1)
 87: (12)                        a2 = np.roll(a1, -int(oy / 72 * dpi), axis=0)
 88: (12)                        return a2
 89: (4)                 class GaussianFilter(BaseFilter):
 90: (8)                     """Simple Gaussian filter."""
 91: (8)                     def __init__(self, sigma, alpha=0.5, color=(0, 0, 0)):
 92: (12)                        self.sigma = sigma
 93: (12)                        self.alpha = alpha
 94: (12)                        self.color = color
 95: (8)                     def get_pad(self, dpi):
 96: (12)                        return int(self.sigma*3 / 72 * dpi)
 97: (8)                     def process_image(self, padded_src, dpi):
 98: (12)                        tgt_image = np.empty_like(padded_src)
 99: (12)                        tgt_image[:, :, :3] = self.color
100: (12)                        tgt_image[:, :, 3] = smooth2d(padded_src[:, :, 3] * self.alpha,
101: (42)                                              self.sigma / 72 * dpi)
102: (12)                        return tgt_image
103: (4)                 class DropShadowFilter(BaseFilter):
104: (8)                     def __init__(self, sigma, alpha=0.3, color=(0, 0, 0), offsets=(0, 0)):
105: (12)                        self.gauss_filter = GaussianFilter(sigma, alpha, color)
106: (12)                        self.offset_filter = OffsetFilter(offsets)
107: (8)                     def get_pad(self, dpi):
108: (12)                        return max(self.gauss_filter.get_pad(dpi),
```

```
109: (23)                                self.offset_filter.get_pad(dpi))
110: (8)                    def process_image(self, padded_src, dpi):
111: (12)                       t1 = self.gauss_filter.process_image(padded_src, dpi)
112: (12)                       t2 = self.offset_filter.process_image(t1, dpi)
113: (12)                       return t2
114: (4)                fig, ax = plt.subplots()
115: (4)                line1, = ax.plot([0.1, 0.5, 0.9], [0.1, 0.9, 0.5], "bo-",
116: (21)                                 mec="b", mfc="w", lw=5, mew=3, ms=10, label="Line 1")
117: (4)                line2, = ax.plot([0.1, 0.5, 0.9], [0.5, 0.2, 0.7], "ro-",
118: (21)                                 mec="r", mfc="w", lw=5, mew=3, ms=10, label="Line 1")
119: (4)                gauss = DropShadowFilter(4)
120: (4)                for line in [line1, line2]:
121: (8)                    xx = line.get_xdata()
122: (8)                    yy = line.get_ydata()
123: (8)                    shadow, = ax.plot(xx, yy)
124: (8)                    shadow.update_from(line)
125: (8)                    transform = mtransforms.offset_copy(line.get_transform(), ax.figure,
126: (44)                                                       x=4.0, y=-6.0, units='points')
127: (8)                    shadow.set_transform(transform)
128: (8)                    shadow.set_zorder(line.get_zorder() - 0.5)
129: (8)                    shadow.set_agg_filter(gauss)
130: (8)                    shadow.set_rasterized(True)  # to support mixed-mode renderers
131: (4)                ax.set_xlim(0., 1.)
132: (4)                ax.set_ylim(0., 1.)
133: (4)                ax.xaxis.set_visible(False)
134: (4)                ax.yaxis.set_visible(False)
135: (0)            def test_too_large_image():
136: (4)                fig = plt.figure(figsize=(300, 1000))
137: (4)                buff = io.BytesIO()
138: (4)                with pytest.raises(ValueError):
139: (8)                    fig.savefig(buff)
140: (0)            def test_chunksize():
141: (4)                x = range(200)
142: (4)                fig, ax = plt.subplots()
143: (4)                ax.plot(x, np.sin(x))
144: (4)                fig.canvas.draw()
145: (4)                fig, ax = plt.subplots()
146: (4)                rcParams['agg.path.chunksize'] = 105
147: (4)                ax.plot(x, np.sin(x))
148: (4)                fig.canvas.draw()
149: (0)            @pytest.mark.backend('Agg')
150: (0)            def test_jpeg_dpi():
151: (4)                plt.plot([0, 1, 2], [0, 1, 0])
152: (4)                buf = io.BytesIO()
153: (4)                plt.savefig(buf, format="jpg", dpi=200)
154: (4)                im = Image.open(buf)
155: (4)                assert im.info['dpi'] == (200, 200)
156: (0)            def test_pil_kwargs_png():
157: (4)                from PIL.PngImagePlugin import PngInfo
158: (4)                buf = io.BytesIO()
159: (4)                pnginfo = PngInfo()
160: (4)                pnginfo.add_text("Software", "test")
161: (4)                plt.figure().savefig(buf, format="png", pil_kwargs={"pnginfo": pnginfo})
162: (4)                im = Image.open(buf)
163: (4)                assert im.info["Software"] == "test"
164: (0)            def test_pil_kwargs_tiff():
165: (4)                buf = io.BytesIO()
166: (4)                pil_kwargs = {"description": "test image"}
167: (4)                plt.figure().savefig(buf, format="tiff", pil_kwargs=pil_kwargs)
168: (4)                im = Image.open(buf)
169: (4)                tags = {TiffTags.TAGS_V2[k].name: v for k, v in im.tag_v2.items()}
170: (4)                assert tags["ImageDescription"] == "test image"
171: (0)            def test_pil_kwargs_webp():
172: (4)                plt.plot([0, 1, 2], [0, 1, 0])
173: (4)                buf_small = io.BytesIO()
174: (4)                pil_kwargs_low = {"quality": 1}
175: (4)                plt.savefig(buf_small, format="webp", pil_kwargs=pil_kwargs_low)
176: (4)                assert len(pil_kwargs_low) == 1
177: (4)                buf_large = io.BytesIO()
```

```
178: (4)                pil_kwargs_high = {"quality": 100}
179: (4)                plt.savefig(buf_large, format="webp", pil_kwargs=pil_kwargs_high)
180: (4)                assert len(pil_kwargs_high) == 1
181: (4)                assert buf_large.getbuffer().nbytes > buf_small.getbuffer().nbytes
182: (0)            def test_webp_alpha():
183: (4)                plt.plot([0, 1, 2], [0, 1, 0])
184: (4)                buf = io.BytesIO()
185: (4)                plt.savefig(buf, format="webp", transparent=True)
186: (4)                im = Image.open(buf)
187: (4)                assert im.mode == "RGBA"
188: (0)            def test_draw_path_collection_error_handling():
189: (4)                fig, ax = plt.subplots()
190: (4)                ax.scatter([1], [1]).set_paths(Path([(0, 1), (2, 3)]))
191: (4)                with pytest.raises(TypeError):
192: (8)                    fig.canvas.draw()
193: (0)            def test_chunksize_fails():
194: (4)                N = 100_000
195: (4)                dpi = 500
196: (4)                w = 5*dpi
197: (4)                h = 6*dpi
198: (4)                x = np.linspace(0, w, N)
199: (4)                y = np.ones(N) * h
200: (4)                y[::2] = 0
201: (4)                path = Path(np.vstack((x, y)).T)
202: (4)                path.simplify_threshold = 0
203: (4)                ra = RendererAgg(w, h, dpi)
204: (4)                gc = ra.new_gc()
205: (4)                gc.set_linewidth(1)
206: (4)                gc.set_foreground('r')
207: (4)                gc.set_hatch('/')
208: (4)                with pytest.raises(OverflowError, match='cannot split hatched path'):
209: (8)                    ra.draw_path(gc, path, IdentityTransform())
210: (4)                gc.set_hatch(None)
211: (4)                with pytest.raises(OverflowError, match='cannot split filled path'):
212: (8)                    ra.draw_path(gc, path, IdentityTransform(), (1, 0, 0))
213: (4)                with rc_context({'agg.path.chunksize': 0}):
214: (8)                    with pytest.raises(OverflowError, match='Please set'):
215: (12)                       ra.draw_path(gc, path, IdentityTransform())
216: (4)                with rc_context({'agg.path.chunksize': 1_000_000}):
217: (8)                    with pytest.raises(OverflowError, match='Please reduce'):
218: (12)                       ra.draw_path(gc, path, IdentityTransform())
219: (4)                with rc_context({'agg.path.chunksize': 90_000}):
220: (8)                    with pytest.raises(OverflowError, match='Please reduce'):
221: (12)                       ra.draw_path(gc, path, IdentityTransform())
222: (4)                path.should_simplify = False
223: (4)                with pytest.raises(OverflowError, match="should_simplify is False"):
224: (8)                    ra.draw_path(gc, path, IdentityTransform())
225: (0)            def test_non_tuple_rgbaface():
226: (4)                fig = plt.figure()
227: (4)                fig.add_subplot(projection="3d").scatter(
228: (8)                    [0, 1, 2], [0, 1, 2], path_effects=[patheffects.Stroke(linewidth=4)])
229: (4)                fig.canvas.draw()
```

File 146 - test_animation.py:

```
1: (0)              import os
2: (0)              from pathlib import Path
3: (0)              import platform
4: (0)              import re
5: (0)              import shutil
6: (0)              import subprocess
7: (0)              import sys
8: (0)              import weakref
9: (0)              import numpy as np
10: (0)             import pytest
11: (0)             import matplotlib as mpl
12: (0)             from matplotlib import pyplot as plt
```

```
(0)             from matplotlib import animation
(0)             from matplotlib.testing.decorators import check_figures_equal
(0)             @pytest.fixture()
(0)             def anim(request):
(4)                 """Create a simple animation (with options)."""
(4)                 fig, ax = plt.subplots()
(4)                 line, = ax.plot([], [])
(4)                 ax.set_xlim(0, 10)
(4)                 ax.set_ylim(-1, 1)
(4)                 def init():
(8)                     line.set_data([], [])
(8)                     return line,
(4)                 def animate(i):
(8)                     x = np.linspace(0, 10, 100)
(8)                     y = np.sin(x + i)
(8)                     line.set_data(x, y)
(8)                     return line,
(4)                 kwargs = dict(getattr(request, 'param', {}))  # make a copy
(4)                 klass = kwargs.pop('klass', animation.FuncAnimation)
(4)                 if 'frames' not in kwargs:
(8)                     kwargs['frames'] = 5
(4)                 return klass(fig=fig, func=animate, init_func=init, **kwargs)
(0)             class NullMovieWriter(animation.AbstractMovieWriter):
(4)                 """
(4)                 A minimal MovieWriter.  It doesn't actually write anything.
(4)                 It just saves the arguments that were given to the setup() and
(4)                 grab_frame() methods as attributes, and counts how many times
(4)                 grab_frame() is called.
(4)                 This class doesn't have an __init__ method with the appropriate
(4)                 signature, and it doesn't define an isAvailable() method, so
(4)                 it cannot be added to the 'writers' registry.
(4)                 """
(4)                 def setup(self, fig, outfile, dpi, *args):
(8)                     self.fig = fig
(8)                     self.outfile = outfile
(8)                     self.dpi = dpi
(8)                     self.args = args
(8)                     self._count = 0
(4)                 def grab_frame(self, **savefig_kwargs):
(8)                     from matplotlib.animation import _validate_grabframe_kwargs
(8)                     _validate_grabframe_kwargs(savefig_kwargs)
(8)                     self.savefig_kwargs = savefig_kwargs
(8)                     self._count += 1
(4)                 def finish(self):
(8)                     pass
(0)             def test_null_movie_writer(anim):
(4)                 plt.rcParams["savefig.facecolor"] = "auto"
(4)                 filename = "unused.null"
(4)                 dpi = 50
(4)                 savefig_kwargs = dict(foo=0)
(4)                 writer = NullMovieWriter()
(4)                 anim.save(filename, dpi=dpi, writer=writer,
(14)                          savefig_kwargs=savefig_kwargs)
(4)                 assert writer.fig == plt.figure(1)  # The figure used by anim fixture
(4)                 assert writer.outfile == filename
(4)                 assert writer.dpi == dpi
(4)                 assert writer.args == ()
(4)                 for k, v in savefig_kwargs.items():
(8)                     assert writer.savefig_kwargs[k] == v
(4)                 assert writer._count == anim._save_count
(0)             @pytest.mark.parametrize('anim', [dict(klass=dict)], indirect=['anim'])
(0)             def test_animation_delete(anim):
(4)                 if platform.python_implementation() == 'PyPy':
(8)                     np.testing.break_cycles()
(4)                 anim = animation.FuncAnimation(**anim)
(4)                 with pytest.warns(Warning, match='Animation was deleted'):
(8)                     del anim
(8)                     np.testing.break_cycles()
(0)             def test_movie_writer_dpi_default():
```

```
 82: (4)                class DummyMovieWriter(animation.MovieWriter):
 83: (8)                    def _run(self):
 84: (12)                       pass
 85: (4)                fig = plt.figure()
 86: (4)                filename = "unused.null"
 87: (4)                fps = 5
 88: (4)                codec = "unused"
 89: (4)                bitrate = 1
 90: (4)                extra_args = ["unused"]
 91: (4)                writer = DummyMovieWriter(fps, codec, bitrate, extra_args)
 92: (4)                writer.setup(fig, filename)
 93: (4)                assert writer.dpi == fig.dpi
 94: (0)            @animation.writers.register('null')
 95: (0)            class RegisteredNullMovieWriter(NullMovieWriter):
 96: (4)                def __init__(self, fps=None, codec=None, bitrate=None,
 97: (17)                            extra_args=None, metadata=None):
 98: (8)                    pass
 99: (4)                @classmethod
 100: (4)               def isAvailable(cls):
 101: (8)                   return True
 102: (0)           WRITER_OUTPUT = [
 103: (4)               ('ffmpeg', 'movie.mp4'),
 104: (4)               ('ffmpeg_file', 'movie.mp4'),
 105: (4)               ('imagemagick', 'movie.gif'),
 106: (4)               ('imagemagick_file', 'movie.gif'),
 107: (4)               ('pillow', 'movie.gif'),
 108: (4)               ('html', 'movie.html'),
 109: (4)               ('null', 'movie.null')
 110: (0)           ]
 111: (0)           def gen_writers():
 112: (4)               for writer, output in WRITER_OUTPUT:
 113: (8)                   if not animation.writers.is_available(writer):
 114: (12)                      mark = pytest.mark.skip(
 115: (16)                          f"writer '{writer}' not available on this system")
 116: (12)                      yield pytest.param(writer, None, output, marks=[mark])
 117: (12)                      yield pytest.param(writer, None, Path(output), marks=[mark])
 118: (12)                      continue
 119: (8)                   writer_class = animation.writers[writer]
 120: (8)                   for frame_format in getattr(writer_class, 'supported_formats',
 [None]):
 121: (12)                      yield writer, frame_format, output
 122: (12)                      yield writer, frame_format, Path(output)
 123: (0)           @pytest.mark.parametrize('writer, frame_format, output', gen_writers())
 124: (0)           @pytest.mark.parametrize('anim', [dict(klass=dict)], indirect=['anim'])
 125: (0)           def test_save_animation_smoketest(tmpdir, writer, frame_format, output, anim):
 126: (4)               if frame_format is not None:
 127: (8)                   plt.rcParams["animation.frame_format"] = frame_format
 128: (4)               anim = animation.FuncAnimation(**anim)
 129: (4)               dpi = None
 130: (4)               codec = None
 131: (4)               if writer == 'ffmpeg':
 132: (8)                   anim._fig.set_size_inches((10.85, 9.21))
 133: (8)                   dpi = 100.
 134: (8)                   codec = 'h264'
 135: (4)               with tmpdir.as_cwd():
 136: (8)                   anim.save(output, fps=30, writer=writer, bitrate=500, dpi=dpi,
 137: (18)                            codec=codec)
 138: (4)               del anim
 139: (0)           @pytest.mark.parametrize('writer, frame_format, output', gen_writers())
 140: (0)           def test_grabframe(tmpdir, writer, frame_format, output):
 141: (4)               WriterClass = animation.writers[writer]
 142: (4)               if frame_format is not None:
 143: (8)                   plt.rcParams["animation.frame_format"] = frame_format
 144: (4)               fig, ax = plt.subplots()
 145: (4)               dpi = None
 146: (4)               codec = None
 147: (4)               if writer == 'ffmpeg':
 148: (8)                   fig.set_size_inches((10.85, 9.21))
 149: (8)                   dpi = 100.
```

```
150: (8)                    codec = 'h264'
151: (4)                test_writer = WriterClass()
152: (4)                with tmpdir.as_cwd():
153: (8)                    with test_writer.saving(fig, output, dpi):
154: (12)                       test_writer.grab_frame()
155: (12)                       for k in {'dpi', 'bbox_inches', 'format'}:
156: (16)                           with pytest.raises(
157: (24)                                   TypeError,
158: (24)                                   match=f"grab_frame got an unexpected keyword argument
{k!r}"
159: (16)                           ):
160: (20)                               test_writer.grab_frame(**{k: object()})
161: (0)            @pytest.mark.parametrize('writer', [
162: (4)                pytest.param(
163: (8)                    'ffmpeg', marks=pytest.mark.skipif(
164: (12)                       not animation.FFMpegWriter.isAvailable(),
165: (12)                       reason='Requires FFMpeg')),
166: (4)                pytest.param(
167: (8)                    'imagemagick', marks=pytest.mark.skipif(
168: (12)                       not animation.ImageMagickWriter.isAvailable(),
169: (12)                       reason='Requires ImageMagick')),
170: (0)            ])
171: (0)            @pytest.mark.parametrize('html, want', [
172: (4)                ('none', None),
173: (4)                ('html5', '<video width'),
174: (4)                ('jshtml', '<script ')
175: (0)            ])
176: (0)            @pytest.mark.parametrize('anim', [dict(klass=dict)], indirect=['anim'])
177: (0)            def test_animation_repr_html(writer, html, want, anim):
178: (4)                if platform.python_implementation() == 'PyPy':
179: (8)                    np.testing.break_cycles()
180: (4)                if (writer == 'imagemagick' and html == 'html5'
181: (12)                       and not animation.FFMpegWriter.isAvailable()):
182: (8)                    pytest.skip('Requires FFMpeg')
183: (4)                anim = animation.FuncAnimation(**anim)
184: (4)                with plt.rc_context({'animation.writer': writer,
185: (25)                                    'animation.html': html}):
186: (8)                    html = anim._repr_html_()
187: (4)                if want is None:
188: (8)                    assert html is None
189: (8)                    with pytest.warns(UserWarning):
190: (12)                       del anim  # Animation was never run, so will warn on cleanup.
191: (12)                       np.testing.break_cycles()
192: (4)                else:
193: (8)                    assert want in html
194: (0)            @pytest.mark.parametrize(
195: (4)                'anim',
196: (4)                [{'save_count': 10, 'frames': iter(range(5))}],
197: (4)                indirect=['anim']
198: (0)            )
199: (0)            def test_no_length_frames(anim):
200: (4)                anim.save('unused.null', writer=NullMovieWriter())
201: (0)            def test_movie_writer_registry():
202: (4)                assert len(animation.writers._registered) > 0
203: (4)                mpl.rcParams['animation.ffmpeg_path'] = "not_available_ever_xxxx"
204: (4)                assert not animation.writers.is_available("ffmpeg")
205: (4)                bin = "true" if sys.platform != 'win32' else "where"
206: (4)                mpl.rcParams['animation.ffmpeg_path'] = bin
207: (4)                assert animation.writers.is_available("ffmpeg")
208: (0)            @pytest.mark.parametrize(
209: (4)                "method_name",
210: (4)                [pytest.param("to_html5_video", marks=pytest.mark.skipif(
211: (8)                    not animation.writers.is_available(mpl.rcParams["animation.writer"]),
212: (8)                    reason="animation writer not installed")),
213: (5)                 "to_jshtml"])
214: (0)            @pytest.mark.parametrize('anim', [dict(frames=1)], indirect=['anim'])
215: (0)            def test_embed_limit(method_name, caplog, tmpdir, anim):
216: (4)                caplog.set_level("WARNING")
217: (4)                with tmpdir.as_cwd():
```

```
218: (8)                    with mpl.rc_context({"animation.embed_limit": 1e-6}):  # ~1 byte.
219: (12)                       getattr(anim, method_name)()
220: (4)                assert len(caplog.records) == 1
221: (4)                record, = caplog.records
222: (4)                assert (record.name == "matplotlib.animation"
223: (12)                       and record.levelname == "WARNING")
224: (0)            @pytest.mark.parametrize(
225: (4)                "method_name",
226: (4)                [pytest.param("to_html5_video", marks=pytest.mark.skipif(
227: (8)                    not animation.writers.is_available(mpl.rcParams["animation.writer"]),
228: (8)                    reason="animation writer not installed")),
229: (5)                 "to_jshtml"])
230: (0)            @pytest.mark.parametrize('anim', [dict(frames=1)], indirect=['anim'])
231: (0)            def test_cleanup_temporaries(method_name, tmpdir, anim):
232: (4)                with tmpdir.as_cwd():
233: (8)                    getattr(anim, method_name)()
234: (8)                    assert list(Path(str(tmpdir)).iterdir()) == []
235: (0)            @pytest.mark.skipif(shutil.which("/bin/sh") is None, reason="requires a POSIX
OS")
236: (0)            def test_failing_ffmpeg(tmpdir, monkeypatch, anim):
237: (4)                """
238: (4)                Test that we correctly raise a CalledProcessError when ffmpeg fails.
239: (4)                To do so, mock ffmpeg using a simple executable shell script that
240: (4)                succeeds when called with no arguments (so that it gets registered by
241: (4)                `isAvailable`), but fails otherwise, and add it to the $PATH.
242: (4)                """
243: (4)                with tmpdir.as_cwd():
244: (8)                    monkeypatch.setenv("PATH", ".:" + os.environ["PATH"])
245: (8)                    exe_path = Path(str(tmpdir), "ffmpeg")
246: (8)                    exe_path.write_bytes(b"#!/bin/sh\n[[ $@ -eq 0 ]]\n")
247: (8)                    os.chmod(exe_path, 0o755)
248: (8)                    with pytest.raises(subprocess.CalledProcessError):
249: (12)                       anim.save("test.mpeg")
250: (0)            @pytest.mark.parametrize("cache_frame_data", [False, True])
251: (0)            def test_funcanimation_cache_frame_data(cache_frame_data):
252: (4)                fig, ax = plt.subplots()
253: (4)                line, = ax.plot([], [])
254: (4)                class Frame(dict):
255: (8)                    pass
256: (4)                def init():
257: (8)                    line.set_data([], [])
258: (8)                    return line,
259: (4)                def animate(frame):
260: (8)                    line.set_data(frame['x'], frame['y'])
261: (8)                    return line,
262: (4)                frames_generated = []
263: (4)                def frames_generator():
264: (8)                    for _ in range(5):
265: (12)                       x = np.linspace(0, 10, 100)
266: (12)                       y = np.random.rand(100)
267: (12)                       frame = Frame(x=x, y=y)
268: (12)                       frames_generated.append(weakref.ref(frame))
269: (12)                       yield frame
270: (4)                MAX_FRAMES = 100
271: (4)                anim = animation.FuncAnimation(fig, animate, init_func=init,
272: (35)                                              frames=frames_generator,
273: (35)                                              cache_frame_data=cache_frame_data,
274: (35)                                              save_count=MAX_FRAMES)
275: (4)                writer = NullMovieWriter()
276: (4)                anim.save('unused.null', writer=writer)
277: (4)                assert len(frames_generated) == 5
278: (4)                np.testing.break_cycles()
279: (4)                for f in frames_generated:
280: (8)                    assert (f() is None) != cache_frame_data
281: (0)            @pytest.mark.parametrize('return_value', [
282: (4)                None,
283: (4)                'string',
284: (4)                1,
285: (4)                ('string', ),
```

```
(4)                 'artist',
(0)             ])
(0)             def test_draw_frame(return_value):
(4)                 fig, ax = plt.subplots()
(4)                 line, = ax.plot([])
(4)                 def animate(i):
(8)                     line.set_data([0, 1], [0, i])
(8)                     if return_value == 'artist':
(12)                        return line
(8)                     else:
(12)                        return return_value
(4)                 with pytest.raises(RuntimeError):
(8)                     animation.FuncAnimation(
(12)                        fig, animate, blit=True, cache_frame_data=False
(8)                     )
(0)             def test_exhausted_animation(tmpdir):
(4)                 fig, ax = plt.subplots()
(4)                 def update(frame):
(8)                     return []
(4)                 anim = animation.FuncAnimation(
(8)                     fig, update, frames=iter(range(10)), repeat=False,
(8)                     cache_frame_data=False
(4)                 )
(4)                 with tmpdir.as_cwd():
(8)                     anim.save("test.gif", writer='pillow')
(4)                 with pytest.warns(UserWarning, match="exhausted"):
(8)                     anim._start()
(0)             def test_no_frame_warning(tmpdir):
(4)                 fig, ax = plt.subplots()
(4)                 def update(frame):
(8)                     return []
(4)                 anim = animation.FuncAnimation(
(8)                     fig, update, frames=[], repeat=False,
(8)                     cache_frame_data=False
(4)                 )
(4)                 with pytest.warns(UserWarning, match="exhausted"):
(8)                     anim._start()
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_animation_frame(tmpdir, fig_test, fig_ref):
(4)                 ax = fig_test.add_subplot()
(4)                 ax.set_xlim(0, 2 * np.pi)
(4)                 ax.set_ylim(-1, 1)
(4)                 x = np.linspace(0, 2 * np.pi, 100)
(4)                 line, = ax.plot([], [])
(4)                 def init():
(8)                     line.set_data([], [])
(8)                     return line,
(4)                 def animate(i):
(8)                     line.set_data(x, np.sin(x + i / 100))
(8)                     return line,
(4)                 anim = animation.FuncAnimation(
(8)                     fig_test, animate, init_func=init, frames=5,
(8)                     blit=True, repeat=False)
(4)                 with tmpdir.as_cwd():
(8)                     anim.save("test.gif")
(4)                 ax = fig_ref.add_subplot()
(4)                 ax.set_xlim(0, 2 * np.pi)
(4)                 ax.set_ylim(-1, 1)
(4)                 ax.plot(x, np.sin(x + 4 / 100))
(0)             @pytest.mark.parametrize('anim', [dict(klass=dict)], indirect=['anim'])
(0)             def test_save_count_override_warnings_has_length(anim):
(4)                 save_count = 5
(4)                 frames = list(range(2))
(4)                 match_target = (
(8)                     f'You passed in an explicit {save_count=} '
(8)                     "which is being ignored in favor of "
(8)                     f"{len(frames)=}."
(4)                 )
(4)                 with pytest.warns(UserWarning, match=re.escape(match_target)):
```

```
355: (8)                    anim = animation.FuncAnimation(
356: (12)                       **{**anim, 'frames': frames, 'save_count': save_count}
357: (8)                    )
358: (4)                assert anim._save_count == len(frames)
359: (4)                anim._init_draw()
360: (0)            @pytest.mark.parametrize('anim', [dict(klass=dict)], indirect=['anim'])
361: (0)            def test_save_count_override_warnings_scaler(anim):
362: (4)                save_count = 5
363: (4)                frames = 7
364: (4)                match_target = (
365: (8)                    f'You passed in an explicit {save_count=} ' +
366: (8)                    "which is being ignored in favor of " +
367: (8)                    f"{frames=}."
368: (4)                )
369: (4)                with pytest.warns(UserWarning, match=re.escape(match_target)):
370: (8)                    anim = animation.FuncAnimation(
371: (12)                       **{**anim, 'frames': frames, 'save_count': save_count}
372: (8)                    )
373: (4)                assert anim._save_count == frames
374: (4)                anim._init_draw()
375: (0)            @pytest.mark.parametrize('anim', [dict(klass=dict)], indirect=['anim'])
376: (0)            def test_disable_cache_warning(anim):
377: (4)                cache_frame_data = True
378: (4)                frames = iter(range(5))
379: (4)                match_target = (
380: (8)                    f"{frames=!r} which we can infer the length of, "
381: (8)                    "did not pass an explicit *save_count* "
382: (8)                    f"and passed {cache_frame_data=}.  To avoid a possibly "
383: (8)                    "unbounded cache, frame data caching has been disabled. "
384: (8)                    "To suppress this warning either pass "
385: (8)                    "`cache_frame_data=False` or `save_count=MAX_FRAMES`."
386: (4)                )
387: (4)                with pytest.warns(UserWarning, match=re.escape(match_target)):
388: (8)                    anim = animation.FuncAnimation(
389: (12)                       **{**anim, 'cache_frame_data': cache_frame_data, 'frames': frames}
390: (8)                    )
391: (4)                assert anim._cache_frame_data is False
392: (4)                anim._init_draw()
393: (0)            def test_movie_writer_invalid_path(anim):
394: (4)                if sys.platform == "win32":
395: (8)                    match_str = r"\[WinError 3] .*'\\\\foo\\\\bar\\\\aardvark'"
396: (4)                else:
397: (8)                    match_str = r"\[Errno 2] .*'/foo"
398: (4)                with pytest.raises(FileNotFoundError, match=match_str):
399: (8)                    anim.save("/foo/bar/aardvark/thiscannotreallyexist.mp4",
400: (18)                             writer=animation.FFMpegFileWriter())
```

File 147 - test_api.py:

```
1: (0)              from __future__ import annotations
2: (0)              import re
3: (0)              import typing
4: (0)              from typing import Any, Callable, TypeVar
5: (0)              import numpy as np
6: (0)              import pytest
7: (0)              import matplotlib as mpl
8: (0)              from matplotlib import _api
9: (0)              if typing.TYPE_CHECKING:
10: (4)                 from typing_extensions import Self
11: (0)             T = TypeVar('T')
12: (0)             @pytest.mark.parametrize('target,shape_repr,test_shape',
13: (25)                                     [((None, ), "(N,)", (1, 3)),
14: (26)                                      ((None, 3), "(N, 3)", (1,)),
15: (26)                                      ((None, 3), "(N, 3)", (1, 2)),
16: (26)                                      ((1, 5), "(1, 5)", (1, 9)),
17: (26)                                      ((None, 2, None), "(M, 2, N)", (1, 3, 1))
18: (26)                                      ])
```

```
(0)             def test_check_shape(target: tuple[int | None, ...],
(21)                                 shape_repr: str,
(21)                                 test_shape: tuple[int, ...]) -> None:
(4)                 error_pattern = "^" + re.escape(
(8)                     f"'aardvark' must be {len(target)}D with shape {shape_repr}, but your input "
(8)                     f"has shape {test_shape}")
(4)                 data = np.zeros(test_shape)
(4)                 with pytest.raises(ValueError, match=error_pattern):
(8)                     _api.check_shape(target, aardvark=data)
(0)             def test_classproperty_deprecation() -> None:
(4)                 class A:
(8)                     @_api.deprecated("0.0.0")
(8)                     @_api.classproperty
(8)                     def f(cls: Self) -> None:
(12)                        pass
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning):
(8)                     A.f
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning):
(8)                     a = A()
(8)                     a.f
(0)             def test_deprecate_privatize_attribute() -> None:
(4)                 class C:
(8)                     def __init__(self) -> None: self._attr = 1
(8)                     def _meth(self, arg: T) -> T: return arg
(8)                     attr: int = _api.deprecate_privatize_attribute("0.0")
(8)                     meth: Callable = _api.deprecate_privatize_attribute("0.0")
(4)                 c = C()
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning):
(8)                     assert c.attr == 1
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning):
(8)                     c.attr = 2
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning):
(8)                     assert c.attr == 2
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning):
(8)                     assert c.meth(42) == 42
(0)             def test_delete_parameter() -> None:
(4)                 @_api.delete_parameter("3.0", "foo")
(4)                 def func1(foo: Any = None) -> None:
(8)                     pass
(4)                 @_api.delete_parameter("3.0", "foo")
(4)                 def func2(**kwargs: Any) -> None:
(8)                     pass
(4)                 for func in [func1, func2]:  # type: ignore[list-item]
(8)                     func()  # No warning.
(8)                     with pytest.warns(mpl.MatplotlibDeprecationWarning):
(12)                        func(foo="bar")
(4)                 def pyplot_wrapper(foo: Any = _api.deprecation._deprecated_parameter) -> None:
(8)                     func1(foo)
(4)                 pyplot_wrapper()  # No warning.
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning):
(8)                     func(foo="bar")
(0)             def test_make_keyword_only() -> None:
(4)                 @_api.make_keyword_only("3.0", "arg")
(4)                 def func(pre: Any, arg: Any, post: Any = None) -> None:
(8)                     pass
(4)                 func(1, arg=2)  # Check that no warning is emitted.
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning):
(8)                     func(1, 2)
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning):
(8)                     func(1, 2, 3)
(0)             def test_deprecation_alternative() -> None:
(4)                 alternative = "`.f1`, `f2`, `f3(x) <.f3>` or `f4(x)<f4>`"
(4)                 @_api.deprecated("1", alternative=alternative)
(4)                 def f() -> None:
(8)                     pass
(4)                 if f.__doc__ is None:
(8)                     pytest.skip('Documentation is disabled')
```

```
86: (4)                 assert alternative in f.__doc__
87: (0)             def test_empty_check_in_list() -> None:
88: (4)                 with pytest.raises(TypeError, match="No argument to check!"):
89: (8)                     _api.check_in_list(["a"])
```

File 148 - test_arrow_patches.py:

```
1: (0)              import pytest
2: (0)              import platform
3: (0)              import matplotlib.pyplot as plt
4: (0)              from matplotlib.testing.decorators import image_comparison
5: (0)              import matplotlib.patches as mpatches
6: (0)              def draw_arrow(ax, t, r):
7: (4)                  ax.annotate('', xy=(0.5, 0.5 + r), xytext=(0.5, 0.5), size=30,
8: (16)                             arrowprops=dict(arrowstyle=t,
9: (32)                                             fc="b", ec='k'))
10: (0)             @image_comparison(['fancyarrow_test_image'],
11: (18)                              tol=0.012 if platform.machine() == 'arm64' else 0)
12: (0)             def test_fancyarrow():
13: (4)                 r = [0.4, 0.3, 0.2, 0.1, 0]
14: (4)                 t = ["fancy", "simple", mpatches.ArrowStyle.Fancy()]
15: (4)                 fig, axs = plt.subplots(len(t), len(r), squeeze=False,
16: (28)                                        figsize=(8, 4.5), subplot_kw=dict(aspect=1))
17: (4)                 for i_r, r1 in enumerate(r):
18: (8)                     for i_t, t1 in enumerate(t):
19: (12)                        ax = axs[i_t, i_r]
20: (12)                        draw_arrow(ax, t1, r1)
21: (12)                        ax.tick_params(labelleft=False, labelbottom=False)
22: (0)             @image_comparison(['boxarrow_test_image.png'])
23: (0)             def test_boxarrow():
24: (4)                 styles = mpatches.BoxStyle.get_styles()
25: (4)                 n = len(styles)
26: (4)                 spacing = 1.2
27: (4)                 figheight = (n * spacing + .5)
28: (4)                 fig = plt.figure(figsize=(4 / 1.5, figheight / 1.5))
29: (4)                 fontsize = 0.3 * 72
30: (4)                 for i, stylename in enumerate(sorted(styles)):
31: (8)                     fig.text(0.5, ((n - i) * spacing - 0.5)/figheight, stylename,
32: (17)                             ha="center",
33: (17)                             size=fontsize,
34: (17)                             transform=fig.transFigure,
35: (17)                             bbox=dict(boxstyle=stylename, fc="w", ec="k"))
36: (0)             def __prepare_fancyarrow_dpi_cor_test():
37: (4)                 """
38: (4)                 Convenience function that prepares and returns a FancyArrowPatch. It aims
39: (4)                 at being used to test that the size of the arrow head does not depend on
40: (4)                 the DPI value of the exported picture.
41: (4)                 NB: this function *is not* a test in itself!
42: (4)                 """
43: (4)                 fig2 = plt.figure("fancyarrow_dpi_cor_test", figsize=(4, 3), dpi=50)
44: (4)                 ax = fig2.add_subplot()
45: (4)                 ax.set_xlim([0, 1])
46: (4)                 ax.set_ylim([0, 1])
47: (4)                 ax.add_patch(mpatches.FancyArrowPatch(posA=(0.3, 0.4), posB=(0.8, 0.6),
48: (42)                                                      lw=3, arrowstyle='->',
49: (42)                                                      mutation_scale=100))
50: (4)                 return fig2
51: (0)             @image_comparison(['fancyarrow_dpi_cor_100dpi.png'], remove_text=True,
52: (18)                              tol=0 if platform.machine() == 'x86_64' else 0.02,
53: (18)                              savefig_kwarg=dict(dpi=100))
54: (0)             def test_fancyarrow_dpi_cor_100dpi():
55: (4)                 """
56: (4)                 Check the export of a FancyArrowPatch @ 100 DPI. FancyArrowPatch is
57: (4)                 instantiated through a dedicated function because another similar test
58: (4)                 checks a similar export but with a different DPI value.
59: (4)                 Remark: test only a rasterized format.
60: (4)                 """
```

```
 61: (4)                 __prepare_fancyarrow_dpi_cor_test()
 62: (0)             @image_comparison(['fancyarrow_dpi_cor_200dpi.png'], remove_text=True,
 63: (18)                            tol=0 if platform.machine() == 'x86_64' else 0.02,
 64: (18)                            savefig_kwarg=dict(dpi=200))
 65: (0)             def test_fancyarrow_dpi_cor_200dpi():
 66: (4)                 """
 67: (4)                 As test_fancyarrow_dpi_cor_100dpi, but exports @ 200 DPI. The relative
 size
 68: (4)                 of the arrow head should be the same.
 69: (4)                 """
 70: (4)                 __prepare_fancyarrow_dpi_cor_test()
 71: (0)             @image_comparison(['fancyarrow_dash.png'], remove_text=True, style='default')
 72: (0)             def test_fancyarrow_dash():
 73: (4)                 fig, ax = plt.subplots()
 74: (4)                 e = mpatches.FancyArrowPatch((0, 0), (0.5, 0.5),
 75: (33)                                             arrowstyle='-|>',
 76: (33)                                             connectionstyle='angle3,angleA=0,angleB=90',
 77: (33)                                             mutation_scale=10.0,
 78: (33)                                             linewidth=2,
 79: (33)                                             linestyle='dashed',
 80: (33)                                             color='k')
 81: (4)                 e2 = mpatches.FancyArrowPatch((0, 0), (0.5, 0.5),
 82: (34)                                              arrowstyle='-|>',
 83: (34)                                              connectionstyle='angle3',
 84: (34)                                              mutation_scale=10.0,
 85: (34)                                              linewidth=2,
 86: (34)                                              linestyle='dotted',
 87: (34)                                              color='k')
 88: (4)                 ax.add_patch(e)
 89: (4)                 ax.add_patch(e2)
 90: (0)             @image_comparison(['arrow_styles.png'], style='mpl20', remove_text=True,
 91: (18)                            tol=0 if platform.machine() == 'x86_64' else 0.02)
 92: (0)             def test_arrow_styles():
 93: (4)                 styles = mpatches.ArrowStyle.get_styles()
 94: (4)                 n = len(styles)
 95: (4)                 fig, ax = plt.subplots(figsize=(8, 8))
 96: (4)                 ax.set_xlim(0, 1)
 97: (4)                 ax.set_ylim(-1, n)
 98: (4)                 fig.subplots_adjust(left=0, right=1, bottom=0, top=1)
 99: (4)                 for i, stylename in enumerate(sorted(styles)):
 100: (8)                    patch = mpatches.FancyArrowPatch((0.1 + (i % 2)*0.05, i),
 101: (41)                                                    (0.45 + (i % 2)*0.05, i),
 102: (41)                                                    arrowstyle=stylename,
 103: (41)                                                    mutation_scale=25)
 104: (8)                    ax.add_patch(patch)
 105: (4)                for i, stylename in enumerate([']-[', ']-', '-[', '|-|']):
 106: (8)                    style = stylename
 107: (8)                    if stylename[0] != '-':
 108: (12)                       style += ',angleA=ANGLE'
 109: (8)                    if stylename[-1] != '-':
 110: (12)                       style += ',angleB=ANGLE'
 111: (8)                    for j, angle in enumerate([-30, 60]):
 112: (12)                       arrowstyle = style.replace('ANGLE', str(angle))
 113: (12)                       patch = mpatches.FancyArrowPatch((0.55, 2*i + j), (0.9, 2*i + j),
 114: (45)                                                        arrowstyle=arrowstyle,
 115: (45)                                                        mutation_scale=25)
 116: (12)                       ax.add_patch(patch)
 117: (0)            @image_comparison(['connection_styles.png'], style='mpl20', remove_text=True,
 118: (18)                           tol=0.013 if platform.machine() == 'arm64' else 0)
 119: (0)            def test_connection_styles():
 120: (4)                styles = mpatches.ConnectionStyle.get_styles()
 121: (4)                n = len(styles)
 122: (4)                fig, ax = plt.subplots(figsize=(6, 10))
 123: (4)                ax.set_xlim(0, 1)
 124: (4)                ax.set_ylim(-1, n)
 125: (4)                for i, stylename in enumerate(sorted(styles)):
 126: (8)                    patch = mpatches.FancyArrowPatch((0.1, i), (0.8, i + 0.5),
 127: (41)                                                    arrowstyle="->",
 128: (41)                                                    connectionstyle=stylename,
```

```
129: (41)                                                   mutation_scale=25)
130: (8)                    ax.add_patch(patch)
131: (0)            def test_invalid_intersection():
132: (4)                conn_style_1 = mpatches.ConnectionStyle.Angle3(angleA=20, angleB=200)
133: (4)                p1 = mpatches.FancyArrowPatch((.2, .2), (.5, .5),
134: (34)                                       connectionstyle=conn_style_1)
135: (4)                with pytest.raises(ValueError):
136: (8)                    plt.gca().add_patch(p1)
137: (4)                conn_style_2 = mpatches.ConnectionStyle.Angle3(angleA=20, angleB=199.9)
138: (4)                p2 = mpatches.FancyArrowPatch((.2, .2), (.5, .5),
139: (34)                                       connectionstyle=conn_style_2)
140: (4)                plt.gca().add_patch(p2)

----------------------------------------

File 149 - test_artist.py:

1: (0)              import io
2: (0)              from itertools import chain
3: (0)              import numpy as np
4: (0)              import pytest
5: (0)              import matplotlib.colors as mcolors
6: (0)              import matplotlib.pyplot as plt
7: (0)              import matplotlib.patches as mpatches
8: (0)              import matplotlib.lines as mlines
9: (0)              import matplotlib.path as mpath
10: (0)             import matplotlib.transforms as mtransforms
11: (0)             import matplotlib.collections as mcollections
12: (0)             import matplotlib.artist as martist
13: (0)             import matplotlib.backend_bases as mbackend_bases
14: (0)             import matplotlib as mpl
15: (0)             from matplotlib.testing.decorators import check_figures_equal,
image_comparison
16: (0)             def test_patch_transform_of_none():
17: (4)                 ax = plt.axes()
18: (4)                 ax.set_xlim(1, 3)
19: (4)                 ax.set_ylim(1, 3)
20: (4)                 xy_data = (2, 2)
21: (4)                 xy_pix = ax.transData.transform(xy_data)
22: (4)                 e = mpatches.Ellipse(xy_data, width=1, height=1, fc='yellow', alpha=0.5)
23: (4)                 ax.add_patch(e)
24: (4)                 assert e._transform == ax.transData
25: (4)                 e = mpatches.Ellipse(xy_pix, width=120, height=120, fc='coral',
26: (25)                                     transform=None, alpha=0.5)
27: (4)                 assert e.is_transform_set()
28: (4)                 ax.add_patch(e)
29: (4)                 assert isinstance(e._transform, mtransforms.IdentityTransform)
30: (4)                 e = mpatches.Ellipse(xy_pix, width=100, height=100,
31: (25)                                     transform=mtransforms.IdentityTransform(), alpha=0.5)
32: (4)                 ax.add_patch(e)
33: (4)                 assert isinstance(e._transform, mtransforms.IdentityTransform)
34: (4)                 e = mpatches.Ellipse(xy_pix, width=120, height=120, fc='coral',
35: (25)                                     alpha=0.5)
36: (4)                 intermediate_transform = e.get_transform()
37: (4)                 assert not e.is_transform_set()
38: (4)                 ax.add_patch(e)
39: (4)                 assert e.get_transform() != intermediate_transform
40: (4)                 assert e.is_transform_set()
41: (4)                 assert e._transform == ax.transData
42: (0)             def test_collection_transform_of_none():
43: (4)                 ax = plt.axes()
44: (4)                 ax.set_xlim(1, 3)
45: (4)                 ax.set_ylim(1, 3)
46: (4)                 xy_data = (2, 2)
47: (4)                 xy_pix = ax.transData.transform(xy_data)
48: (4)                 e = mpatches.Ellipse(xy_data, width=1, height=1)
49: (4)                 c = mcollections.PatchCollection([e], facecolor='yellow', alpha=0.5)
50: (4)                 ax.add_collection(c)
51: (4)                 assert c.get_offset_transform() + c.get_transform() == ax.transData
```

```
52: (4)                 e = mpatches.Ellipse(xy_pix, width=120, height=120)
53: (4)                 c = mcollections.PatchCollection([e], facecolor='coral',
54: (37)                                                 alpha=0.5)
55: (4)                 c.set_transform(None)
56: (4)                 ax.add_collection(c)
57: (4)                 assert isinstance(c.get_transform(), mtransforms.IdentityTransform)
58: (4)                 e = mpatches.Ellipse(xy_pix, width=100, height=100)
59: (4)                 c = mcollections.PatchCollection([e],
60: (37)
transform=mtransforms.IdentityTransform(),
61: (37)                                                 alpha=0.5)
62: (4)                 ax.add_collection(c)
63: (4)                 assert isinstance(c.get_offset_transform(), mtransforms.IdentityTransform)
64: (0)             @image_comparison(["clip_path_clipping"], remove_text=True)
65: (0)             def test_clipping():
66: (4)                 exterior = mpath.Path.unit_rectangle().deepcopy()
67: (4)                 exterior.vertices *= 4
68: (4)                 exterior.vertices -= 2
69: (4)                 interior = mpath.Path.unit_circle().deepcopy()
70: (4)                 interior.vertices = interior.vertices[::-1]
71: (4)                 clip_path = mpath.Path.make_compound_path(exterior, interior)
72: (4)                 star = mpath.Path.unit_regular_star(6).deepcopy()
73: (4)                 star.vertices *= 2.6
74: (4)                 fig, (ax1, ax2) = plt.subplots(1, 2, sharex=True, sharey=True)
75: (4)                 col = mcollections.PathCollection([star], lw=5, edgecolor='blue',
76: (38)                                                  facecolor='red', alpha=0.7, hatch='*')
77: (4)                 col.set_clip_path(clip_path, ax1.transData)
78: (4)                 ax1.add_collection(col)
79: (4)                 patch = mpatches.PathPatch(star, lw=5, edgecolor='blue', facecolor='red',
80: (31)                                           alpha=0.7, hatch='*')
81: (4)                 patch.set_clip_path(clip_path, ax2.transData)
82: (4)                 ax2.add_patch(patch)
83: (4)                 ax1.set_xlim([-3, 3])
84: (4)                 ax1.set_ylim([-3, 3])
85: (0)             @check_figures_equal(extensions=['png'])
86: (0)             def test_clipping_zoom(fig_test, fig_ref):
87: (4)                 ax_test = fig_test.add_axes([0, 0, 1, 1])
88: (4)                 l, = ax_test.plot([-3, 3], [-3, 3])
89: (4)                 p = mpath.Path([[0, 0], [1, 0], [1, 1], [0, 1], [0, 0]])
90: (4)                 p = mpatches.PathPatch(p, transform=ax_test.transData)
91: (4)                 l.set_clip_path(p)
92: (4)                 ax_ref = fig_ref.add_axes([0, 0, 1, 1])
93: (4)                 ax_ref.plot([-3, 3], [-3, 3])
94: (4)                 ax_ref.set(xlim=(0.5, 0.75), ylim=(0.5, 0.75))
95: (4)                 ax_test.set(xlim=(0.5, 0.75), ylim=(0.5, 0.75))
96: (0)             def test_cull_markers():
97: (4)                 x = np.random.random(20000)
98: (4)                 y = np.random.random(20000)
99: (4)                 fig, ax = plt.subplots()
100: (4)                ax.plot(x, y, 'k.')
101: (4)                ax.set_xlim(2, 3)
102: (4)                pdf = io.BytesIO()
103: (4)                fig.savefig(pdf, format="pdf")
104: (4)                assert len(pdf.getvalue()) < 8000
105: (4)                svg = io.BytesIO()
106: (4)                fig.savefig(svg, format="svg")
107: (4)                assert len(svg.getvalue()) < 20000
108: (0)            @image_comparison(['hatching'], remove_text=True, style='default')
109: (0)            def test_hatching():
110: (4)                fig, ax = plt.subplots(1, 1)
111: (4)                rect1 = mpatches.Rectangle((0, 0), 3, 4, hatch='/')
112: (4)                ax.add_patch(rect1)
113: (4)                rect2 = mcollections.RegularPolyCollection(
114: (8)                    4, sizes=[16000], offsets=[(1.5, 6.5)], offset_transform=ax.transData,
115: (8)                    hatch='/')
116: (4)                ax.add_collection(rect2)
117: (4)                rect3 = mpatches.Rectangle((4, 0), 3, 4, hatch='/', edgecolor='C1')
118: (4)                ax.add_patch(rect3)
119: (4)                rect4 = mcollections.RegularPolyCollection(
```

```
120: (8)                    4, sizes=[16000], offsets=[(5.5, 6.5)], offset_transform=ax.transData,
121: (8)                    hatch='/', edgecolor='C1')
122: (4)                ax.add_collection(rect4)
123: (4)                ax.set_xlim(0, 7)
124: (4)                ax.set_ylim(0, 9)
125: (0)            def test_remove():
126: (4)                fig, ax = plt.subplots()
127: (4)                im = ax.imshow(np.arange(36).reshape(6, 6))
128: (4)                ln, = ax.plot(range(5))
129: (4)                assert fig.stale
130: (4)                assert ax.stale
131: (4)                fig.canvas.draw()
132: (4)                assert not fig.stale
133: (4)                assert not ax.stale
134: (4)                assert not ln.stale
135: (4)                assert im in ax._mouseover_set
136: (4)                assert ln not in ax._mouseover_set
137: (4)                assert im.axes is ax
138: (4)                im.remove()
139: (4)                ln.remove()
140: (4)                for art in [im, ln]:
141: (8)                    assert art.axes is None
142: (8)                    assert art.figure is None
143: (4)                assert im not in ax._mouseover_set
144: (4)                assert fig.stale
145: (4)                assert ax.stale
146: (0)            @image_comparison(["default_edges.png"], remove_text=True, style='default')
147: (0)            def test_default_edges():
148: (4)                plt.rcParams['text.kerning_factor'] = 6
149: (4)                fig, [[ax1, ax2], [ax3, ax4]] = plt.subplots(2, 2)
150: (4)                ax1.plot(np.arange(10), np.arange(10), 'x',
151: (13)                         np.arange(10) + 1, np.arange(10), 'o')
152: (4)                ax2.bar(np.arange(10), np.arange(10), align='edge')
153: (4)                ax3.text(0, 0, "BOX", size=24, bbox=dict(boxstyle='sawtooth'))
154: (4)                ax3.set_xlim((-1, 1))
155: (4)                ax3.set_ylim((-1, 1))
156: (4)                pp1 = mpatches.PathPatch(
157: (8)                    mpath.Path([(0, 0), (1, 0), (1, 1), (0, 0)],
158: (19)                               [mpath.Path.MOVETO, mpath.Path.CURVE3,
159: (20)                                mpath.Path.CURVE3, mpath.Path.CLOSEPOLY]),
160: (8)                    fc="none", transform=ax4.transData)
161: (4)                ax4.add_patch(pp1)
162: (0)            def test_properties():
163: (4)                ln = mlines.Line2D([], [])
164: (4)                ln.properties()  # Check that no warning is emitted.
165: (0)            def test_setp():
166: (4)                plt.setp([])
167: (4)                plt.setp([[]])
168: (4)                fig, ax = plt.subplots()
169: (4)                lines1 = ax.plot(range(3))
170: (4)                lines2 = ax.plot(range(3))
171: (4)                martist.setp(chain(lines1, lines2), 'lw', 5)
172: (4)                plt.setp(ax.spines.values(), color='green')
173: (4)                sio = io.StringIO()
174: (4)                plt.setp(lines1, 'zorder', file=sio)
175: (4)                assert sio.getvalue() == '  zorder: float\n'
176: (0)            def test_None_zorder():
177: (4)                fig, ax = plt.subplots()
178: (4)                ln, = ax.plot(range(5), zorder=None)
179: (4)                assert ln.get_zorder() == mlines.Line2D.zorder
180: (4)                ln.set_zorder(123456)
181: (4)                assert ln.get_zorder() == 123456
182: (4)                ln.set_zorder(None)
183: (4)                assert ln.get_zorder() == mlines.Line2D.zorder
184: (0)            @pytest.mark.parametrize('accept_clause, expected', [
185: (4)                ('', 'unknown'),
186: (4)                ("ACCEPTS: [ '-' | '--' | '-.' ]", "[ '-' | '--' | '-.' ]"),
187: (4)                ('ACCEPTS: Some description.', 'Some description.'),
188: (4)                ('.. ACCEPTS: Some description.', 'Some description.'),
```

```
(4)                 ('arg : int', 'int'),
(4)                 ('*arg : int', 'int'),
(4)                 ('arg : int\nACCEPTS: Something else.', 'Something else. '),
(0)             ])
(0)             def test_artist_inspector_get_valid_values(accept_clause, expected):
(4)                 class TestArtist(martist.Artist):
(8)                     def set_f(self, arg):
(12)                        pass
(4)                 TestArtist.set_f.__doc__ = """
(4)                 Some text.
(4)                 %s
(4)                 """ % accept_clause
(4)                 valid_values = martist.ArtistInspector(TestArtist).get_valid_values('f')
(4)                 assert valid_values == expected
(0)             def test_artist_inspector_get_aliases():
(4)                 ai = martist.ArtistInspector(mlines.Line2D)
(4)                 aliases = ai.get_aliases()
(4)                 assert aliases["linewidth"] == {"lw"}
(0)             def test_set_alpha():
(4)                 art = martist.Artist()
(4)                 with pytest.raises(TypeError, match='^alpha must be numeric or None'):
(8)                     art.set_alpha('string')
(4)                 with pytest.raises(TypeError, match='^alpha must be numeric or None'):
(8)                     art.set_alpha([1, 2, 3])
(4)                 with pytest.raises(ValueError, match="outside 0-1 range"):
(8)                     art.set_alpha(1.1)
(4)                 with pytest.raises(ValueError, match="outside 0-1 range"):
(8)                     art.set_alpha(np.nan)
(0)             def test_set_alpha_for_array():
(4)                 art = martist.Artist()
(4)                 with pytest.raises(TypeError, match='^alpha must be numeric or None'):
(8)                     art._set_alpha_for_array('string')
(4)                 with pytest.raises(ValueError, match="outside 0-1 range"):
(8)                     art._set_alpha_for_array(1.1)
(4)                 with pytest.raises(ValueError, match="outside 0-1 range"):
(8)                     art._set_alpha_for_array(np.nan)
(4)                 with pytest.raises(ValueError, match="alpha must be between 0 and 1"):
(8)                     art._set_alpha_for_array([0.5, 1.1])
(4)                 with pytest.raises(ValueError, match="alpha must be between 0 and 1"):
(8)                     art._set_alpha_for_array([0.5, np.nan])
(0)             def test_callbacks():
(4)                 def func(artist):
(8)                     func.counter += 1
(4)                 func.counter = 0
(4)                 art = martist.Artist()
(4)                 oid = art.add_callback(func)
(4)                 assert func.counter == 0
(4)                 art.pchanged()  # must call the callback
(4)                 assert func.counter == 1
(4)                 art.set_zorder(10)  # setting a property must also call the callback
(4)                 assert func.counter == 2
(4)                 art.remove_callback(oid)
(4)                 art.pchanged()  # must not call the callback anymore
(4)                 assert func.counter == 2
(0)             def test_set_signature():
(4)                 """Test autogenerated ``set()`` for Artist subclasses."""
(4)                 class MyArtist1(martist.Artist):
(8)                     def set_myparam1(self, val):
(12)                        pass
(4)                 assert hasattr(MyArtist1.set, '_autogenerated_signature')
(4)                 assert 'myparam1' in MyArtist1.set.__doc__
(4)                 class MyArtist2(MyArtist1):
(8)                     def set_myparam2(self, val):
(12)                        pass
(4)                 assert hasattr(MyArtist2.set, '_autogenerated_signature')
(4)                 assert 'myparam1' in MyArtist2.set.__doc__
(4)                 assert 'myparam2' in MyArtist2.set.__doc__
(0)             def test_set_is_overwritten():
(4)                 """set() defined in Artist subclasses should not be overwritten."""
```

```
(4)                 class MyArtist3(martist.Artist):
(8)                     def set(self, **kwargs):
(12)                        """Not overwritten."""
(4)                 assert not hasattr(MyArtist3.set, '_autogenerated_signature')
(4)                 assert MyArtist3.set.__doc__ == "Not overwritten."
(4)                 class MyArtist4(MyArtist3):
(8)                     pass
(4)                 assert MyArtist4.set is MyArtist3.set
(0)             def test_format_cursor_data_BoundaryNorm():
(4)                 """Test if cursor data is correct when using BoundaryNorm."""
(4)                 X = np.empty((3, 3))
(4)                 X[0, 0] = 0.9
(4)                 X[0, 1] = 0.99
(4)                 X[0, 2] = 0.999
(4)                 X[1, 0] = -1
(4)                 X[1, 1] = 0
(4)                 X[1, 2] = 1
(4)                 X[2, 0] = 0.09
(4)                 X[2, 1] = 0.009
(4)                 X[2, 2] = 0.0009
(4)                 fig, ax = plt.subplots()
(4)                 fig.suptitle("-1..1 to 0..256 in 0.1")
(4)                 norm = mcolors.BoundaryNorm(np.linspace(-1, 1, 20), 256)
(4)                 img = ax.imshow(X, cmap='RdBu_r', norm=norm)
(4)                 labels_list = [
(8)                     "[0.9]",
(8)                     "[1.]",
(8)                     "[1.]",
(8)                     "[-1.0]",
(8)                     "[0.0]",
(8)                     "[1.0]",
(8)                     "[0.09]",
(8)                     "[0.009]",
(8)                     "[0.0009]",
(4)                 ]
(4)                 for v, label in zip(X.flat, labels_list):
(8)                     assert img.format_cursor_data(v) == label
(4)                 plt.close()
(4)                 fig, ax = plt.subplots()
(4)                 fig.suptitle("-1..1 to 0..256 in 0.01")
(4)                 cmap = mpl.colormaps['RdBu_r'].resampled(200)
(4)                 norm = mcolors.BoundaryNorm(np.linspace(-1, 1, 200), 200)
(4)                 img = ax.imshow(X, cmap=cmap, norm=norm)
(4)                 labels_list = [
(8)                     "[0.90]",
(8)                     "[0.99]",
(8)                     "[1.0]",
(8)                     "[-1.00]",
(8)                     "[0.00]",
(8)                     "[1.00]",
(8)                     "[0.09]",
(8)                     "[0.009]",
(8)                     "[0.0009]",
(4)                 ]
(4)                 for v, label in zip(X.flat, labels_list):
(8)                     assert img.format_cursor_data(v) == label
(4)                 plt.close()
(4)                 fig, ax = plt.subplots()
(4)                 fig.suptitle("-1..1 to 0..256 in 0.001")
(4)                 cmap = mpl.colormaps['RdBu_r'].resampled(2000)
(4)                 norm = mcolors.BoundaryNorm(np.linspace(-1, 1, 2000), 2000)
(4)                 img = ax.imshow(X, cmap=cmap, norm=norm)
(4)                 labels_list = [
(8)                     "[0.900]",
(8)                     "[0.990]",
(8)                     "[0.999]",
(8)                     "[-1.000]",
(8)                     "[0.000]",
(8)                     "[1.000]",
```

```
327: (8)                    "[0.090]",
328: (8)                    "[0.009]",
329: (8)                    "[0.0009]",
330: (4)                ]
331: (4)                for v, label in zip(X.flat, labels_list):
332: (8)                    assert img.format_cursor_data(v) == label
333: (4)                plt.close()
334: (4)                X = np.empty((7, 1))
335: (4)                X[0] = -1.0
336: (4)                X[1] = 0.0
337: (4)                X[2] = 0.1
338: (4)                X[3] = 0.5
339: (4)                X[4] = 0.9
340: (4)                X[5] = 1.0
341: (4)                X[6] = 2.0
342: (4)                labels_list = [
343: (8)                    "[-1.0]",
344: (8)                    "[0.0]",
345: (8)                    "[0.1]",
346: (8)                    "[0.5]",
347: (8)                    "[0.9]",
348: (8)                    "[1.0]",
349: (8)                    "[2.0]",
350: (4)                ]
351: (4)                fig, ax = plt.subplots()
352: (4)                fig.suptitle("noclip, neither")
353: (4)                norm = mcolors.BoundaryNorm(
354: (8)                    np.linspace(0, 1, 4, endpoint=True), 256, clip=False,
extend='neither')
355: (4)                img = ax.imshow(X, cmap='RdBu_r', norm=norm)
356: (4)                for v, label in zip(X.flat, labels_list):
357: (8)                    assert img.format_cursor_data(v) == label
358: (4)                plt.close()
359: (4)                fig, ax = plt.subplots()
360: (4)                fig.suptitle("noclip, min")
361: (4)                norm = mcolors.BoundaryNorm(
362: (8)                    np.linspace(0, 1, 4, endpoint=True), 256, clip=False, extend='min')
363: (4)                img = ax.imshow(X, cmap='RdBu_r', norm=norm)
364: (4)                for v, label in zip(X.flat, labels_list):
365: (8)                    assert img.format_cursor_data(v) == label
366: (4)                plt.close()
367: (4)                fig, ax = plt.subplots()
368: (4)                fig.suptitle("noclip, max")
369: (4)                norm = mcolors.BoundaryNorm(
370: (8)                    np.linspace(0, 1, 4, endpoint=True), 256, clip=False, extend='max')
371: (4)                img = ax.imshow(X, cmap='RdBu_r', norm=norm)
372: (4)                for v, label in zip(X.flat, labels_list):
373: (8)                    assert img.format_cursor_data(v) == label
374: (4)                plt.close()
375: (4)                fig, ax = plt.subplots()
376: (4)                fig.suptitle("noclip, both")
377: (4)                norm = mcolors.BoundaryNorm(
378: (8)                    np.linspace(0, 1, 4, endpoint=True), 256, clip=False, extend='both')
379: (4)                img = ax.imshow(X, cmap='RdBu_r', norm=norm)
380: (4)                for v, label in zip(X.flat, labels_list):
381: (8)                    assert img.format_cursor_data(v) == label
382: (4)                plt.close()
383: (4)                fig, ax = plt.subplots()
384: (4)                fig.suptitle("clip, neither")
385: (4)                norm = mcolors.BoundaryNorm(
386: (8)                    np.linspace(0, 1, 4, endpoint=True), 256, clip=True, extend='neither')
387: (4)                img = ax.imshow(X, cmap='RdBu_r', norm=norm)
388: (4)                for v, label in zip(X.flat, labels_list):
389: (8)                    assert img.format_cursor_data(v) == label
390: (4)                plt.close()
391: (0)            def test_auto_no_rasterize():
392: (4)                class Gen1(martist.Artist):
393: (8)                    ...
394: (4)                assert 'draw' in Gen1.__dict__
```

```
395: (4)                assert Gen1.__dict__['draw'] is Gen1.draw
396: (4)                class Gen2(Gen1):
397: (8)                    ...
398: (4)                assert 'draw' not in Gen2.__dict__
399: (4)                assert Gen2.draw is Gen1.draw
400: (0)            def test_draw_wraper_forward_input():
401: (4)                class TestKlass(martist.Artist):
402: (8)                    def draw(self, renderer, extra):
403: (12)                       return extra
404: (4)                art = TestKlass()
405: (4)                renderer = mbackend_bases.RendererBase()
406: (4)                assert 'aardvark' == art.draw(renderer, 'aardvark')
407: (4)                assert 'aardvark' == art.draw(renderer, extra='aardvark')
```

File 150 - test_axes.py:

```
1: (0)              import contextlib
2: (0)              from collections import namedtuple
3: (0)              import datetime
4: (0)              from decimal import Decimal
5: (0)              from functools import partial
6: (0)              import inspect
7: (0)              import io
8: (0)              from itertools import product
9: (0)              import platform
10: (0)             from types import SimpleNamespace
11: (0)             import dateutil.tz
12: (0)             import numpy as np
13: (0)             from numpy import ma
14: (0)             from cycler import cycler
15: (0)             import pytest
16: (0)             import matplotlib
17: (0)             import matplotlib as mpl
18: (0)             from matplotlib import rc_context, patheffects
19: (0)             import matplotlib.colors as mcolors
20: (0)             import matplotlib.dates as mdates
21: (0)             from matplotlib.figure import Figure
22: (0)             from matplotlib.axes import Axes
23: (0)             import matplotlib.font_manager as mfont_manager
24: (0)             import matplotlib.markers as mmarkers
25: (0)             import matplotlib.patches as mpatches
26: (0)             import matplotlib.path as mpath
27: (0)             from matplotlib.projections.geo import HammerAxes
28: (0)             from matplotlib.projections.polar import PolarAxes
29: (0)             import matplotlib.pyplot as plt
30: (0)             import matplotlib.text as mtext
31: (0)             import matplotlib.ticker as mticker
32: (0)             import matplotlib.transforms as mtransforms
33: (0)             import mpl_toolkits.axisartist as AA  # type: ignore
34: (0)             from numpy.testing import (
35: (4)                 assert_allclose, assert_array_equal, assert_array_almost_equal)
36: (0)             from matplotlib.testing.decorators import (
37: (4)                 image_comparison, check_figures_equal, remove_ticks_and_titles)
38: (0)             from matplotlib.testing._markers import needs_usetex
39: (0)             @check_figures_equal(extensions=["png"])
40: (0)             def test_invisible_axes(fig_test, fig_ref):
41: (4)                 ax = fig_test.subplots()
42: (4)                 ax.set_visible(False)
43: (0)             def test_get_labels():
44: (4)                 fig, ax = plt.subplots()
45: (4)                 ax.set_xlabel('x label')
46: (4)                 ax.set_ylabel('y label')
47: (4)                 assert ax.get_xlabel() == 'x label'
48: (4)                 assert ax.get_ylabel() == 'y label'
49: (0)             def test_repr():
50: (4)                 fig, ax = plt.subplots()
51: (4)                 ax.set_label('label')
```

```
(4)                 ax.set_title('title')
(4)                 ax.set_xlabel('x')
(4)                 ax.set_ylabel('y')
(4)                 assert repr(ax) == (
(8)                     "<Axes: "
(8)                     "label='label', title={'center': 'title'}, xlabel='x', ylabel='y'>")
(0)             @check_figures_equal()
(0)             def test_label_loc_vertical(fig_test, fig_ref):
(4)                 ax = fig_test.subplots()
(4)                 sc = ax.scatter([1, 2], [1, 2], c=[1, 2], label='scatter')
(4)                 ax.legend()
(4)                 ax.set_ylabel('Y Label', loc='top')
(4)                 ax.set_xlabel('X Label', loc='right')
(4)                 cbar = fig_test.colorbar(sc)
(4)                 cbar.set_label("Z Label", loc='top')
(4)                 ax = fig_ref.subplots()
(4)                 sc = ax.scatter([1, 2], [1, 2], c=[1, 2], label='scatter')
(4)                 ax.legend()
(4)                 ax.set_ylabel('Y Label', y=1, ha='right')
(4)                 ax.set_xlabel('X Label', x=1, ha='right')
(4)                 cbar = fig_ref.colorbar(sc)
(4)                 cbar.set_label("Z Label", y=1, ha='right')
(0)             @check_figures_equal()
(0)             def test_label_loc_horizontal(fig_test, fig_ref):
(4)                 ax = fig_test.subplots()
(4)                 sc = ax.scatter([1, 2], [1, 2], c=[1, 2], label='scatter')
(4)                 ax.legend()
(4)                 ax.set_ylabel('Y Label', loc='bottom')
(4)                 ax.set_xlabel('X Label', loc='left')
(4)                 cbar = fig_test.colorbar(sc, orientation='horizontal')
(4)                 cbar.set_label("Z Label", loc='left')
(4)                 ax = fig_ref.subplots()
(4)                 sc = ax.scatter([1, 2], [1, 2], c=[1, 2], label='scatter')
(4)                 ax.legend()
(4)                 ax.set_ylabel('Y Label', y=0, ha='left')
(4)                 ax.set_xlabel('X Label', x=0, ha='left')
(4)                 cbar = fig_ref.colorbar(sc, orientation='horizontal')
(4)                 cbar.set_label("Z Label", x=0, ha='left')
(0)             @check_figures_equal()
(0)             def test_label_loc_rc(fig_test, fig_ref):
(4)                 with matplotlib.rc_context({"xaxis.labellocation": "right",
(32)                                            "yaxis.labellocation": "top"}):
(8)                     ax = fig_test.subplots()
(8)                     sc = ax.scatter([1, 2], [1, 2], c=[1, 2], label='scatter')
(8)                     ax.legend()
(8)                     ax.set_ylabel('Y Label')
(8)                     ax.set_xlabel('X Label')
(8)                     cbar = fig_test.colorbar(sc, orientation='horizontal')
(8)                     cbar.set_label("Z Label")
(4)                 ax = fig_ref.subplots()
(4)                 sc = ax.scatter([1, 2], [1, 2], c=[1, 2], label='scatter')
(4)                 ax.legend()
(4)                 ax.set_ylabel('Y Label', y=1, ha='right')
(4)                 ax.set_xlabel('X Label', x=1, ha='right')
(4)                 cbar = fig_ref.colorbar(sc, orientation='horizontal')
(4)                 cbar.set_label("Z Label", x=1, ha='right')
(0)             def test_label_shift():
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_xlabel("Test label", loc="left")
(4)                 ax.set_xlabel("Test label", loc="center")
(4)                 assert ax.xaxis.get_label().get_horizontalalignment() == "center"
(4)                 ax.set_xlabel("Test label", loc="right")
(4)                 assert ax.xaxis.get_label().get_horizontalalignment() == "right"
(4)                 ax.set_xlabel("Test label", loc="center")
(4)                 assert ax.xaxis.get_label().get_horizontalalignment() == "center"
(4)                 ax.set_ylabel("Test label", loc="top")
(4)                 ax.set_ylabel("Test label", loc="center")
(4)                 assert ax.yaxis.get_label().get_horizontalalignment() == "center"
(4)                 ax.set_ylabel("Test label", loc="bottom")
```

```
121: (4)                assert ax.yaxis.get_label().get_horizontalalignment() == "left"
122: (4)                ax.set_ylabel("Test label", loc="center")
123: (4)                assert ax.yaxis.get_label().get_horizontalalignment() == "center"
124: (0)            @check_figures_equal(extensions=["png"])
125: (0)            def test_acorr(fig_test, fig_ref):
126: (4)                np.random.seed(19680801)
127: (4)                Nx = 512
128: (4)                x = np.random.normal(0, 1, Nx).cumsum()
129: (4)                maxlags = Nx-1
130: (4)                ax_test = fig_test.subplots()
131: (4)                ax_test.acorr(x, maxlags=maxlags)
132: (4)                ax_ref = fig_ref.subplots()
133: (4)                norm_auto_corr = np.correlate(x, x, mode="full")/np.dot(x, x)
134: (4)                lags = np.arange(-maxlags, maxlags+1)
135: (4)                norm_auto_corr = norm_auto_corr[Nx-1-maxlags:Nx+maxlags]
136: (4)                ax_ref.vlines(lags, [0], norm_auto_corr)
137: (4)                ax_ref.axhline(y=0, xmin=0, xmax=1)
138: (0)            @check_figures_equal(extensions=["png"])
139: (0)            def test_acorr_integers(fig_test, fig_ref):
140: (4)                np.random.seed(19680801)
141: (4)                Nx = 51
142: (4)                x = (np.random.rand(Nx) * 10).cumsum()
143: (4)                x = (np.ceil(x)).astype(np.int64)
144: (4)                maxlags = Nx-1
145: (4)                ax_test = fig_test.subplots()
146: (4)                ax_test.acorr(x, maxlags=maxlags)
147: (4)                ax_ref = fig_ref.subplots()
148: (4)                norm_auto_corr = np.correlate(x, x, mode="full")/np.dot(x, x)
149: (4)                lags = np.arange(-maxlags, maxlags+1)
150: (4)                norm_auto_corr = norm_auto_corr[Nx-1-maxlags:Nx+maxlags]
151: (4)                ax_ref.vlines(lags, [0], norm_auto_corr)
152: (4)                ax_ref.axhline(y=0, xmin=0, xmax=1)
153: (0)            @check_figures_equal(extensions=["png"])
154: (0)            def test_spy(fig_test, fig_ref):
155: (4)                np.random.seed(19680801)
156: (4)                a = np.ones(32 * 32)
157: (4)                a[:16 * 32] = 0
158: (4)                np.random.shuffle(a)
159: (4)                a = a.reshape((32, 32))
160: (4)                axs_test = fig_test.subplots(2)
161: (4)                axs_test[0].spy(a)
162: (4)                axs_test[1].spy(a, marker=".", origin="lower")
163: (4)                axs_ref = fig_ref.subplots(2)
164: (4)                axs_ref[0].imshow(a, cmap="gray_r", interpolation="nearest")
165: (4)                axs_ref[0].xaxis.tick_top()
166: (4)                axs_ref[1].plot(*np.nonzero(a)[::-1], ".", markersize=10)
167: (4)                axs_ref[1].set(
168: (8)                    aspect=1, xlim=axs_ref[0].get_xlim(), ylim=axs_ref[0].get_ylim()
[::-1])
169: (4)                for ax in axs_ref:
170: (8)                    ax.xaxis.set_ticks_position("both")
171: (0)            def test_spy_invalid_kwargs():
172: (4)                fig, ax = plt.subplots()
173: (4)                for unsupported_kw in [{'interpolation': 'nearest'},
174: (27)                                       {'marker': 'o', 'linestyle': 'solid'}]:
175: (8)                    with pytest.raises(TypeError):
176: (12)                       ax.spy(np.eye(3, 3), **unsupported_kw)
177: (0)            @check_figures_equal(extensions=["png"])
178: (0)            def test_matshow(fig_test, fig_ref):
179: (4)                mpl.style.use("mpl20")
180: (4)                a = np.random.rand(32, 32)
181: (4)                fig_test.add_subplot().matshow(a)
182: (4)                ax_ref = fig_ref.add_subplot()
183: (4)                ax_ref.imshow(a)
184: (4)                ax_ref.xaxis.tick_top()
185: (4)                ax_ref.xaxis.set_ticks_position('both')
186: (0)            @image_comparison(['formatter_ticker_001',
187: (19)                              'formatter_ticker_002',
188: (19)                              'formatter_ticker_003',
```

```
(19)                                'formatter_ticker_004',
(19)                                'formatter_ticker_005',
(19)                                ],
(18)                               tol=0.031 if platform.machine() == 'arm64' else 0)
(0)             def test_formatter_ticker():
(4)                 import matplotlib.testing.jpl_units as units
(4)                 units.register()
(4)                 matplotlib.rcParams['lines.markeredgewidth'] = 30
(4)                 xdata = [x*units.sec for x in range(10)]
(4)                 ydata1 = [(1.5*y - 0.5)*units.km for y in range(10)]
(4)                 ydata2 = [(1.75*y - 1.0)*units.km for y in range(10)]
(4)                 ax = plt.figure().subplots()
(4)                 ax.set_xlabel("x-label 001")
(4)                 ax = plt.figure().subplots()
(4)                 ax.set_xlabel("x-label 001")
(4)                 ax.plot(xdata, ydata1, color='blue', xunits="sec")
(4)                 ax = plt.figure().subplots()
(4)                 ax.set_xlabel("x-label 001")
(4)                 ax.plot(xdata, ydata1, color='blue', xunits="sec")
(4)                 ax.set_xlabel("x-label 003")
(4)                 ax = plt.figure().subplots()
(4)                 ax.plot(xdata, ydata1, color='blue', xunits="sec")
(4)                 ax.plot(xdata, ydata2, color='green', xunits="hour")
(4)                 ax.set_xlabel("x-label 004")
(4)                 ax = plt.figure().subplots()
(4)                 ax.plot(xdata, ydata1, color='blue', xunits="sec")
(4)                 ax.plot(xdata, ydata2, color='green', xunits="hour")
(4)                 ax.set_xlabel("x-label 005")
(4)                 ax.autoscale_view()
(0)             def test_funcformatter_auto_formatter():
(4)                 def _formfunc(x, pos):
(8)                     return ''
(4)                 ax = plt.figure().subplots()
(4)                 assert ax.xaxis.isDefault_majfmt
(4)                 assert ax.xaxis.isDefault_minfmt
(4)                 assert ax.yaxis.isDefault_majfmt
(4)                 assert ax.yaxis.isDefault_minfmt
(4)                 ax.xaxis.set_major_formatter(_formfunc)
(4)                 assert not ax.xaxis.isDefault_majfmt
(4)                 assert ax.xaxis.isDefault_minfmt
(4)                 assert ax.yaxis.isDefault_majfmt
(4)                 assert ax.yaxis.isDefault_minfmt
(4)                 targ_funcformatter = mticker.FuncFormatter(_formfunc)
(4)                 assert isinstance(ax.xaxis.get_major_formatter(),
(22)                                  mticker.FuncFormatter)
(4)                 assert ax.xaxis.get_major_formatter().func == targ_funcformatter.func
(0)             def test_strmethodformatter_auto_formatter():
(4)                 formstr = '{x}_{pos}'
(4)                 ax = plt.figure().subplots()
(4)                 assert ax.xaxis.isDefault_majfmt
(4)                 assert ax.xaxis.isDefault_minfmt
(4)                 assert ax.yaxis.isDefault_majfmt
(4)                 assert ax.yaxis.isDefault_minfmt
(4)                 ax.yaxis.set_minor_formatter(formstr)
(4)                 assert ax.xaxis.isDefault_majfmt
(4)                 assert ax.xaxis.isDefault_minfmt
(4)                 assert ax.yaxis.isDefault_majfmt
(4)                 assert not ax.yaxis.isDefault_minfmt
(4)                 targ_strformatter = mticker.StrMethodFormatter(formstr)
(4)                 assert isinstance(ax.yaxis.get_minor_formatter(),
(22)                                  mticker.StrMethodFormatter)
(4)                 assert ax.yaxis.get_minor_formatter().fmt == targ_strformatter.fmt
(0)             @image_comparison(["twin_axis_locators_formatters"])
(0)             def test_twin_axis_locators_formatters():
(4)                 vals = np.linspace(0, 1, num=5, endpoint=True)
(4)                 locs = np.sin(np.pi * vals / 2.0)
(4)                 majl = plt.FixedLocator(locs)
(4)                 minl = plt.FixedLocator([0.1, 0.2, 0.3])
(4)                 fig = plt.figure()
```

```
(4)                 ax1 = fig.add_subplot(1, 1, 1)
(4)                 ax1.plot([0.1, 100], [0, 1])
(4)                 ax1.yaxis.set_major_locator(majl)
(4)                 ax1.yaxis.set_minor_locator(minl)
(4)                 ax1.yaxis.set_major_formatter(plt.FormatStrFormatter('%08.2lf'))
(4)                 ax1.yaxis.set_minor_formatter(plt.FixedFormatter(['tricks', 'mind',
(54)                                                                  'jedi']))
(4)                 ax1.xaxis.set_major_locator(plt.LinearLocator())
(4)                 ax1.xaxis.set_minor_locator(plt.FixedLocator([15, 35, 55, 75]))
(4)                 ax1.xaxis.set_major_formatter(plt.FormatStrFormatter('%05.2lf'))
(4)                 ax1.xaxis.set_minor_formatter(plt.FixedFormatter(['c', '3', 'p', 'o']))
(4)                 ax1.twiny()
(4)                 ax1.twinx()
(0)             def test_twinx_cla():
(4)                 fig, ax = plt.subplots()
(4)                 ax2 = ax.twinx()
(4)                 ax3 = ax2.twiny()
(4)                 plt.draw()
(4)                 assert not ax2.xaxis.get_visible()
(4)                 assert not ax2.patch.get_visible()
(4)                 ax2.cla()
(4)                 ax3.cla()
(4)                 assert not ax2.xaxis.get_visible()
(4)                 assert not ax2.patch.get_visible()
(4)                 assert ax2.yaxis.get_visible()
(4)                 assert ax3.xaxis.get_visible()
(4)                 assert not ax3.patch.get_visible()
(4)                 assert not ax3.yaxis.get_visible()
(4)                 assert ax.xaxis.get_visible()
(4)                 assert ax.patch.get_visible()
(4)                 assert ax.yaxis.get_visible()
(0)             @pytest.mark.parametrize('twin', ('x', 'y'))
(0)             def test_twin_units(twin):
(4)                 axis_name = f'{twin}axis'
(4)                 twin_func = f'twin{twin}'
(4)                 a = ['0', '1']
(4)                 b = ['a', 'b']
(4)                 fig = Figure()
(4)                 ax1 = fig.subplots()
(4)                 ax1.plot(a, b)
(4)                 assert getattr(ax1, axis_name).units is not None
(4)                 ax2 = getattr(ax1, twin_func)()
(4)                 assert getattr(ax2, axis_name).units is not None
(4)                 assert getattr(ax2, axis_name).units is getattr(ax1, axis_name).units
(0)             @pytest.mark.parametrize('twin', ('x', 'y'))
(0)             @check_figures_equal(extensions=['png'], tol=0.19)
(0)             def test_twin_logscale(fig_test, fig_ref, twin):
(4)                 twin_func = f'twin{twin}'  # test twinx or twiny
(4)                 set_scale = f'set_{twin}scale'
(4)                 x = np.arange(1, 100)
(4)                 ax_test = fig_test.add_subplot(2, 1, 1)
(4)                 ax_twin = getattr(ax_test, twin_func)()
(4)                 getattr(ax_test, set_scale)('log')
(4)                 ax_twin.plot(x, x)
(4)                 ax_test = fig_test.add_subplot(2, 1, 2)
(4)                 getattr(ax_test, set_scale)('log')
(4)                 ax_twin = getattr(ax_test, twin_func)()
(4)                 ax_twin.plot(x, x)
(4)                 for i in [1, 2]:
(8)                     ax_ref = fig_ref.add_subplot(2, 1, i)
(8)                     getattr(ax_ref, set_scale)('log')
(8)                     ax_ref.plot(x, x)
(8)                     Path = matplotlib.path.Path
(8)                     fig_ref.add_artist(
(12)                        matplotlib.patches.PathPatch(
(16)                            Path([[0, 0], [0, 1],
(22)                                  [0, 1], [1, 1],
(22)                                  [1, 1], [1, 0],
(22)                                  [1, 0], [0, 0]],
```

```
(21)                                [Path.MOVETO, Path.LINETO] * 4),
(16)                            transform=ax_ref.transAxes,
(16)                            facecolor='none',
(16)                            edgecolor=mpl.rcParams['axes.edgecolor'],
(16)                            linewidth=mpl.rcParams['axes.linewidth'],
(16)                            capstyle='projecting'))
(4)                 remove_ticks_and_titles(fig_test)
(4)                 remove_ticks_and_titles(fig_ref)
(0)             @image_comparison(['twin_autoscale.png'],
(18)                              tol=0.009 if platform.machine() == 'arm64' else 0)
(0)             def test_twinx_axis_scales():
(4)                 x = np.array([0, 0.5, 1])
(4)                 y = 0.5 * x
(4)                 x2 = np.array([0, 1, 2])
(4)                 y2 = 2 * x2
(4)                 fig = plt.figure()
(4)                 ax = fig.add_axes((0, 0, 1, 1), autoscalex_on=False, autoscaley_on=False)
(4)                 ax.plot(x, y, color='blue', lw=10)
(4)                 ax2 = plt.twinx(ax)
(4)                 ax2.plot(x2, y2, 'r--', lw=5)
(4)                 ax.margins(0, 0)
(4)                 ax2.margins(0, 0)
(0)             def test_twin_inherit_autoscale_setting():
(4)                 fig, ax = plt.subplots()
(4)                 ax_x_on = ax.twinx()
(4)                 ax.set_autoscalex_on(False)
(4)                 ax_x_off = ax.twinx()
(4)                 assert ax_x_on.get_autoscalex_on()
(4)                 assert not ax_x_off.get_autoscalex_on()
(4)                 ax_y_on = ax.twiny()
(4)                 ax.set_autoscaley_on(False)
(4)                 ax_y_off = ax.twiny()
(4)                 assert ax_y_on.get_autoscaley_on()
(4)                 assert not ax_y_off.get_autoscaley_on()
(0)             def test_inverted_cla():
(4)                 fig = plt.figure(0)
(4)                 ax = fig.gca()
(4)                 assert not ax.xaxis_inverted()
(4)                 assert not ax.yaxis_inverted()
(4)                 img = np.random.random((100, 100))
(4)                 ax.imshow(img)
(4)                 assert not ax.xaxis_inverted()
(4)                 assert ax.yaxis_inverted()
(4)                 ax.cla()
(4)                 x = np.linspace(0, 2*np.pi, 100)
(4)                 ax.plot(x, np.cos(x))
(4)                 assert not ax.xaxis_inverted()
(4)                 assert not ax.yaxis_inverted()
(4)                 ax.cla()
(4)                 ax.imshow(img)
(4)                 plt.autoscale()
(4)                 assert not ax.xaxis_inverted()
(4)                 assert ax.yaxis_inverted()
(4)                 for ax in fig.axes:
(8)                     ax.remove()
(4)                 ax0 = plt.subplot(211)
(4)                 ax1 = plt.subplot(212, sharey=ax0)
(4)                 ax0.yaxis.set_inverted(True)
(4)                 assert ax1.yaxis_inverted()
(4)                 ax1.plot(x, np.cos(x))
(4)                 ax0.cla()
(4)                 assert not ax1.yaxis_inverted()
(4)                 ax1.cla()
(4)                 ax0.imshow(img)
(4)                 ax1.plot(x, np.cos(x))
(4)                 ax1.cla()
(4)                 assert ax.yaxis_inverted()
(4)                 plt.close(fig)
(0)             def test_subclass_clear_cla():
```

```
(4)                 with pytest.warns(PendingDeprecationWarning,
(22)                                  match='Overriding `Axes.cla`'):
(8)                     class ClaAxes(Axes):
(12)                        def cla(self):
(16)                            nonlocal called
(16)                            called = True
(4)                 with pytest.warns(PendingDeprecationWarning,
(22)                                  match='Overriding `Axes.cla`'):
(8)                     class ClaSuperAxes(Axes):
(12)                        def cla(self):
(16)                            nonlocal called
(16)                            called = True
(16)                            super().cla()
(4)                 class SubClaAxes(ClaAxes):
(8)                     pass
(4)                 class ClearAxes(Axes):
(8)                     def clear(self):
(12)                        nonlocal called
(12)                        called = True
(4)                 class ClearSuperAxes(Axes):
(8)                     def clear(self):
(12)                        nonlocal called
(12)                        called = True
(12)                        super().clear()
(4)                 class SubClearAxes(ClearAxes):
(8)                     pass
(4)                 fig = Figure()
(4)                 for axes_class in [ClaAxes, ClaSuperAxes, SubClaAxes,
(23)                                   ClearAxes, ClearSuperAxes, SubClearAxes]:
(8)                     called = False
(8)                     ax = axes_class(fig, [0, 0, 1, 1])
(8)                     assert called
(8)                     called = False
(8)                     ax.cla()
(8)                     assert called
(0)             def test_cla_not_redefined_internally():
(4)                 for klass in Axes.__subclasses__():
(8)                     if 'test_subclass_clear_cla' not in klass.__qualname__:
(12)                        assert 'cla' not in klass.__dict__
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_minorticks_on_rcParams_both(fig_test, fig_ref):
(4)                 with matplotlib.rc_context({"xtick.minor.visible": True,
(32)                                            "ytick.minor.visible": True}):
(8)                     ax_test = fig_test.subplots()
(8)                     ax_test.plot([0, 1], [0, 1])
(4)                 ax_ref = fig_ref.subplots()
(4)                 ax_ref.plot([0, 1], [0, 1])
(4)                 ax_ref.minorticks_on()
(0)             @image_comparison(["autoscale_tiny_range"], remove_text=True)
(0)             def test_autoscale_tiny_range():
(4)                 fig, axs = plt.subplots(2, 2)
(4)                 for i, ax in enumerate(axs.flat):
(8)                     y1 = 10**(-11 - i)
(8)                     ax.plot([0, 1], [1, 1 + y1])
(0)             @mpl.style.context('default')
(0)             def test_autoscale_tight():
(4)                 fig, ax = plt.subplots(1, 1)
(4)                 ax.plot([1, 2, 3, 4])
(4)                 ax.autoscale(enable=True, axis='x', tight=False)
(4)                 ax.autoscale(enable=True, axis='y', tight=True)
(4)                 assert_allclose(ax.get_xlim(), (-0.15, 3.15))
(4)                 assert_allclose(ax.get_ylim(), (1.0, 4.0))
(4)                 assert ax.get_autoscalex_on()
(4)                 assert ax.get_autoscaley_on()
(4)                 assert ax.get_autoscale_on()
(4)                 ax.autoscale(enable=None)
(4)                 assert_allclose(ax.get_xlim(), (-0.15, 3.15))
(4)                 assert_allclose(ax.get_ylim(), (1.0, 4.0))
(4)                 assert ax.get_autoscalex_on()
```

```
465: (4)                assert ax.get_autoscaley_on()
466: (4)                assert ax.get_autoscale_on()
467: (0)            @mpl.style.context('default')
468: (0)            def test_autoscale_log_shared():
469: (4)                x = np.arange(100, dtype=float)
470: (4)                fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
471: (4)                ax1.loglog(x, x)
472: (4)                ax2.semilogx(x, x)
473: (4)                ax1.autoscale(tight=True)
474: (4)                ax2.autoscale(tight=True)
475: (4)                plt.draw()
476: (4)                lims = (x[1], x[-1])
477: (4)                assert_allclose(ax1.get_xlim(), lims)
478: (4)                assert_allclose(ax1.get_ylim(), lims)
479: (4)                assert_allclose(ax2.get_xlim(), lims)
480: (4)                assert_allclose(ax2.get_ylim(), (x[0], x[-1]))
481: (0)            @mpl.style.context('default')
482: (0)            def test_use_sticky_edges():
483: (4)                fig, ax = plt.subplots()
484: (4)                ax.imshow([[0, 1], [2, 3]], origin='lower')
485: (4)                assert_allclose(ax.get_xlim(), (-0.5, 1.5))
486: (4)                assert_allclose(ax.get_ylim(), (-0.5, 1.5))
487: (4)                ax.use_sticky_edges = False
488: (4)                ax.autoscale()
489: (4)                xlim = (-0.5 - 2 * ax._xmargin, 1.5 + 2 * ax._xmargin)
490: (4)                ylim = (-0.5 - 2 * ax._ymargin, 1.5 + 2 * ax._ymargin)
491: (4)                assert_allclose(ax.get_xlim(), xlim)
492: (4)                assert_allclose(ax.get_ylim(), ylim)
493: (4)                ax.use_sticky_edges = True
494: (4)                ax.autoscale()
495: (4)                assert_allclose(ax.get_xlim(), (-0.5, 1.5))
496: (4)                assert_allclose(ax.get_ylim(), (-0.5, 1.5))
497: (0)            @check_figures_equal(extensions=["png"])
498: (0)            def test_sticky_shared_axes(fig_test, fig_ref):
499: (4)                Z = np.arange(15).reshape(3, 5)
500: (4)                ax0 = fig_test.add_subplot(211)
501: (4)                ax1 = fig_test.add_subplot(212, sharex=ax0)
502: (4)                ax1.pcolormesh(Z)
503: (4)                ax0 = fig_ref.add_subplot(212)
504: (4)                ax1 = fig_ref.add_subplot(211, sharex=ax0)
505: (4)                ax0.pcolormesh(Z)
506: (0)            @image_comparison(['sticky_tolerance.png'], remove_text=True, style="mpl20")
507: (0)            def test_sticky_tolerance():
508: (4)                fig, axs = plt.subplots(2, 2)
509: (4)                width = .1
510: (4)                axs.flat[0].bar(x=0, height=width, bottom=20000.6)
511: (4)                axs.flat[0].bar(x=1, height=width, bottom=20000.1)
512: (4)                axs.flat[1].bar(x=0, height=-width, bottom=20000.6)
513: (4)                axs.flat[1].bar(x=1, height=-width, bottom=20000.1)
514: (4)                axs.flat[2].barh(y=0, width=-width, left=-20000.6)
515: (4)                axs.flat[2].barh(y=1, width=-width, left=-20000.1)
516: (4)                axs.flat[3].barh(y=0, width=width, left=-20000.6)
517: (4)                axs.flat[3].barh(y=1, width=width, left=-20000.1)
518: (0)            @image_comparison(['sticky_tolerance_cf.png'], remove_text=True,
style="mpl20")
519: (0)            def test_sticky_tolerance_contourf():
520: (4)                fig, ax = plt.subplots()
521: (4)                x = y = [14496.71, 14496.75]
522: (4)                data = [[0, 1], [2, 3]]
523: (4)                ax.contourf(x, y, data)
524: (0)            def test_nargs_stem():
525: (4)                with pytest.raises(TypeError, match='0 were given'):
526: (8)                    plt.stem()
527: (0)            def test_nargs_legend():
528: (4)                with pytest.raises(TypeError, match='3 were given'):
529: (8)                    ax = plt.subplot()
530: (8)                    ax.legend(['First'], ['Second'], 3)
531: (0)            def test_nargs_pcolorfast():
532: (4)                with pytest.raises(TypeError, match='2 were given'):
```

```
(8)                     ax = plt.subplot()
(8)                     ax.pcolorfast([(0, 1), (0, 2)], [[1, 2, 3], [1, 2, 3]])
(0)             @image_comparison(['offset_points'], remove_text=True)
(0)             def test_basic_annotate():
(4)                 t = np.arange(0.0, 5.0, 0.01)
(4)                 s = np.cos(2.0*np.pi * t)
(4)                 fig = plt.figure()
(4)                 ax = fig.add_subplot(autoscale_on=False, xlim=(-1, 5), ylim=(-3, 5))
(4)                 line, = ax.plot(t, s, lw=3, color='purple')
(4)                 ax.annotate('local max', xy=(3, 1), xycoords='data',
(16)                            xytext=(3, 3), textcoords='offset points')
(0)             @image_comparison(['arrow_simple.png'], remove_text=True)
(0)             def test_arrow_simple():
(4)                 length_includes_head = (True, False)
(4)                 shape = ('full', 'left', 'right')
(4)                 head_starts_at_zero = (True, False)
(4)                 kwargs = product(length_includes_head, shape, head_starts_at_zero)
(4)                 fig, axs = plt.subplots(3, 4)
(4)                 for i, (ax, kwarg) in enumerate(zip(axs.flat, kwargs)):
(8)                     ax.set_xlim(-2, 2)
(8)                     ax.set_ylim(-2, 2)
(8)                     (length_includes_head, shape, head_starts_at_zero) = kwarg
(8)                     theta = 2 * np.pi * i / 12
(8)                     ax.arrow(0, 0, np.sin(theta), np.cos(theta),
(17)                             width=theta/100,
(17)                             length_includes_head=length_includes_head,
(17)                             shape=shape,
(17)                             head_starts_at_zero=head_starts_at_zero,
(17)                             head_width=theta / 10,
(17)                             head_length=theta / 10)
(0)             def test_arrow_empty():
(4)                 _, ax = plt.subplots()
(4)                 ax.arrow(0, 0, 0, 0, head_length=0)
(0)             def test_arrow_in_view():
(4)                 _, ax = plt.subplots()
(4)                 ax.arrow(1, 1, 1, 1)
(4)                 assert ax.get_xlim() == (0.8, 2.2)
(4)                 assert ax.get_ylim() == (0.8, 2.2)
(0)             def test_annotate_default_arrow():
(4)                 fig, ax = plt.subplots()
(4)                 ann = ax.annotate("foo", (0, 1), xytext=(2, 3))
(4)                 assert ann.arrow_patch is None
(4)                 ann = ax.annotate("foo", (0, 1), xytext=(2, 3), arrowprops={})
(4)                 assert ann.arrow_patch is not None
(0)             def test_annotate_signature():
(4)                 """Check that the signature of Axes.annotate() matches Annotation."""
(4)                 fig, ax = plt.subplots()
(4)                 annotate_params = inspect.signature(ax.annotate).parameters
(4)                 annotation_params = inspect.signature(mtext.Annotation).parameters
(4)                 assert list(annotate_params.keys()) == list(annotation_params.keys())
(4)                 for p1, p2 in zip(annotate_params.values(), annotation_params.values()):
(8)                     assert p1 == p2
(0)             @image_comparison(['fill_units.png'], savefig_kwarg={'dpi': 60})
(0)             def test_fill_units():
(4)                 import matplotlib.testing.jpl_units as units
(4)                 units.register()
(4)                 t = units.Epoch("ET", dt=datetime.datetime(2009, 4, 27))
(4)                 value = 10.0 * units.deg
(4)                 day = units.Duration("ET", 24.0 * 60.0 * 60.0)
(4)                 dt = np.arange('2009-04-27', '2009-04-29', dtype='datetime64[D]')
(4)                 dtn = mdates.date2num(dt)
(4)                 fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
(4)                 ax1.plot([t], [value], yunits='deg', color='red')
(4)                 ind = [0, 0, 1, 1]
(4)                 ax1.fill(dtn[ind], [0.0, 0.0, 90.0, 0.0], 'b')
(4)                 ax2.plot([t], [value], yunits='deg', color='red')
(4)                 ax2.fill([t, t, t + day, t + day],
(13)                         [0.0, 0.0, 90.0, 0.0], 'b')
(4)                 ax3.plot([t], [value], yunits='deg', color='red')
```

```
(4)                 ax3.fill(dtn[ind],
(13)                         [0 * units.deg, 0 * units.deg, 90 * units.deg, 0 * units.deg],
(13)                         'b')
(4)                 ax4.plot([t], [value], yunits='deg', color='red')
(4)                 ax4.fill([t, t, t + day, t + day],
(13)                         [0 * units.deg, 0 * units.deg, 90 * units.deg, 0 * units.deg],
(13)                         facecolor="blue")
(4)                 fig.autofmt_xdate()
(0)             def test_plot_format_kwarg_redundant():
(4)                 with pytest.warns(UserWarning, match="marker .* redundantly defined"):
(8)                     plt.plot([0], [0], 'o', marker='x')
(4)                 with pytest.warns(UserWarning, match="linestyle .* redundantly defined"):
(8)                     plt.plot([0], [0], '-', linestyle='--')
(4)                 with pytest.warns(UserWarning, match="color .* redundantly defined"):
(8)                     plt.plot([0], [0], 'r', color='blue')
(4)                 plt.errorbar([0], [0], fmt='none', color='blue')
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_errorbar_dashes(fig_test, fig_ref):
(4)                 x = [1, 2, 3, 4]
(4)                 y = np.sin(x)
(4)                 ax_ref = fig_ref.gca()
(4)                 ax_test = fig_test.gca()
(4)                 line, *_ = ax_ref.errorbar(x, y, xerr=np.abs(y), yerr=np.abs(y))
(4)                 line.set_dashes([2, 2])
(4)                 ax_test.errorbar(x, y, xerr=np.abs(y), yerr=np.abs(y), dashes=[2, 2])
(0)             def test_errorbar_mapview_kwarg():
(4)                 D = {ii: ii for ii in range(10)}
(4)                 fig, ax = plt.subplots()
(4)                 ax.errorbar(x=D.keys(), y=D.values(), xerr=D.values())
(0)             @image_comparison(['single_point', 'single_point'])
(0)             def test_single_point():
(4)                 matplotlib.rcParams['lines.marker'] = 'o'
(4)                 matplotlib.rcParams['axes.grid'] = True
(4)                 fig, (ax1, ax2) = plt.subplots(2)
(4)                 ax1.plot([0], [0], 'o')
(4)                 ax2.plot([1], [1], 'o')
(4)                 data = {'a': [0], 'b': [1]}
(4)                 fig, (ax1, ax2) = plt.subplots(2)
(4)                 ax1.plot('a', 'a', 'o', data=data)
(4)                 ax2.plot('b', 'b', 'o', data=data)
(0)             @image_comparison(['single_date.png'], style='mpl20')
(0)             def test_single_date():
(4)                 plt.rcParams['axes.formatter.limits'] = -7, 7
(4)                 dt = mdates.date2num(np.datetime64('0000-12-31'))
(4)                 time1 = [721964.0]
(4)                 data1 = [-65.54]
(4)                 fig, ax = plt.subplots(2, 1)
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning):
(8)                     ax[0].plot_date(time1 + dt, data1, 'o', color='r')
(4)                 ax[1].plot(time1, data1, 'o', color='r')
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_shaped_data(fig_test, fig_ref):
(4)                 row = np.arange(10).reshape((1, -1))
(4)                 col = np.arange(0, 100, 10).reshape((-1, 1))
(4)                 axs = fig_test.subplots(2)
(4)                 axs[0].plot(row)  # Actually plots nothing (columns are single points).
(4)                 axs[1].plot(col)  # Same as plotting 1d.
(4)                 axs = fig_ref.subplots(2)
(4)                 axs[0].set(xlim=(-.06, .06), ylim=(0, 9))
(4)                 axs[1].plot(col.ravel())
(0)             def test_structured_data():
(4)                 pts = np.array([(1, 1), (2, 2)], dtype=[("ones", float), ("twos", float)])
(4)                 axs = plt.figure().subplots(2)
(4)                 axs[0].plot("ones", "twos", data=pts)
(4)                 axs[1].plot("ones", "twos", "r", data=pts)
(0)             @image_comparison(['aitoff_proj'], extensions=["png"],
(18)                              remove_text=True, style='mpl20')
(0)             def test_aitoff_proj():
(4)                 """
```

```
671: (4)                Test aitoff projection ref.:
672: (4)                https://github.com/matplotlib/matplotlib/pull/14451
673: (4)                """
674: (4)                x = np.linspace(-np.pi, np.pi, 20)
675: (4)                y = np.linspace(-np.pi / 2, np.pi / 2, 20)
676: (4)                X, Y = np.meshgrid(x, y)
677: (4)                fig, ax = plt.subplots(figsize=(8, 4.2),
678: (27)                                       subplot_kw=dict(projection="aitoff"))
679: (4)                ax.grid()
680: (4)                ax.plot(X.flat, Y.flat, 'o', markersize=4)
681: (0)            @image_comparison(['axvspan_epoch'])
682: (0)            def test_axvspan_epoch():
683: (4)                import matplotlib.testing.jpl_units as units
684: (4)                units.register()
685: (4)                t0 = units.Epoch("ET", dt=datetime.datetime(2009, 1, 21))
686: (4)                tf = units.Epoch("ET", dt=datetime.datetime(2009, 1, 22))
687: (4)                dt = units.Duration("ET", units.day.convert("sec"))
688: (4)                ax = plt.gca()
689: (4)                ax.axvspan(t0, tf, facecolor="blue", alpha=0.25)
690: (4)                ax.set_xlim(t0 - 5.0*dt, tf + 5.0*dt)
691: (0)            @image_comparison(['axhspan_epoch'], tol=0.02)
692: (0)            def test_axhspan_epoch():
693: (4)                import matplotlib.testing.jpl_units as units
694: (4)                units.register()
695: (4)                t0 = units.Epoch("ET", dt=datetime.datetime(2009, 1, 21))
696: (4)                tf = units.Epoch("ET", dt=datetime.datetime(2009, 1, 22))
697: (4)                dt = units.Duration("ET", units.day.convert("sec"))
698: (4)                ax = plt.gca()
699: (4)                ax.axhspan(t0, tf, facecolor="blue", alpha=0.25)
700: (4)                ax.set_ylim(t0 - 5.0*dt, tf + 5.0*dt)
701: (0)            @image_comparison(['hexbin_extent.png', 'hexbin_extent.png'],
remove_text=True)
702: (0)            def test_hexbin_extent():
703: (4)                fig, ax = plt.subplots()
704: (4)                data = (np.arange(2000) / 2000).reshape((2, 1000))
705: (4)                x, y = data
706: (4)                ax.hexbin(x, y, extent=[.1, .3, .6, .7])
707: (4)                data = {"x": x, "y": y}
708: (4)                fig, ax = plt.subplots()
709: (4)                ax.hexbin("x", "y", extent=[.1, .3, .6, .7], data=data)
710: (0)            def test_hexbin_bad_extents():
711: (4)                fig, ax = plt.subplots()
712: (4)                data = (np.arange(20) / 20).reshape((2, 10))
713: (4)                x, y = data
714: (4)                with pytest.raises(ValueError, match="In extent, xmax must be greater than
xmin"):
715: (8)                    ax.hexbin(x, y, extent=(1, 0, 0, 1))
716: (4)                with pytest.raises(ValueError, match="In extent, ymax must be greater than
ymin"):
717: (8)                    ax.hexbin(x, y, extent=(0, 1, 1, 0))
718: (0)            @image_comparison(['hexbin_empty.png'], remove_text=True)
719: (0)            def test_hexbin_empty():
720: (4)                fig, ax = plt.subplots()
721: (4)                ax.hexbin([], [])
722: (4)                ax.hexbin([], [], bins='log')
723: (4)                ax.hexbin([], [], C=[], reduce_C_function=np.max)
724: (4)                ax.hexbin([], [], bins=np.arange(10))
725: (0)            def test_hexbin_pickable():
726: (4)                fig, ax = plt.subplots()
727: (4)                data = (np.arange(200) / 200).reshape((2, 100))
728: (4)                x, y = data
729: (4)                hb = ax.hexbin(x, y, extent=[.1, .3, .6, .7], picker=-1)
730: (4)                mouse_event = SimpleNamespace(x=400, y=300)
731: (4)                assert hb.contains(mouse_event)[0]
732: (0)            @image_comparison(['hexbin_log.png'], style='mpl20')
733: (0)            def test_hexbin_log():
734: (4)                plt.rcParams['pcolormesh.snap'] = False
735: (4)                np.random.seed(19680801)
736: (4)                n = 100000
```

```
(4)                 x = np.random.standard_normal(n)
(4)                 y = 2.0 + 3.0 * x + 4.0 * np.random.standard_normal(n)
(4)                 y = np.power(2, y * 0.5)
(4)                 fig, ax = plt.subplots()
(4)                 h = ax.hexbin(x, y, yscale='log', bins='log',
(18)                              marginals=True, reduce_C_function=np.sum)
(4)                 plt.colorbar(h)
(0)             @image_comparison(["hexbin_linear.png"], style="mpl20", remove_text=True)
(0)             def test_hexbin_linear():
(4)                 np.random.seed(19680801)
(4)                 n = 100000
(4)                 x = np.random.standard_normal(n)
(4)                 y = 2.0 + 3.0 * x + 4.0 * np.random.standard_normal(n)
(4)                 fig, ax = plt.subplots()
(4)                 ax.hexbin(x, y, gridsize=(10, 5), marginals=True,
(14)                          reduce_C_function=np.sum)
(0)             def test_hexbin_log_clim():
(4)                 x, y = np.arange(200).reshape((2, 100))
(4)                 fig, ax = plt.subplots()
(4)                 h = ax.hexbin(x, y, bins='log', vmin=2, vmax=100)
(4)                 assert h.get_clim() == (2, 100)
(0)             @check_figures_equal(extensions=['png'])
(0)             def test_hexbin_mincnt_behavior_upon_C_parameter(fig_test, fig_ref):
(4)                 datapoints = [
(8)                     (0, 0),
(8)                     (0, 0),
(8)                     (6, 0),
(8)                     (0, 6),
(4)                 ]
(4)                 X, Y = zip(*datapoints)
(4)                 C = [1] * len(X)
(4)                 extent = [-10., 10, -10., 10]
(4)                 gridsize = (7, 7)
(4)                 ax_test = fig_test.subplots()
(4)                 ax_ref = fig_ref.subplots()
(4)                 ax_ref.hexbin(
(8)                     X, Y,
(8)                     extent=extent,
(8)                     gridsize=gridsize,
(8)                     mincnt=1,
(4)                 )
(4)                 ax_ref.set_facecolor("green")  # for contrast of background
(4)                 ax_test.hexbin(
(8)                     X, Y,
(8)                     C=[1] * len(X),
(8)                     reduce_C_function=lambda v: sum(v),
(8)                     mincnt=1,
(8)                     extent=extent,
(8)                     gridsize=gridsize,
(4)                 )
(4)                 ax_test.set_facecolor("green")
(0)             def test_inverted_limits():
(4)                 fig, ax = plt.subplots()
(4)                 ax.invert_xaxis()
(4)                 ax.plot([-5, -3, 2, 4], [1, 2, -3, 5])
(4)                 assert ax.get_xlim() == (4, -5)
(4)                 assert ax.get_ylim() == (-3, 5)
(4)                 plt.close()
(4)                 fig, ax = plt.subplots()
(4)                 ax.invert_yaxis()
(4)                 ax.plot([-5, -3, 2, 4], [1, 2, -3, 5])
(4)                 assert ax.get_xlim() == (-5, 4)
(4)                 assert ax.get_ylim() == (5, -3)
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_yscale("log")
(4)                 ax.set_ylim(10, 1)
(4)                 assert ax.get_ylim() == (10, 1)
(0)             @image_comparison(['nonfinite_limits'])
(0)             def test_nonfinite_limits():
```

```
806: (4)                x = np.arange(0., np.e, 0.01)
807: (4)                with np.errstate(divide='ignore'):
808: (8)                    y = np.log(x)
809: (4)                x[len(x)//2] = np.nan
810: (4)                fig, ax = plt.subplots()
811: (4)                ax.plot(x, y)
812: (0)            @mpl.style.context('default')
813: (0)            @pytest.mark.parametrize('plot_fun',
814: (25)                                   ['scatter', 'plot', 'fill_between'])
815: (0)            @check_figures_equal(extensions=["png"])
816: (0)            def test_limits_empty_data(plot_fun, fig_test, fig_ref):
817: (4)                x = np.arange("2010-01-01", "2011-01-01", dtype="datetime64[D]")
818: (4)                ax_test = fig_test.subplots()
819: (4)                ax_ref = fig_ref.subplots()
820: (4)                getattr(ax_test, plot_fun)([], [])
821: (4)                for ax in [ax_test, ax_ref]:
822: (8)                    getattr(ax, plot_fun)(x, range(len(x)), color='C0')
823: (0)            @image_comparison(['imshow', 'imshow'], remove_text=True, style='mpl20')
824: (0)            def test_imshow():
825: (4)                matplotlib.rcParams['image.interpolation'] = 'nearest'
826: (4)                N = 100
827: (4)                (x, y) = np.indices((N, N))
828: (4)                x -= N//2
829: (4)                y -= N//2
830: (4)                r = np.sqrt(x**2+y**2-x*y)
831: (4)                fig, ax = plt.subplots()
832: (4)                ax.imshow(r)
833: (4)                data = {"r": r}
834: (4)                fig, ax = plt.subplots()
835: (4)                ax.imshow("r", data=data)
836: (0)            @image_comparison(
837: (4)                ['imshow_clip'], style='mpl20',
838: (4)                tol=1.24 if platform.machine() in ('aarch64', 'ppc64le', 's390x') else 0)
839: (0)            def test_imshow_clip():
840: (4)                matplotlib.rcParams['image.interpolation'] = 'nearest'
841: (4)                N = 100
842: (4)                (x, y) = np.indices((N, N))
843: (4)                x -= N//2
844: (4)                y -= N//2
845: (4)                r = np.sqrt(x**2+y**2-x*y)
846: (4)                fig, ax = plt.subplots()
847: (4)                c = ax.contour(r, [N/4])
848: (4)                clip_path = mtransforms.TransformedPath(c.get_paths()[0],
c.get_transform())
849: (4)                ax.imshow(r, clip_path=clip_path)
850: (0)            def test_imshow_norm_vminvmax():
851: (4)                """Parameters vmin, vmax should error if norm is given."""
852: (4)                a = [[1, 2], [3, 4]]
853: (4)                ax = plt.axes()
854: (4)                with pytest.raises(ValueError,
855: (23)                                   match="Passing a Normalize instance simultaneously "
856: (29)                                         "with vmin/vmax is not supported."):
857: (8)                    ax.imshow(a, norm=mcolors.Normalize(-10, 10), vmin=0, vmax=5)
858: (0)            @image_comparison(['polycollection_joinstyle'], remove_text=True)
859: (0)            def test_polycollection_joinstyle():
860: (4)                fig, ax = plt.subplots()
861: (4)                verts = np.array([[1, 1], [1, 2], [2, 2], [2, 1]])
862: (4)                c = mpl.collections.PolyCollection([verts], linewidths=40)
863: (4)                ax.add_collection(c)
864: (4)                ax.set_xbound(0, 3)
865: (4)                ax.set_ybound(0, 3)
866: (0)            @pytest.mark.parametrize(
867: (4)                'x, y1, y2', [
868: (8)                    (np.zeros((2, 2)), 3, 3),
869: (8)                    (np.arange(0.0, 2, 0.02), np.zeros((2, 2)), 3),
870: (8)                    (np.arange(0.0, 2, 0.02), 3, np.zeros((2, 2)))
871: (4)                ], ids=[
872: (8)                    '2d_x_input',
873: (8)                    '2d_y1_input',
```

```
(8)                     '2d_y2_input'
(4)                 ]
(0)             )
(0)             def test_fill_between_input(x, y1, y2):
(4)                 fig, ax = plt.subplots()
(4)                 with pytest.raises(ValueError):
(8)                     ax.fill_between(x, y1, y2)
(0)             @pytest.mark.parametrize(
(4)                 'y, x1, x2', [
(8)                     (np.zeros((2, 2)), 3, 3),
(8)                     (np.arange(0.0, 2, 0.02), np.zeros((2, 2)), 3),
(8)                     (np.arange(0.0, 2, 0.02), 3, np.zeros((2, 2)))
(4)                 ], ids=[
(8)                     '2d_y_input',
(8)                     '2d_x1_input',
(8)                     '2d_x2_input'
(4)                 ]
(0)             )
(0)             def test_fill_betweenx_input(y, x1, x2):
(4)                 fig, ax = plt.subplots()
(4)                 with pytest.raises(ValueError):
(8)                     ax.fill_betweenx(y, x1, x2)
(0)             @image_comparison(['fill_between_interpolate'], remove_text=True,
(18)                            tol=0.012 if platform.machine() == 'arm64' else 0)
(0)             def test_fill_between_interpolate():
(4)                 x = np.arange(0.0, 2, 0.02)
(4)                 y1 = np.sin(2*np.pi*x)
(4)                 y2 = 1.2*np.sin(4*np.pi*x)
(4)                 fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
(4)                 ax1.plot(x, y1, x, y2, color='black')
(4)                 ax1.fill_between(x, y1, y2, where=y2 >= y1, facecolor='white', hatch='/',
(21)                                 interpolate=True)
(4)                 ax1.fill_between(x, y1, y2, where=y2 <= y1, facecolor='red',
(21)                                 interpolate=True)
(4)                 y2 = np.ma.masked_greater(y2, 1.0)
(4)                 y2[0] = np.ma.masked
(4)                 ax2.plot(x, y1, x, y2, color='black')
(4)                 ax2.fill_between(x, y1, y2, where=y2 >= y1, facecolor='green',
(21)                                 interpolate=True)
(4)                 ax2.fill_between(x, y1, y2, where=y2 <= y1, facecolor='red',
(21)                                 interpolate=True)
(0)             @image_comparison(['fill_between_interpolate_decreasing'],
(18)                            style='mpl20', remove_text=True)
(0)             def test_fill_between_interpolate_decreasing():
(4)                 p = np.array([724.3, 700, 655])
(4)                 t = np.array([9.4, 7, 2.2])
(4)                 prof = np.array([7.9, 6.6, 3.8])
(4)                 fig, ax = plt.subplots(figsize=(9, 9))
(4)                 ax.plot(t, p, 'tab:red')
(4)                 ax.plot(prof, p, 'k')
(4)                 ax.fill_betweenx(p, t, prof, where=prof < t,
(21)                                 facecolor='blue', interpolate=True, alpha=0.4)
(4)                 ax.fill_betweenx(p, t, prof, where=prof > t,
(21)                                 facecolor='red', interpolate=True, alpha=0.4)
(4)                 ax.set_xlim(0, 30)
(4)                 ax.set_ylim(800, 600)
(0)             @image_comparison(['fill_between_interpolate_nan'], remove_text=True)
(0)             def test_fill_between_interpolate_nan():
(4)                 x = np.arange(10)
(4)                 y1 = np.asarray([8, 18, np.nan, 18, 8, 18, 24, 18, 8, 18])
(4)                 y2 = np.asarray([18, 11, 8, 11, 18, 26, 32, 30, np.nan, np.nan])
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(x, y1, c='k')
(4)                 ax.plot(x, y2, c='b')
(4)                 ax.fill_between(x, y1, y2, where=y2 >= y1, facecolor="green",
(20)                                interpolate=True, alpha=0.5)
(4)                 ax.fill_between(x, y1, y2, where=y1 >= y2, facecolor="red",
(20)                                interpolate=True, alpha=0.5)
(0)             @image_comparison(['symlog.pdf'])
```

```
(0)             def test_symlog():
(4)                 x = np.array([0, 1, 2, 4, 6, 9, 12, 24])
(4)                 y = np.array([1000000, 500000, 100000, 100, 5, 0, 0, 0])
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(x, y)
(4)                 ax.set_yscale('symlog')
(4)                 ax.set_xscale('linear')
(4)                 ax.set_ylim(-1, 10000000)
(0)             @image_comparison(['symlog2.pdf'], remove_text=True)
(0)             def test_symlog2():
(4)                 x = np.arange(-50, 50, 0.001)
(4)                 fig, axs = plt.subplots(5, 1)
(4)                 for ax, linthresh in zip(axs, [20., 2., 1., 0.1, 0.01]):
(8)                     ax.plot(x, x)
(8)                     ax.set_xscale('symlog', linthresh=linthresh)
(8)                     ax.grid(True)
(4)                 axs[-1].set_ylim(-0.1, 0.1)
(0)             def test_pcolorargs_5205():
(4)                 x = [-1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5]
(4)                 y = [-1.5, -1.25, -1.0, -0.75, -0.5, -0.25, 0,
(9)                      0.25, 0.5, 0.75, 1.0, 1.25, 1.5]
(4)                 X, Y = np.meshgrid(x, y)
(4)                 Z = np.hypot(X, Y)
(4)                 plt.pcolor(Z)
(4)                 plt.pcolor(list(Z))
(4)                 plt.pcolor(x, y, Z[:-1, :-1])
(4)                 plt.pcolor(X, Y, list(Z[:-1, :-1]))
(0)             @image_comparison(['pcolormesh'], remove_text=True)
(0)             def test_pcolormesh():
(4)                 plt.rcParams['pcolormesh.snap'] = False
(4)                 n = 12
(4)                 x = np.linspace(-1.5, 1.5, n)
(4)                 y = np.linspace(-1.5, 1.5, n*2)
(4)                 X, Y = np.meshgrid(x, y)
(4)                 Qx = np.cos(Y) - np.cos(X)
(4)                 Qz = np.sin(Y) + np.sin(X)
(4)                 Qx = (Qx + 1.1)
(4)                 Z = np.hypot(X, Y) / 5
(4)                 Z = (Z - Z.min()) / np.ptp(Z)
(4)                 Zm = ma.masked_where(np.abs(Qz) < 0.5 * np.max(Qz), Z)
(4)                 _, (ax1, ax2, ax3) = plt.subplots(1, 3)
(4)                 ax1.pcolormesh(Qx, Qz, Zm[:-1, :-1], lw=0.5, edgecolors='k')
(4)                 ax2.pcolormesh(Qx, Qz, Zm[:-1, :-1], lw=2, edgecolors=['b', 'w'])
(4)                 ax3.pcolormesh(Qx, Qz, Zm, shading="gouraud")
(0)             @image_comparison(['pcolormesh_small'], extensions=["eps"])
(0)             def test_pcolormesh_small():
(4)                 n = 3
(4)                 x = np.linspace(-1.5, 1.5, n)
(4)                 y = np.linspace(-1.5, 1.5, n*2)
(4)                 X, Y = np.meshgrid(x, y)
(4)                 Qx = np.cos(Y) - np.cos(X)
(4)                 Qz = np.sin(Y) + np.sin(X)
(4)                 Qx = (Qx + 1.1)
(4)                 Z = np.hypot(X, Y) / 5
(4)                 Z = (Z - Z.min()) / np.ptp(Z)
(4)                 Zm = ma.masked_where(np.abs(Qz) < 0.5 * np.max(Qz), Z)
(4)                 Zm2 = ma.masked_where(Qz < -0.5 * np.max(Qz), Z)
(4)                 fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
(4)                 ax1.pcolormesh(Qx, Qz, Zm[:-1, :-1], lw=0.5, edgecolors='k')
(4)                 ax2.pcolormesh(Qx, Qz, Zm[:-1, :-1], lw=2, edgecolors=['b', 'w'])
(4)                 ax3.pcolormesh(Qx, Qz, Zm, shading="gouraud")
(4)                 ax4.pcolormesh(Qx, Qz, Zm2, shading="gouraud")
(4)                 for ax in fig.axes:
(8)                     ax.set_axis_off()
(0)             @image_comparison(['pcolormesh_alpha'], extensions=["png", "pdf"],
(18)                              remove_text=True)
(0)             def test_pcolormesh_alpha():
(4)                 plt.rcParams['pcolormesh.snap'] = False
(4)                 n = 12
```

```
(4)                 X, Y = np.meshgrid(
(8)                     np.linspace(-1.5, 1.5, n),
(8)                     np.linspace(-1.5, 1.5, n*2)
(4)                 )
(4)                 Qx = X
(4)                 Qy = Y + np.sin(X)
(4)                 Z = np.hypot(X, Y) / 5
(4)                 Z = (Z - Z.min()) / np.ptp(Z)
(4)                 vir = mpl.colormaps["viridis"].resampled(16)
(4)                 colors = vir(np.arange(16))
(4)                 colors[:, 3] = 0.5 + 0.5*np.sin(np.arange(16))
(4)                 cmap = mcolors.ListedColormap(colors)
(4)                 fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
(4)                 for ax in ax1, ax2, ax3, ax4:
(8)                     ax.add_patch(mpatches.Rectangle(
(12)                        (0, -1.5), 1.5, 3, facecolor=[.7, .1, .1, .5], zorder=0
(8)                     ))
(4)                 ax1.pcolormesh(Qx, Qy, Z[:-1, :-1], cmap=vir, alpha=0.4,
(19)                               shading='flat', zorder=1)
(4)                 ax2.pcolormesh(Qx, Qy, Z, cmap=vir, alpha=0.4, shading='gouraud',
zorder=1)
(4)                 ax3.pcolormesh(Qx, Qy, Z[:-1, :-1], cmap=cmap, shading='flat', zorder=1)
(4)                 ax4.pcolormesh(Qx, Qy, Z, cmap=cmap, shading='gouraud', zorder=1)
(0)             @pytest.mark.parametrize("dims,alpha", [(3, 1), (4, 0.5)])
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_pcolormesh_rgba(fig_test, fig_ref, dims, alpha):
(4)                 ax = fig_test.subplots()
(4)                 c = np.ones((5, 6, dims), dtype=float) / 2
(4)                 ax.pcolormesh(c)
(4)                 ax = fig_ref.subplots()
(4)                 ax.pcolormesh(c[..., 0], cmap="gray", vmin=0, vmax=1, alpha=alpha)
(0)             @image_comparison(['pcolormesh_datetime_axis.png'], style='mpl20')
(0)             def test_pcolormesh_datetime_axis():
(4)                 plt.rcParams['pcolormesh.snap'] = False
(4)                 fig = plt.figure()
(4)                 fig.subplots_adjust(hspace=0.4, top=0.98, bottom=.15)
(4)                 base = datetime.datetime(2013, 1, 1)
(4)                 x = np.array([base + datetime.timedelta(days=d) for d in range(21)])
(4)                 y = np.arange(21)
(4)                 z1, z2 = np.meshgrid(np.arange(20), np.arange(20))
(4)                 z = z1 * z2
(4)                 plt.subplot(221)
(4)                 plt.pcolormesh(x[:-1], y[:-1], z[:-1, :-1])
(4)                 plt.subplot(222)
(4)                 plt.pcolormesh(x, y, z)
(4)                 x = np.repeat(x[np.newaxis], 21, axis=0)
(4)                 y = np.repeat(y[:, np.newaxis], 21, axis=1)
(4)                 plt.subplot(223)
(4)                 plt.pcolormesh(x[:-1, :-1], y[:-1, :-1], z[:-1, :-1])
(4)                 plt.subplot(224)
(4)                 plt.pcolormesh(x, y, z)
(4)                 for ax in fig.get_axes():
(8)                     for label in ax.get_xticklabels():
(12)                        label.set_ha('right')
(12)                        label.set_rotation(30)
(0)             @image_comparison(['pcolor_datetime_axis.png'], style='mpl20')
(0)             def test_pcolor_datetime_axis():
(4)                 fig = plt.figure()
(4)                 fig.subplots_adjust(hspace=0.4, top=0.98, bottom=.15)
(4)                 base = datetime.datetime(2013, 1, 1)
(4)                 x = np.array([base + datetime.timedelta(days=d) for d in range(21)])
(4)                 y = np.arange(21)
(4)                 z1, z2 = np.meshgrid(np.arange(20), np.arange(20))
(4)                 z = z1 * z2
(4)                 plt.subplot(221)
(4)                 plt.pcolor(x[:-1], y[:-1], z[:-1, :-1])
(4)                 plt.subplot(222)
(4)                 plt.pcolor(x, y, z)
(4)                 x = np.repeat(x[np.newaxis], 21, axis=0)
```

```
(4)             y = np.repeat(y[:, np.newaxis], 21, axis=1)
(4)             plt.subplot(223)
(4)             plt.pcolor(x[:-1, :-1], y[:-1, :-1], z[:-1, :-1])
(4)             plt.subplot(224)
(4)             plt.pcolor(x, y, z)
(4)             for ax in fig.get_axes():
(8)                 for label in ax.get_xticklabels():
(12)                    label.set_ha('right')
(12)                    label.set_rotation(30)
(0)         def test_pcolorargs():
(4)             n = 12
(4)             x = np.linspace(-1.5, 1.5, n)
(4)             y = np.linspace(-1.5, 1.5, n*2)
(4)             X, Y = np.meshgrid(x, y)
(4)             Z = np.hypot(X, Y) / 5
(4)             _, ax = plt.subplots()
(4)             with pytest.raises(TypeError):
(8)                 ax.pcolormesh(y, x, Z)
(4)             with pytest.raises(TypeError):
(8)                 ax.pcolormesh(X, Y, Z.T)
(4)             with pytest.raises(TypeError):
(8)                 ax.pcolormesh(x, y, Z[:-1, :-1], shading="gouraud")
(4)             with pytest.raises(TypeError):
(8)                 ax.pcolormesh(X, Y, Z[:-1, :-1], shading="gouraud")
(4)             x[0] = np.nan
(4)             with pytest.raises(ValueError):
(8)                 ax.pcolormesh(x, y, Z[:-1, :-1])
(4)             with np.errstate(invalid='ignore'):
(8)                 x = np.ma.array(x, mask=(x < 0))
(4)             with pytest.raises(ValueError):
(8)                 ax.pcolormesh(x, y, Z[:-1, :-1])
(4)             x = [359, 0, 1]
(4)             y = [-10, 10]
(4)             X, Y = np.meshgrid(x, y)
(4)             Z = np.zeros(X.shape)
(4)             with pytest.warns(UserWarning,
(22)                              match='are not monotonically increasing or decreasing'):
(8)                 ax.pcolormesh(X, Y, Z, shading='auto')
(0)         def test_pcolormesh_underflow_error():
(4)             """
(4)             Test that underflow errors don't crop up in pcolormesh.  Probably
(4)             a numpy bug (https://github.com/numpy/numpy/issues/25810).
(4)             """
(4)             with np.errstate(under="raise"):
(8)                 x = np.arange(0, 3, 0.1)
(8)                 y = np.arange(0, 6, 0.1)
(8)                 z = np.random.randn(len(y), len(x))
(8)                 fig, ax = plt.subplots()
(8)                 ax.pcolormesh(x, y, z)
(0)         def test_pcolorargs_with_read_only():
(4)             x = np.arange(6).reshape(2, 3)
(4)             xmask = np.broadcast_to([False, True, False], x.shape)  # read-only array
(4)             assert xmask.flags.writeable is False
(4)             masked_x = np.ma.array(x, mask=xmask)
(4)             plt.pcolormesh(masked_x)
(4)             x = np.linspace(0, 1, 10)
(4)             y = np.linspace(0, 1, 10)
(4)             X, Y = np.meshgrid(x, y)
(4)             Z = np.sin(2 * np.pi * X) * np.cos(2 * np.pi * Y)
(4)             mask = np.zeros(10, dtype=bool)
(4)             mask[-1] = True
(4)             mask = np.broadcast_to(mask, Z.shape)
(4)             assert mask.flags.writeable is False
(4)             masked_Z = np.ma.array(Z, mask=mask)
(4)             plt.pcolormesh(X, Y, masked_Z)
(4)             masked_X = np.ma.array(X, mask=mask)
(4)             masked_Y = np.ma.array(Y, mask=mask)
(4)             plt.pcolor(masked_X, masked_Y, masked_Z)
(0)         @check_figures_equal(extensions=["png"])
```

```
(0)  def test_pcolornearest(fig_test, fig_ref):
(4)      ax = fig_test.subplots()
(4)      x = np.arange(0, 10)
(4)      y = np.arange(0, 3)
(4)      np.random.seed(19680801)
(4)      Z = np.random.randn(2, 9)
(4)      ax.pcolormesh(x, y, Z, shading='flat')
(4)      ax = fig_ref.subplots()
(4)      x2 = x[:-1] + np.diff(x) / 2
(4)      y2 = y[:-1] + np.diff(y) / 2
(4)      ax.pcolormesh(x2, y2, Z, shading='nearest')
(0)  @check_figures_equal(extensions=["png"])
(0)  def test_pcolornearestunits(fig_test, fig_ref):
(4)      ax = fig_test.subplots()
(4)      x = [datetime.datetime.fromtimestamp(x * 3600) for x in range(10)]
(4)      y = np.arange(0, 3)
(4)      np.random.seed(19680801)
(4)      Z = np.random.randn(2, 9)
(4)      ax.pcolormesh(x, y, Z, shading='flat')
(4)      ax = fig_ref.subplots()
(4)      x2 = [datetime.datetime.fromtimestamp((x + 0.5) * 3600) for x in range(9)]
(4)      y2 = y[:-1] + np.diff(y) / 2
(4)      ax.pcolormesh(x2, y2, Z, shading='nearest')
(0)  def test_pcolorflaterror():
(4)      fig, ax = plt.subplots()
(4)      x = np.arange(0, 9)
(4)      y = np.arange(0, 3)
(4)      np.random.seed(19680801)
(4)      Z = np.random.randn(3, 9)
(4)      with pytest.raises(TypeError, match='Dimensions of C'):
(8)          ax.pcolormesh(x, y, Z, shading='flat')
(0)  def test_samesizepcolorflaterror():
(4)      fig, ax = plt.subplots()
(4)      x, y = np.meshgrid(np.arange(5), np.arange(3))
(4)      Z = x + y
(4)      with pytest.raises(TypeError, match=r".*one smaller than X"):
(8)          ax.pcolormesh(x, y, Z, shading='flat')
(0)  @pytest.mark.parametrize('snap', [False, True])
(0)  @check_figures_equal(extensions=["png"])
(0)  def test_pcolorauto(fig_test, fig_ref, snap):
(4)      ax = fig_test.subplots()
(4)      x = np.arange(0, 10)
(4)      y = np.arange(0, 4)
(4)      np.random.seed(19680801)
(4)      Z = np.random.randn(3, 9)
(4)      ax.pcolormesh(x, y, Z, snap=snap)
(4)      ax = fig_ref.subplots()
(4)      x2 = x[:-1] + np.diff(x) / 2
(4)      y2 = y[:-1] + np.diff(y) / 2
(4)      ax.pcolormesh(x2, y2, Z, snap=snap)
(0)  @image_comparison(['canonical'], tol=0.02 if platform.machine() == 'arm64'
else 0)
(0)  def test_canonical():
(4)      fig, ax = plt.subplots()
(4)      ax.plot([1, 2, 3])
(0)  @image_comparison(['arc_angles.png'], remove_text=True, style='default')
(0)  def test_arc_angles():
(4)      w = 2
(4)      h = 1
(4)      centre = (0.2, 0.5)
(4)      scale = 2
(4)      fig, axs = plt.subplots(3, 3)
(4)      for i, ax in enumerate(axs.flat):
(8)          theta2 = i * 360 / 9
(8)          theta1 = theta2 - 45
(8)          ax.add_patch(mpatches.Ellipse(centre, w, h, alpha=0.3))
(8)          ax.add_patch(mpatches.Arc(centre, w, h, theta1=theta1, theta2=theta2))
(8)          ax.plot([scale * np.cos(np.deg2rad(theta1)) + centre[0],
(17)                 centre[0],
```

```
 1217: (17)                          scale * np.cos(np.deg2rad(theta2)) + centre[0]],
 1218: (16)                         [scale * np.sin(np.deg2rad(theta1)) + centre[1],
 1219: (17)                          centre[1],
 1220: (17)                          scale * np.sin(np.deg2rad(theta2)) + centre[1]])
 1221: (8)                   ax.set_xlim(-scale, scale)
 1222: (8)                   ax.set_ylim(-scale, scale)
 1223: (8)                   w *= 10
 1224: (8)                   h *= 10
 1225: (8)                   centre = (centre[0] * 10, centre[1] * 10)
 1226: (8)                   scale *= 10
 1227: (0)           @image_comparison(['arc_ellipse'], remove_text=True)
 1228: (0)           def test_arc_ellipse():
 1229: (4)               xcenter, ycenter = 0.38, 0.52
 1230: (4)               width, height = 1e-1, 3e-1
 1231: (4)               angle = -30
 1232: (4)               theta = np.deg2rad(np.arange(360))
 1233: (4)               x = width / 2. * np.cos(theta)
 1234: (4)               y = height / 2. * np.sin(theta)
 1235: (4)               rtheta = np.deg2rad(angle)
 1236: (4)               R = np.array([
 1237: (8)                   [np.cos(rtheta), -np.sin(rtheta)],
 1238: (8)                   [np.sin(rtheta), np.cos(rtheta)]])
 1239: (4)               x, y = np.dot(R, [x, y])
 1240: (4)               x += xcenter
 1241: (4)               y += ycenter
 1242: (4)               fig = plt.figure()
 1243: (4)               ax = fig.add_subplot(211, aspect='auto')
 1244: (4)               ax.fill(x, y, alpha=0.2, facecolor='yellow', edgecolor='yellow',
 1245: (12)                      linewidth=1, zorder=1)
 1246: (4)               e1 = mpatches.Arc((xcenter, ycenter), width, height,
 1247: (22)                                angle=angle, linewidth=2, fill=False, zorder=2)
 1248: (4)               ax.add_patch(e1)
 1249: (4)               ax = fig.add_subplot(212, aspect='equal')
 1250: (4)               ax.fill(x, y, alpha=0.2, facecolor='green', edgecolor='green', zorder=1)
 1251: (4)               e2 = mpatches.Arc((xcenter, ycenter), width, height,
 1252: (22)                                angle=angle, linewidth=2, fill=False, zorder=2)
 1253: (4)               ax.add_patch(e2)
 1254: (0)           def test_marker_as_markerstyle():
 1255: (4)               fix, ax = plt.subplots()
 1256: (4)               m = mmarkers.MarkerStyle('o')
 1257: (4)               ax.plot([1, 2, 3], [3, 2, 1], marker=m)
 1258: (4)               ax.scatter([1, 2, 3], [4, 3, 2], marker=m)
 1259: (4)               ax.errorbar([1, 2, 3], [5, 4, 3], marker=m)
 1260: (0)           @image_comparison(['markevery'], remove_text=True)
 1261: (0)           def test_markevery():
 1262: (4)               x = np.linspace(0, 10, 100)
 1263: (4)               y = np.sin(x) * np.sqrt(x/10 + 0.5)
 1264: (4)               fig, ax = plt.subplots()
 1265: (4)               ax.plot(x, y, 'o', label='default')
 1266: (4)               ax.plot(x, y, 'd', markevery=None, label='mark all')
 1267: (4)               ax.plot(x, y, 's', markevery=10, label='mark every 10')
 1268: (4)               ax.plot(x, y, '+', markevery=(5, 20), label='mark every 5 starting at 10')
 1269: (4)               ax.legend()
 1270: (0)           @image_comparison(['markevery_line'], remove_text=True, tol=0.005)
 1271: (0)           def test_markevery_line():
 1272: (4)               x = np.linspace(0, 10, 100)
 1273: (4)               y = np.sin(x) * np.sqrt(x/10 + 0.5)
 1274: (4)               fig, ax = plt.subplots()
 1275: (4)               ax.plot(x, y, '-o', label='default')
 1276: (4)               ax.plot(x, y, '-d', markevery=None, label='mark all')
 1277: (4)               ax.plot(x, y, '-s', markevery=10, label='mark every 10')
 1278: (4)               ax.plot(x, y, '-+', markevery=(5, 20), label='mark every 5 starting at
 10')
 1279: (4)               ax.legend()
 1280: (0)           @image_comparison(['markevery_linear_scales'], remove_text=True, tol=0.001)
 1281: (0)           def test_markevery_linear_scales():
 1282: (4)               cases = [None,
 1283: (13)                       8,
 1284: (13)                       (30, 8),
```

```
(13)                        [16, 24, 30], [0, -1],
(13)                        slice(100, 200, 3),
(13)                        0.1, 0.3, 1.5,
(13)                        (0.0, 0.1), (0.45, 0.1)]
(4)                 cols = 3
(4)                 gs = matplotlib.gridspec.GridSpec(len(cases) // cols + 1, cols)
(4)                 delta = 0.11
(4)                 x = np.linspace(0, 10 - 2 * delta, 200) + delta
(4)                 y = np.sin(x) + 1.0 + delta
(4)                 for i, case in enumerate(cases):
(8)                     row = (i // cols)
(8)                     col = i % cols
(8)                     plt.subplot(gs[row, col])
(8)                     plt.title('markevery=%s' % str(case))
(8)                     plt.plot(x, y, 'o', ls='-', ms=4,  markevery=case)
(0)             @image_comparison(['markevery_linear_scales_zoomed'], remove_text=True)
(0)             def test_markevery_linear_scales_zoomed():
(4)                 cases = [None,
(13)                         8,
(13)                         (30, 8),
(13)                         [16, 24, 30], [0, -1],
(13)                         slice(100, 200, 3),
(13)                         0.1, 0.3, 1.5,
(13)                         (0.0, 0.1), (0.45, 0.1)]
(4)                 cols = 3
(4)                 gs = matplotlib.gridspec.GridSpec(len(cases) // cols + 1, cols)
(4)                 delta = 0.11
(4)                 x = np.linspace(0, 10 - 2 * delta, 200) + delta
(4)                 y = np.sin(x) + 1.0 + delta
(4)                 for i, case in enumerate(cases):
(8)                     row = (i // cols)
(8)                     col = i % cols
(8)                     plt.subplot(gs[row, col])
(8)                     plt.title('markevery=%s' % str(case))
(8)                     plt.plot(x, y, 'o', ls='-', ms=4,  markevery=case)
(8)                     plt.xlim((6, 6.7))
(8)                     plt.ylim((1.1, 1.7))
(0)             @image_comparison(['markevery_log_scales'], remove_text=True)
(0)             def test_markevery_log_scales():
(4)                 cases = [None,
(13)                         8,
(13)                         (30, 8),
(13)                         [16, 24, 30], [0, -1],
(13)                         slice(100, 200, 3),
(13)                         0.1, 0.3, 1.5,
(13)                         (0.0, 0.1), (0.45, 0.1)]
(4)                 cols = 3
(4)                 gs = matplotlib.gridspec.GridSpec(len(cases) // cols + 1, cols)
(4)                 delta = 0.11
(4)                 x = np.linspace(0, 10 - 2 * delta, 200) + delta
(4)                 y = np.sin(x) + 1.0 + delta
(4)                 for i, case in enumerate(cases):
(8)                     row = (i // cols)
(8)                     col = i % cols
(8)                     plt.subplot(gs[row, col])
(8)                     plt.title('markevery=%s' % str(case))
(8)                     plt.xscale('log')
(8)                     plt.yscale('log')
(8)                     plt.plot(x, y, 'o', ls='-', ms=4,  markevery=case)
(0)             @image_comparison(['markevery_polar'], style='default', remove_text=True)
(0)             def test_markevery_polar():
(4)                 cases = [None,
(13)                         8,
(13)                         (30, 8),
(13)                         [16, 24, 30], [0, -1],
(13)                         slice(100, 200, 3),
(13)                         0.1, 0.3, 1.5,
(13)                         (0.0, 0.1), (0.45, 0.1)]
(4)                 cols = 3
```

```
(4)             gs = matplotlib.gridspec.GridSpec(len(cases) // cols + 1, cols)
(4)             r = np.linspace(0, 3.0, 200)
(4)             theta = 2 * np.pi * r
(4)             for i, case in enumerate(cases):
(8)                 row = (i // cols)
(8)                 col = i % cols
(8)                 plt.subplot(gs[row, col], polar=True)
(8)                 plt.title('markevery=%s' % str(case))
(8)                 plt.plot(theta, r, 'o', ls='-', ms=4,  markevery=case)
(0)         @image_comparison(['markevery_linear_scales_nans'], remove_text=True)
(0)         def test_markevery_linear_scales_nans():
(4)             cases = [None,
(13)                     8,
(13)                     (30, 8),
(13)                     [16, 24, 30], [0, -1],
(13)                     slice(100, 200, 3),
(13)                     0.1, 0.3, 1.5,
(13)                     (0.0, 0.1), (0.45, 0.1)]
(4)             cols = 3
(4)             gs = matplotlib.gridspec.GridSpec(len(cases) // cols + 1, cols)
(4)             delta = 0.11
(4)             x = np.linspace(0, 10 - 2 * delta, 200) + delta
(4)             y = np.sin(x) + 1.0 + delta
(4)             y[:10] = y[-20:] = y[50:70] = np.nan
(4)             for i, case in enumerate(cases):
(8)                 row = (i // cols)
(8)                 col = i % cols
(8)                 plt.subplot(gs[row, col])
(8)                 plt.title('markevery=%s' % str(case))
(8)                 plt.plot(x, y, 'o', ls='-', ms=4,  markevery=case)
(0)         @image_comparison(['marker_edges'], remove_text=True)
(0)         def test_marker_edges():
(4)             x = np.linspace(0, 1, 10)
(4)             fig, ax = plt.subplots()
(4)             ax.plot(x, np.sin(x), 'y.', ms=30.0, mew=0, mec='r')
(4)             ax.plot(x+0.1, np.sin(x), 'y.', ms=30.0, mew=1, mec='r')
(4)             ax.plot(x+0.2, np.sin(x), 'y.', ms=30.0, mew=2, mec='b')
(0)         @image_comparison(['bar_tick_label_single.png', 'bar_tick_label_single.png'])
(0)         def test_bar_tick_label_single():
(4)             ax = plt.gca()
(4)             ax.bar(0, 1, align='edge', tick_label='0')
(4)             data = {"a": 0, "b": 1}
(4)             fig, ax = plt.subplots()
(4)             ax = plt.gca()
(4)             ax.bar("a", "b", align='edge', tick_label='0', data=data)
(0)         def test_nan_bar_values():
(4)             fig, ax = plt.subplots()
(4)             ax.bar([0, 1], [np.nan, 4])
(0)         def test_bar_ticklabel_fail():
(4)             fig, ax = plt.subplots()
(4)             ax.bar([], [])
(0)         @image_comparison(['bar_tick_label_multiple.png'])
(0)         def test_bar_tick_label_multiple():
(4)             ax = plt.gca()
(4)             ax.bar([1, 2.5], [1, 2], width=[0.2, 0.5], tick_label=['a', 'b'],
(11)                    align='center')
(0)         @image_comparison(['bar_tick_label_multiple_old_label_alignment.png'])
(0)         def test_bar_tick_label_multiple_old_alignment():
(4)             matplotlib.rcParams["ytick.alignment"] = "center"
(4)             ax = plt.gca()
(4)             ax.bar([1, 2.5], [1, 2], width=[0.2, 0.5], tick_label=['a', 'b'],
(11)                    align='center')
(0)         @check_figures_equal(extensions=["png"])
(0)         def test_bar_decimal_center(fig_test, fig_ref):
(4)             ax = fig_test.subplots()
(4)             x0 = [1.5, 8.4, 5.3, 4.2]
(4)             y0 = [1.1, 2.2, 3.3, 4.4]
(4)             x = [Decimal(x) for x in x0]
(4)             y = [Decimal(y) for y in y0]
```

```
(4)             ax.bar(x, y, align='center')
(4)             ax = fig_ref.subplots()
(4)             ax.bar(x0, y0, align='center')
(0)         @check_figures_equal(extensions=["png"])
(0)         def test_barh_decimal_center(fig_test, fig_ref):
(4)             ax = fig_test.subplots()
(4)             x0 = [1.5, 8.4, 5.3, 4.2]
(4)             y0 = [1.1, 2.2, 3.3, 4.4]
(4)             x = [Decimal(x) for x in x0]
(4)             y = [Decimal(y) for y in y0]
(4)             ax.barh(x, y, height=[0.5, 0.5, 1, 1], align='center')
(4)             ax = fig_ref.subplots()
(4)             ax.barh(x0, y0, height=[0.5, 0.5, 1, 1], align='center')
(0)         @check_figures_equal(extensions=["png"])
(0)         def test_bar_decimal_width(fig_test, fig_ref):
(4)             x = [1.5, 8.4, 5.3, 4.2]
(4)             y = [1.1, 2.2, 3.3, 4.4]
(4)             w0 = [0.7, 1.45, 1, 2]
(4)             w = [Decimal(i) for i in w0]
(4)             ax = fig_test.subplots()
(4)             ax.bar(x, y, width=w, align='center')
(4)             ax = fig_ref.subplots()
(4)             ax.bar(x, y, width=w0, align='center')
(0)         @check_figures_equal(extensions=["png"])
(0)         def test_barh_decimal_height(fig_test, fig_ref):
(4)             x = [1.5, 8.4, 5.3, 4.2]
(4)             y = [1.1, 2.2, 3.3, 4.4]
(4)             h0 = [0.7, 1.45, 1, 2]
(4)             h = [Decimal(i) for i in h0]
(4)             ax = fig_test.subplots()
(4)             ax.barh(x, y, height=h, align='center')
(4)             ax = fig_ref.subplots()
(4)             ax.barh(x, y, height=h0, align='center')
(0)         def test_bar_color_none_alpha():
(4)             ax = plt.gca()
(4)             rects = ax.bar([1, 2], [2, 4], alpha=0.3, color='none', edgecolor='r')
(4)             for rect in rects:
(8)                 assert rect.get_facecolor() == (0, 0, 0, 0)
(8)                 assert rect.get_edgecolor() == (1, 0, 0, 0.3)
(0)         def test_bar_edgecolor_none_alpha():
(4)             ax = plt.gca()
(4)             rects = ax.bar([1, 2], [2, 4], alpha=0.3, color='r', edgecolor='none')
(4)             for rect in rects:
(8)                 assert rect.get_facecolor() == (1, 0, 0, 0.3)
(8)                 assert rect.get_edgecolor() == (0, 0, 0, 0)
(0)         @image_comparison(['barh_tick_label.png'])
(0)         def test_barh_tick_label():
(4)             ax = plt.gca()
(4)             ax.barh([1, 2.5], [1, 2], height=[0.2, 0.5], tick_label=['a', 'b'],
(12)                    align='center')
(0)         def test_bar_timedelta():
(4)             """Smoketest that bar can handle width and height in delta units."""
(4)             fig, ax = plt.subplots()
(4)             ax.bar(datetime.datetime(2018, 1, 1), 1.,
(11)                   width=datetime.timedelta(hours=3))
(4)             ax.bar(datetime.datetime(2018, 1, 1), 1.,
(11)                   xerr=datetime.timedelta(hours=2),
(11)                   width=datetime.timedelta(hours=3))
(4)             fig, ax = plt.subplots()
(4)             ax.barh(datetime.datetime(2018, 1, 1), 1,
(12)                    height=datetime.timedelta(hours=3))
(4)             ax.barh(datetime.datetime(2018, 1, 1), 1,
(12)                    height=datetime.timedelta(hours=3),
(12)                    yerr=datetime.timedelta(hours=2))
(4)             fig, ax = plt.subplots()
(4)             ax.barh([datetime.datetime(2018, 1, 1), datetime.datetime(2018, 1, 1)],
(12)                    np.array([1, 1.5]),
(12)                    height=datetime.timedelta(hours=3))
(4)             ax.barh([datetime.datetime(2018, 1, 1), datetime.datetime(2018, 1, 1)],
```

```
1492: (12)                      np.array([1, 1.5]),
1493: (12)                      height=[datetime.timedelta(hours=t) for t in [1, 2]])
1494: (4)               ax.broken_barh([(datetime.datetime(2018, 1, 1),
1495: (21)                              datetime.timedelta(hours=1))],
1496: (19)                            (10, 20))
1497: (0)           def test_bar_datetime_start():
1498: (4)               """test that tickers are correct for datetimes"""
1499: (4)               start = np.array([np.datetime64('2012-01-01'), np.datetime64('2012-02-
01'),
1500: (22)                               np.datetime64('2012-01-15')])
1501: (4)               stop = np.array([np.datetime64('2012-02-07'), np.datetime64('2012-02-13'),
1502: (21)                              np.datetime64('2012-02-12')])
1503: (4)               fig, ax = plt.subplots()
1504: (4)               ax.bar([0, 1, 3], height=stop-start, bottom=start)
1505: (4)               assert isinstance(ax.yaxis.get_major_formatter(),
mdates.AutoDateFormatter)
1506: (4)               fig, ax = plt.subplots()
1507: (4)               ax.barh([0, 1, 3], width=stop-start, left=start)
1508: (4)               assert isinstance(ax.xaxis.get_major_formatter(),
mdates.AutoDateFormatter)
1509: (0)           def test_boxplot_dates_pandas(pd):
1510: (4)               data = np.random.rand(5, 2)
1511: (4)               years = pd.date_range('1/1/2000',
1512: (26)                                    periods=2, freq=pd.DateOffset(years=1)).year
1513: (4)               plt.figure()
1514: (4)               plt.boxplot(data, positions=years)
1515: (0)           def test_boxplot_capwidths():
1516: (4)               data = np.random.rand(5, 3)
1517: (4)               fig, axs = plt.subplots(9)
1518: (4)               axs[0].boxplot(data, capwidths=[0.3, 0.2, 0.1], widths=[0.1, 0.2, 0.3])
1519: (4)               axs[1].boxplot(data, capwidths=[0.3, 0.2, 0.1], widths=0.2)
1520: (4)               axs[2].boxplot(data, capwidths=[0.3, 0.2, 0.1])
1521: (4)               axs[3].boxplot(data, capwidths=0.5, widths=[0.1, 0.2, 0.3])
1522: (4)               axs[4].boxplot(data, capwidths=0.5, widths=0.2)
1523: (4)               axs[5].boxplot(data, capwidths=0.5)
1524: (4)               axs[6].boxplot(data, widths=[0.1, 0.2, 0.3])
1525: (4)               axs[7].boxplot(data, widths=0.2)
1526: (4)               axs[8].boxplot(data)
1527: (0)           def test_pcolor_regression(pd):
1528: (4)               from pandas.plotting import (
1529: (8)                   register_matplotlib_converters,
1530: (8)                   deregister_matplotlib_converters,
1531: (4)               )
1532: (4)               fig = plt.figure()
1533: (4)               ax = fig.add_subplot(111)
1534: (4)               times = [datetime.datetime(2021, 1, 1)]
1535: (4)               while len(times) < 7:
1536: (8)                   times.append(times[-1] + datetime.timedelta(seconds=120))
1537: (4)               y_vals = np.arange(5)
1538: (4)               time_axis, y_axis = np.meshgrid(times, y_vals)
1539: (4)               shape = (len(y_vals) - 1, len(times) - 1)
1540: (4)               z_data = np.arange(shape[0] * shape[1])
1541: (4)               z_data.shape = shape
1542: (4)               try:
1543: (8)                   register_matplotlib_converters()
1544: (8)                   im = ax.pcolormesh(time_axis, y_axis, z_data)
1545: (8)                   fig.canvas.draw()
1546: (4)               finally:
1547: (8)                   deregister_matplotlib_converters()
1548: (0)           def test_bar_pandas(pd):
1549: (4)               df = pd.DataFrame(
1550: (8)                   {'year': [2018, 2018, 2018],
1551: (9)                    'month': [1, 1, 1],
1552: (9)                    'day': [1, 2, 3],
1553: (9)                    'value': [1, 2, 3]})
1554: (4)               df['date'] = pd.to_datetime(df[['year', 'month', 'day']])
1555: (4)               monthly = df[['date', 'value']].groupby(['date']).sum()
1556: (4)               dates = monthly.index
1557: (4)               forecast = monthly['value']
```

```
(4)             baseline = monthly['value']
(4)             fig, ax = plt.subplots()
(4)             ax.bar(dates, forecast, width=10, align='center')
(4)             ax.plot(dates, baseline, color='orange', lw=4)
(0)         def test_bar_pandas_indexed(pd):
(4)             df = pd.DataFrame({"x": [1., 2., 3.], "width": [.2, .4, .6]},
(22)                              index=[1, 2, 3])
(4)             fig, ax = plt.subplots()
(4)             ax.bar(df.x, 1., width=df.width)
(0)         @mpl.style.context('default')
(0)         @check_figures_equal()
(0)         def test_bar_hatches(fig_test, fig_ref):
(4)             ax_test = fig_test.subplots()
(4)             ax_ref = fig_ref.subplots()
(4)             x = [1, 2]
(4)             y = [2, 3]
(4)             hatches = ['x', 'o']
(4)             for i in range(2):
(8)                 ax_ref.bar(x[i], y[i], color='C0', hatch=hatches[i])
(4)             ax_test.bar(x, y, hatch=hatches)
(0)         @pytest.mark.parametrize(
(4)             ("x", "width", "label", "expected_labels", "container_label"),
(4)             [
(8)                 ("x", 1, "x", ["_nolegend_"], "x"),
(8)                 (["a", "b", "c"], [10, 20, 15], ["A", "B", "C"],
(9)                  ["A", "B", "C"], "_nolegend_"),
(8)                 (["a", "b", "c"], [10, 20, 15], ["R", "Y", "_nolegend_"],
(9)                  ["R", "Y", "_nolegend_"], "_nolegend_"),
(8)                 (["a", "b", "c"], [10, 20, 15], "bars",
(9)                  ["_nolegend_", "_nolegend_", "_nolegend_"], "bars"),
(4)             ]
(0)         )
(0)         def test_bar_labels(x, width, label, expected_labels, container_label):
(4)             _, ax = plt.subplots()
(4)             bar_container = ax.bar(x, width, label=label)
(4)             bar_labels = [bar.get_label() for bar in bar_container]
(4)             assert expected_labels == bar_labels
(4)             assert bar_container.get_label() == container_label
(0)         def test_bar_labels_length():
(4)             _, ax = plt.subplots()
(4)             with pytest.raises(ValueError):
(8)                 ax.bar(["x", "y"], [1, 2], label=["X", "Y", "Z"])
(4)             _, ax = plt.subplots()
(4)             with pytest.raises(ValueError):
(8)                 ax.bar(["x", "y"], [1, 2], label=["X"])
(0)         def test_pandas_minimal_plot(pd):
(4)             for x in [pd.Series([1, 2], dtype="float64"),
(14)                      pd.Series([1, 2], dtype="Float64")]:
(8)                 plt.plot(x, x)
(8)                 plt.plot(x.index, x)
(8)                 plt.plot(x)
(8)                 plt.plot(x.index)
(4)             df = pd.DataFrame({'col': [1, 2, 3]})
(4)             plt.plot(df)
(4)             plt.plot(df, df)
(0)         @image_comparison(['hist_log'], remove_text=True)
(0)         def test_hist_log():
(4)             data0 = np.linspace(0, 1, 200)**3
(4)             data = np.concatenate([1 - data0, 1 + data0])
(4)             fig, ax = plt.subplots()
(4)             ax.hist(data, fill=False, log=True)
(0)         @check_figures_equal(extensions=["png"])
(0)         def test_hist_log_2(fig_test, fig_ref):
(4)             axs_test = fig_test.subplots(2, 3)
(4)             axs_ref = fig_ref.subplots(2, 3)
(4)             for i, histtype in enumerate(["bar", "step", "stepfilled"]):
(8)                 axs_test[0, i].set_yscale("log")
(8)                 axs_test[0, i].hist(1, 1, histtype=histtype)
(8)                 axs_test[1, i].hist(1, 1, histtype=histtype)
```

```
(8)                 axs_test[1, i].set_yscale("log")
(8)                 for ax in axs_ref[:, i]:
(12)                    ax.hist(1, 1, log=True, histtype=histtype)
(0)         def test_hist_log_barstacked():
(4)             fig, axs = plt.subplots(2)
(4)             axs[0].hist([[0], [0, 1]], 2, histtype="barstacked")
(4)             axs[0].set_yscale("log")
(4)             axs[1].hist([0, 0, 1], 2, histtype="barstacked")
(4)             axs[1].set_yscale("log")
(4)             fig.canvas.draw()
(4)             assert axs[0].get_ylim() == axs[1].get_ylim()
(0)         @image_comparison(['hist_bar_empty.png'], remove_text=True)
(0)         def test_hist_bar_empty():
(4)             ax = plt.gca()
(4)             ax.hist([], histtype='bar')
(0)         def test_hist_float16():
(4)             np.random.seed(19680801)
(4)             values = np.clip(
(8)                 np.random.normal(0.5, 0.3, size=1000), 0, 1).astype(np.float16)
(4)             h = plt.hist(values, bins=3, alpha=0.5)
(4)             bc = h[2]
(4)             for r in range(1, len(bc)):
(8)                 rleft = bc[r-1].get_corners()
(8)                 rright = bc[r].get_corners()
(8)                 assert rleft[1][0] <= rright[0][0]
(0)         @image_comparison(['hist_step_empty.png'], remove_text=True)
(0)         def test_hist_step_empty():
(4)             ax = plt.gca()
(4)             ax.hist([], histtype='step')
(0)         @image_comparison(['hist_step_filled.png'], remove_text=True)
(0)         def test_hist_step_filled():
(4)             np.random.seed(0)
(4)             x = np.random.randn(1000, 3)
(4)             n_bins = 10
(4)             kwargs = [{'fill': True}, {'fill': False}, {'fill': None}, {}]*2
(4)             types = ['step']*4+['stepfilled']*4
(4)             fig, axs = plt.subplots(nrows=2, ncols=4)
(4)             for kg, _type, ax in zip(kwargs, types, axs.flat):
(8)                 ax.hist(x, n_bins, histtype=_type, stacked=True, **kg)
(8)                 ax.set_title(f'{kg}/{_type}')
(8)                 ax.set_ylim(bottom=-50)
(4)             patches = axs[0, 0].patches
(4)             assert all(p.get_facecolor() == p.get_edgecolor() for p in patches)
(0)         @image_comparison(['hist_density.png'])
(0)         def test_hist_density():
(4)             np.random.seed(19680801)
(4)             data = np.random.standard_normal(2000)
(4)             fig, ax = plt.subplots()
(4)             ax.hist(data, density=True)
(0)         def test_hist_unequal_bins_density():
(4)             rng = np.random.RandomState(57483)
(4)             t = rng.randn(100)
(4)             bins = [-3, -1, -0.5, 0, 1, 5]
(4)             mpl_heights, _, _ = plt.hist(t, bins=bins, density=True)
(4)             np_heights, _ = np.histogram(t, bins=bins, density=True)
(4)             assert_allclose(mpl_heights, np_heights)
(0)         def test_hist_datetime_datasets():
(4)             data = [[datetime.datetime(2017, 1, 1), datetime.datetime(2017, 1, 1)],
(12)                    [datetime.datetime(2017, 1, 1), datetime.datetime(2017, 1, 2)]]
(4)             fig, ax = plt.subplots()
(4)             ax.hist(data, stacked=True)
(4)             ax.hist(data, stacked=False)
(0)         @pytest.mark.parametrize("bins_preprocess",
(25)                                 [mpl.dates.date2num,
(26)                                  lambda bins: bins,
(26)                                  lambda bins: np.asarray(bins, 'datetime64')],
(25)                                 ids=['date2num', 'datetime.datetime',
(30)                                      'np.datetime64'])
(0)         def test_hist_datetime_datasets_bins(bins_preprocess):
```

```
(4)             data = [[datetime.datetime(2019, 1, 5), datetime.datetime(2019, 1, 11),
(13)                     datetime.datetime(2019, 2, 1), datetime.datetime(2019, 3, 1)],
(12)                    [datetime.datetime(2019, 1, 11), datetime.datetime(2019, 2, 5),
(13)                     datetime.datetime(2019, 2, 18), datetime.datetime(2019, 3, 1)]]
(4)             date_edges = [datetime.datetime(2019, 1, 1), datetime.datetime(2019, 2,
1),
(18)                             datetime.datetime(2019, 3, 1)]
(4)             fig, ax = plt.subplots()
(4)             _, bins, _ = ax.hist(data, bins=bins_preprocess(date_edges), stacked=True)
(4)             np.testing.assert_allclose(bins, mpl.dates.date2num(date_edges))
(4)             _, bins, _ = ax.hist(data, bins=bins_preprocess(date_edges),
stacked=False)
(4)             np.testing.assert_allclose(bins, mpl.dates.date2num(date_edges))
(0)         @pytest.mark.parametrize('data, expected_number_of_hists',
(25)                                [([], 1),
(26)                                 ([[]], 1),
(26)                                 ([[], []], 2)])
(0)         def test_hist_with_empty_input(data, expected_number_of_hists):
(4)             hists, _, _ = plt.hist(data)
(4)             hists = np.asarray(hists)
(4)             if hists.ndim == 1:
(8)                 assert 1 == expected_number_of_hists
(4)             else:
(8)                 assert hists.shape[0] == expected_number_of_hists
(0)         @pytest.mark.parametrize("histtype, zorder",
(25)                                [("bar", mpl.patches.Patch.zorder),
(26)                                 ("step", mpl.lines.Line2D.zorder),
(26)                                 ("stepfilled", mpl.patches.Patch.zorder)])
(0)         def test_hist_zorder(histtype, zorder):
(4)             ax = plt.figure().add_subplot()
(4)             ax.hist([1, 2], histtype=histtype)
(4)             assert ax.patches
(4)             for patch in ax.patches:
(8)                 assert patch.get_zorder() == zorder
(0)         @check_figures_equal(extensions=['png'])
(0)         def test_stairs(fig_test, fig_ref):
(4)             import matplotlib.lines as mlines
(4)             y = np.array([6, 14, 32, 37, 48, 32, 21,  4])  # hist
(4)             x = np.array([1., 2., 3., 4., 5., 6., 7., 8., 9.])  # bins
(4)             test_axes = fig_test.subplots(3, 2).flatten()
(4)             test_axes[0].stairs(y, x, baseline=None)
(4)             test_axes[1].stairs(y, x, baseline=None, orientation='horizontal')
(4)             test_axes[2].stairs(y, x)
(4)             test_axes[3].stairs(y, x, orientation='horizontal')
(4)             test_axes[4].stairs(y, x)
(4)             test_axes[4].semilogy()
(4)             test_axes[5].stairs(y, x, orientation='horizontal')
(4)             test_axes[5].semilogx()
(4)             style = {'solid_joinstyle': 'miter', 'solid_capstyle': 'butt'}
(4)             ref_axes = fig_ref.subplots(3, 2).flatten()
(4)             ref_axes[0].plot(x, np.append(y, y[-1]), drawstyle='steps-post', **style)
(4)             ref_axes[1].plot(np.append(y[0], y), x, drawstyle='steps-post', **style)
(4)             ref_axes[2].plot(x, np.append(y, y[-1]), drawstyle='steps-post', **style)
(4)             ref_axes[2].add_line(mlines.Line2D([x[0], x[0]], [0, y[0]], **style))
(4)             ref_axes[2].add_line(mlines.Line2D([x[-1], x[-1]], [0, y[-1]], **style))
(4)             ref_axes[2].set_ylim(0, None)
(4)             ref_axes[3].plot(np.append(y[0], y), x, drawstyle='steps-post', **style)
(4)             ref_axes[3].add_line(mlines.Line2D([0, y[0]], [x[0], x[0]], **style))
(4)             ref_axes[3].add_line(mlines.Line2D([0, y[-1]], [x[-1], x[-1]], **style))
(4)             ref_axes[3].set_xlim(0, None)
(4)             ref_axes[4].plot(x, np.append(y, y[-1]), drawstyle='steps-post', **style)
(4)             ref_axes[4].add_line(mlines.Line2D([x[0], x[0]], [0, y[0]], **style))
(4)             ref_axes[4].add_line(mlines.Line2D([x[-1], x[-1]], [0, y[-1]], **style))
(4)             ref_axes[4].semilogy()
(4)             ref_axes[5].plot(np.append(y[0], y), x, drawstyle='steps-post', **style)
(4)             ref_axes[5].add_line(mlines.Line2D([0, y[0]], [x[0], x[0]], **style))
(4)             ref_axes[5].add_line(mlines.Line2D([0, y[-1]], [x[-1], x[-1]], **style))
(4)             ref_axes[5].semilogx()
(0)         @check_figures_equal(extensions=['png'])
```

```
(0)          def test_stairs_fill(fig_test, fig_ref):
(4)              h, bins = [1, 2, 3, 4, 2], [0, 1, 2, 3, 4, 5]
(4)              bs = -2
(4)              test_axes = fig_test.subplots(2, 2).flatten()
(4)              test_axes[0].stairs(h, bins, fill=True)
(4)              test_axes[1].stairs(h, bins, orientation='horizontal', fill=True)
(4)              test_axes[2].stairs(h, bins, baseline=bs, fill=True)
(4)              test_axes[3].stairs(h, bins, baseline=bs, orientation='horizontal',
(24)                                 fill=True)
(4)              ref_axes = fig_ref.subplots(2, 2).flatten()
(4)              ref_axes[0].fill_between(bins, np.append(h, h[-1]), step='post', lw=0)
(4)              ref_axes[0].set_ylim(0, None)
(4)              ref_axes[1].fill_betweenx(bins, np.append(h, h[-1]), step='post', lw=0)
(4)              ref_axes[1].set_xlim(0, None)
(4)              ref_axes[2].fill_between(bins, np.append(h, h[-1]),
(29)                                      np.ones(len(h)+1)*bs, step='post', lw=0)
(4)              ref_axes[2].set_ylim(bs, None)
(4)              ref_axes[3].fill_betweenx(bins, np.append(h, h[-1]),
(30)                                       np.ones(len(h)+1)*bs, step='post', lw=0)
(4)              ref_axes[3].set_xlim(bs, None)
(0)          @check_figures_equal(extensions=['png'])
(0)          def test_stairs_update(fig_test, fig_ref):
(4)              ylim = -3, 4
(4)              test_ax = fig_test.add_subplot()
(4)              h = test_ax.stairs([1, 2, 3])
(4)              test_ax.set_ylim(ylim)
(4)              h.set_data([3, 2, 1])
(4)              h.set_data(edges=np.arange(4)+2)
(4)              h.set_data([1, 2, 1], np.arange(4)/2)
(4)              h.set_data([1, 2, 3])
(4)              h.set_data(None, np.arange(4))
(4)              assert np.allclose(h.get_data()[0], np.arange(1, 4))
(4)              assert np.allclose(h.get_data()[1], np.arange(4))
(4)              h.set_data(baseline=-2)
(4)              assert h.get_data().baseline == -2
(4)              ref_ax = fig_ref.add_subplot()
(4)              h = ref_ax.stairs([1, 2, 3], baseline=-2)
(4)              ref_ax.set_ylim(ylim)
(0)          @check_figures_equal(extensions=['png'])
(0)          def test_stairs_baseline_0(fig_test, fig_ref):
(4)              test_ax = fig_test.add_subplot()
(4)              test_ax.stairs([5, 6, 7], baseline=None)
(4)              ref_ax = fig_ref.add_subplot()
(4)              style = {'solid_joinstyle': 'miter', 'solid_capstyle': 'butt'}
(4)              ref_ax.plot(range(4), [5, 6, 7, 7], drawstyle='steps-post', **style)
(4)              ref_ax.set_ylim(0, None)
(0)          def test_stairs_empty():
(4)              ax = plt.figure().add_subplot()
(4)              ax.stairs([], [42])
(4)              assert ax.get_xlim() == (39, 45)
(4)              assert ax.get_ylim() == (-0.06, 0.06)
(0)          def test_stairs_invalid_nan():
(4)              with pytest.raises(ValueError, match='Nan values in "edges"'):
(8)                  plt.stairs([1, 2], [0, np.nan, 1])
(0)          def test_stairs_invalid_mismatch():
(4)              with pytest.raises(ValueError, match='Size mismatch'):
(8)                  plt.stairs([1, 2], [0, 1])
(0)          def test_stairs_invalid_update():
(4)              h = plt.stairs([1, 2], [0, 1, 2])
(4)              with pytest.raises(ValueError, match='Nan values in "edges"'):
(8)                  h.set_data(edges=[1, np.nan, 2])
(0)          def test_stairs_invalid_update2():
(4)              h = plt.stairs([1, 2], [0, 1, 2])
(4)              with pytest.raises(ValueError, match='Size mismatch'):
(8)                  h.set_data(edges=np.arange(5))
(0)          @image_comparison(['test_stairs_options.png'], remove_text=True)
(0)          def test_stairs_options():
(4)              x, y = np.array([1, 2, 3, 4, 5]), np.array([1, 2, 3, 4]).astype(float)
(4)              yn = y.copy()
```

```
(4)             yn[1] = np.nan
(4)             fig, ax = plt.subplots()
(4)             ax.stairs(y*3, x, color='green', fill=True, label="A")
(4)             ax.stairs(y, x*3-3, color='red', fill=True,
(14)                      orientation='horizontal', label="B")
(4)             ax.stairs(yn, x, color='orange', ls='--', lw=2, label="C")
(4)             ax.stairs(yn/3, x*3-2, ls='--', lw=2, baseline=0.5,
(14)                      orientation='horizontal', label="D")
(4)             ax.stairs(y[::-1]*3+13, x-1, color='red', ls='--', lw=2, baseline=None,
(14)                      label="E")
(4)             ax.stairs(y[::-1]*3+14, x, baseline=26,
(14)                      color='purple', ls='--', lw=2, label="F")
(4)             ax.stairs(yn[::-1]*3+15, x+1, baseline=np.linspace(27, 25, len(y)),
(14)                      color='blue', ls='--', label="G", fill=True)
(4)             ax.stairs(y[:-1][::-1]*2+11, x[:-1]+0.5, color='black', ls='--', lw=2,
(14)                      baseline=12, hatch='//', label="H")
(4)             ax.legend(loc=0)
(0)         @image_comparison(['test_stairs_datetime.png'])
(0)         def test_stairs_datetime():
(4)             f, ax = plt.subplots(constrained_layout=True)
(4)             ax.stairs(np.arange(36),
(14)                      np.arange(np.datetime64('2001-12-27'),
(24)                                np.datetime64('2002-02-02')))
(4)             plt.xticks(rotation=30)
(0)         @check_figures_equal(extensions=['png'])
(0)         def test_stairs_edge_handling(fig_test, fig_ref):
(4)             test_ax = fig_test.add_subplot()
(4)             test_ax.stairs([1, 2, 3], color='red', fill=True)
(4)             ref_ax = fig_ref.add_subplot()
(4)             st = ref_ax.stairs([1, 2, 3], fill=True)
(4)             st.set_color('red')
(0)         def contour_dat():
(4)             x = np.linspace(-3, 5, 150)
(4)             y = np.linspace(-3, 5, 120)
(4)             z = np.cos(x) + np.sin(y[:, np.newaxis])
(4)             return x, y, z
(0)         @image_comparison(['contour_hatching'], remove_text=True, style='mpl20')
(0)         def test_contour_hatching():
(4)             x, y, z = contour_dat()
(4)             fig, ax = plt.subplots()
(4)             ax.contourf(x, y, z, 7, hatches=['/', '\\', '//', '-'],
(16)                        cmap=mpl.colormaps['gray'],
(16)                        extend='both', alpha=0.5)
(0)         @image_comparison(
(4)             ['contour_colorbar'], style='mpl20',
(4)             tol=0.54 if platform.machine() in ('aarch64', 'ppc64le', 's390x') else 0)
(0)         def test_contour_colorbar():
(4)             x, y, z = contour_dat()
(4)             fig, ax = plt.subplots()
(4)             cs = ax.contourf(x, y, z, levels=np.arange(-1.8, 1.801, 0.2),
(21)                             cmap=mpl.colormaps['RdBu'],
(21)                             vmin=-0.6,
(21)                             vmax=0.6,
(21)                             extend='both')
(4)             cs1 = ax.contour(x, y, z, levels=np.arange(-2.2, -0.599, 0.2),
(21)                             colors=['y'],
(21)                             linestyles='solid',
(21)                             linewidths=2)
(4)             cs2 = ax.contour(x, y, z, levels=np.arange(0.6, 2.2, 0.2),
(21)                             colors=['c'],
(21)                             linewidths=2)
(4)             cbar = fig.colorbar(cs, ax=ax)
(4)             cbar.add_lines(cs1)
(4)             cbar.add_lines(cs2, erase=False)
(0)         @image_comparison(['hist2d', 'hist2d'], remove_text=True, style='mpl20')
(0)         def test_hist2d():
(4)             plt.rcParams['pcolormesh.snap'] = False
(4)             np.random.seed(0)
(4)             x = np.random.randn(100)*2+5
```

```
(4)             y = np.random.randn(100)-2
(4)             fig, ax = plt.subplots()
(4)             ax.hist2d(x, y, bins=10, rasterized=True)
(4)             data = {"x": x, "y": y}
(4)             fig, ax = plt.subplots()
(4)             ax.hist2d("x", "y", bins=10, data=data, rasterized=True)
(0)         @image_comparison(['hist2d_transpose'], remove_text=True, style='mpl20')
(0)         def test_hist2d_transpose():
(4)             plt.rcParams['pcolormesh.snap'] = False
(4)             np.random.seed(0)
(4)             x = np.array([5]*100)
(4)             y = np.random.randn(100)-2
(4)             fig, ax = plt.subplots()
(4)             ax.hist2d(x, y, bins=10, rasterized=True)
(0)         def test_hist2d_density():
(4)             x, y = np.random.random((2, 100))
(4)             ax = plt.figure().subplots()
(4)             for obj in [ax, plt]:
(8)                 obj.hist2d(x, y, density=True)
(0)         class TestScatter:
(4)             @image_comparison(['scatter'], style='mpl20', remove_text=True)
(4)             def test_scatter_plot(self):
(8)                 data = {"x": np.array([3, 4, 2, 6]), "y": np.array([2, 5, 2, 3]),
(16)                        "c": ['r', 'y', 'b', 'lime'], "s": [24, 15, 19, 29],
(16)                        "c2": ['0.5', '0.6', '0.7', '0.8']}
(8)                 fig, ax = plt.subplots()
(8)                 ax.scatter(data["x"] - 1., data["y"] - 1., c=data["c"], s=data["s"])
(8)                 ax.scatter(data["x"] + 1., data["y"] + 1., c=data["c2"], s=data["s"])
(8)                 ax.scatter("x", "y", c="c", s="s", data=data)
(4)             @image_comparison(['scatter_marker.png'], remove_text=True)
(4)             def test_scatter_marker(self):
(8)                 fig, (ax0, ax1, ax2) = plt.subplots(ncols=3)
(8)                 ax0.scatter([3, 4, 2, 6], [2, 5, 2, 3],
(20)                            c=[(1, 0, 0), 'y', 'b', 'lime'],
(20)                            s=[60, 50, 40, 30],
(20)                            edgecolors=['k', 'r', 'g', 'b'],
(20)                            marker='s')
(8)                 ax1.scatter([3, 4, 2, 6], [2, 5, 2, 3],
(20)                            c=[(1, 0, 0), 'y', 'b', 'lime'],
(20)                            s=[60, 50, 40, 30],
(20)                            edgecolors=['k', 'r', 'g', 'b'],
(20)                            marker=mmarkers.MarkerStyle('o', fillstyle='top'))
(8)                 rx, ry = 3, 1
(8)                 area = rx * ry * np.pi
(8)                 theta = np.linspace(0, 2 * np.pi, 21)
(8)                 verts = np.column_stack([np.cos(theta) * rx / area,
(33)                                         np.sin(theta) * ry / area])
(8)                 ax2.scatter([3, 4, 2, 6], [2, 5, 2, 3],
(20)                            c=[(1, 0, 0), 'y', 'b', 'lime'],
(20)                            s=[60, 50, 40, 30],
(20)                            edgecolors=['k', 'r', 'g', 'b'],
(20)                            marker=verts)
(4)             @image_comparison(['scatter_2D'], remove_text=True, extensions=['png'])
(4)             def test_scatter_2D(self):
(8)                 x = np.arange(3)
(8)                 y = np.arange(2)
(8)                 x, y = np.meshgrid(x, y)
(8)                 z = x + y
(8)                 fig, ax = plt.subplots()
(8)                 ax.scatter(x, y, c=z, s=200, edgecolors='face')
(4)             @check_figures_equal(extensions=["png"])
(4)             def test_scatter_decimal(self, fig_test, fig_ref):
(8)                 x0 = np.array([1.5, 8.4, 5.3, 4.2])
(8)                 y0 = np.array([1.1, 2.2, 3.3, 4.4])
(8)                 x = np.array([Decimal(i) for i in x0])
(8)                 y = np.array([Decimal(i) for i in y0])
(8)                 c = ['r', 'y', 'b', 'lime']
(8)                 s = [24, 15, 19, 29]
(8)                 ax = fig_test.subplots()
```

```
1970: (8)                    ax.scatter(x, y, c=c, s=s)
1971: (8)                    ax = fig_ref.subplots()
1972: (8)                    ax.scatter(x0, y0, c=c, s=s)
1973: (4)                def test_scatter_color(self):
1974: (8)                    with pytest.raises(ValueError):
1975: (12)                       plt.scatter([1, 2], [1, 2], color=[0.1, 0.2])
1976: (8)                    with pytest.raises(ValueError):
1977: (12)                       plt.scatter([1, 2, 3], [1, 2, 3], color=[1, 2, 3])
1978: (4)                @pytest.mark.parametrize('kwargs',
1979: (32)                                        [
1980: (36)                                            {'cmap': 'gray'},
1981: (36)                                            {'norm': mcolors.Normalize()},
1982: (36)                                            {'vmin': 0},
1983: (36)                                            {'vmax': 0}
1984: (32)                                        ])
1985: (4)                def test_scatter_color_warning(self, kwargs):
1986: (8)                    warn_match = "No data for colormapping provided "
1987: (8)                    with pytest.warns(Warning, match=warn_match):
1988: (12)                       plt.scatter([], [], **kwargs)
1989: (8)                    with pytest.warns(Warning, match=warn_match):
1990: (12)                       plt.scatter([1, 2], [3, 4], c=[], **kwargs)
1991: (8)                    plt.scatter([], [], c=[], **kwargs)
1992: (8)                    plt.scatter([1, 2], [3, 4], c=[4, 5], **kwargs)
1993: (4)                def test_scatter_unfilled(self):
1994: (8)                    coll = plt.scatter([0, 1, 2], [1, 3, 2], c=['0.1', '0.3', '0.5'],
1995: (27)                                      marker=mmarkers.MarkerStyle('o', fillstyle='none'),
1996: (27)                                      linewidths=[1.1, 1.2, 1.3])
1997: (8)                    assert coll.get_facecolors().shape == (0, 4)  # no facecolors
1998: (8)                    assert_array_equal(coll.get_edgecolors(), [[0.1, 0.1, 0.1, 1],
1999: (51)                                                              [0.3, 0.3, 0.3, 1],
2000: (51)                                                              [0.5, 0.5, 0.5, 1]])
2001: (8)                    assert_array_equal(coll.get_linewidths(), [1.1, 1.2, 1.3])
2002: (4)                @mpl.style.context('default')
2003: (4)                def test_scatter_unfillable(self):
2004: (8)                    coll = plt.scatter([0, 1, 2], [1, 3, 2], c=['0.1', '0.3', '0.5'],
2005: (27)                                      marker='x',
2006: (27)                                      linewidths=[1.1, 1.2, 1.3])
2007: (8)                    assert_array_equal(coll.get_facecolors(), coll.get_edgecolors())
2008: (8)                    assert_array_equal(coll.get_edgecolors(), [[0.1, 0.1, 0.1, 1],
2009: (51)                                                              [0.3, 0.3, 0.3, 1],
2010: (51)                                                              [0.5, 0.5, 0.5, 1]])
2011: (8)                    assert_array_equal(coll.get_linewidths(), [1.1, 1.2, 1.3])
2012: (4)                def test_scatter_size_arg_size(self):
2013: (8)                    x = np.arange(4)
2014: (8)                    with pytest.raises(ValueError, match='same size as x and y'):
2015: (12)                       plt.scatter(x, x, x[1:])
2016: (8)                    with pytest.raises(ValueError, match='same size as x and y'):
2017: (12)                       plt.scatter(x[1:], x[1:], x)
2018: (8)                    with pytest.raises(ValueError, match='float array-like'):
2019: (12)                       plt.scatter(x, x, 'foo')
2020: (4)                def test_scatter_edgecolor_RGB(self):
2021: (8)                    coll = plt.scatter([1, 2, 3], [1, np.nan, np.nan],
2022: (28)                                       edgecolor=(1, 0, 0))
2023: (8)                    assert mcolors.same_color(coll.get_edgecolor(), (1, 0, 0))
2024: (8)                    coll = plt.scatter([1, 2, 3, 4], [1, np.nan, np.nan, 1],
2025: (28)                                       edgecolor=(1, 0, 0, 1))
2026: (8)                    assert mcolors.same_color(coll.get_edgecolor(), (1, 0, 0, 1))
2027: (4)                @check_figures_equal(extensions=["png"])
2028: (4)                def test_scatter_invalid_color(self, fig_test, fig_ref):
2029: (8)                    ax = fig_test.subplots()
2030: (8)                    cmap = mpl.colormaps["viridis"].resampled(16)
2031: (8)                    cmap.set_bad("k", 1)
2032: (8)                    ax.scatter(range(4), range(4),
2033: (19)                              c=[1, np.nan, 2, np.nan], s=[1, 2, 3, 4],
2034: (19)                              cmap=cmap, plotnonfinite=True)
2035: (8)                    ax = fig_ref.subplots()
2036: (8)                    cmap = mpl.colormaps["viridis"].resampled(16)
2037: (8)                    ax.scatter([0, 2], [0, 2], c=[1, 2], s=[1, 3], cmap=cmap)
2038: (8)                    ax.scatter([1, 3], [1, 3], s=[2, 4], color="k")
```

```
(4)    @check_figures_equal(extensions=["png"])
(4)    def test_scatter_no_invalid_color(self, fig_test, fig_ref):
(8)        ax = fig_test.subplots()
(8)        cmap = mpl.colormaps["viridis"].resampled(16)
(8)        cmap.set_bad("k", 1)
(8)        ax.scatter(range(4), range(4),
(19)                   c=[1, np.nan, 2, np.nan], s=[1, 2, 3, 4],
(19)                   cmap=cmap, plotnonfinite=False)
(8)        ax = fig_ref.subplots()
(8)        ax.scatter([0, 2], [0, 2], c=[1, 2], s=[1, 3], cmap=cmap)
(4)    def test_scatter_norm_vminvmax(self):
(8)        """Parameters vmin, vmax should error if norm is given."""
(8)        x = [1, 2, 3]
(8)        ax = plt.axes()
(8)        with pytest.raises(ValueError,
(27)                           match="Passing a Normalize instance simultaneously
"
(33)                                   "with vmin/vmax is not supported."):
(12)           ax.scatter(x, x, c=x, norm=mcolors.Normalize(-10, 10),
(23)                      vmin=0, vmax=5)
(4)    @check_figures_equal(extensions=["png"])
(4)    def test_scatter_single_point(self, fig_test, fig_ref):
(8)        ax = fig_test.subplots()
(8)        ax.scatter(1, 1, c=1)
(8)        ax = fig_ref.subplots()
(8)        ax.scatter([1], [1], c=[1])
(4)    @check_figures_equal(extensions=["png"])
(4)    def test_scatter_different_shapes(self, fig_test, fig_ref):
(8)        x = np.arange(10)
(8)        ax = fig_test.subplots()
(8)        ax.scatter(x, x.reshape(2, 5), c=x.reshape(5, 2))
(8)        ax = fig_ref.subplots()
(8)        ax.scatter(x.reshape(5, 2), x, c=x.reshape(2, 5))
(4)    params_test_scatter_c = [
(8)        ('0.5', None),
(8)        (["rgby"], "conversion"),
(8)        ("red", None),
(8)        ("none", None),
(8)        (None, None),
(8)        (["r", "g", "b", "none"], None),
(8)        ("jaune", "conversion"),
(8)        (["jaune"], "conversion"),  # wrong type before wrong size
(8)        (["jaune"]*4, "conversion"),
(8)        ([0.5]*3, None),  # should emit a warning for user's eyes though
(8)        ([0.5]*4, None),  # NB: no warning as matching size allows mapping
(8)        ([0.5]*5, "shape"),
(8)        (['0.5', '0.4', '0.6', '0.7'], None),
(8)        (['0.5', 'red', '0.6', 'C5'], None),
(8)        (['0.5', 0.5, '0.6', 'C5'], "conversion"),
(8)        ([[1, 0, 0]], None),
(8)        ([[1, 0, 0]]*3, "shape"),
(8)        ([[1, 0, 0]]*4, None),
(8)        ([[1, 0, 0]]*5, "shape"),
(8)        ([[1, 0, 0, 0.5]], None),
(8)        ([[1, 0, 0, 0.5]]*3, "shape"),
(8)        ([[1, 0, 0, 0.5]]*4, None),
(8)        ([[1, 0, 0, 0.5]]*5, "shape"),
(8)        ([[1, 0, 0, 0.5]]*3 + [[1, 0, 0]], None),
(8)        ([[1, 0, 0, 0.5], "red", "0.0"], "shape"),
(8)        ([[1, 0, 0, 0.5], "red", "0.0", "C5"], None),
(8)        ([[1, 0, 0, 0.5], "red", "0.0", "C5", [0, 1, 0]], "shape"),
(8)        ([[1, 0, 0, 0.5], "red", "jaune"], "conversion"),
(8)        ([[1, 0, 0, 0.5], "red", "0.0", "jaune"], "conversion"),
(8)        ([[1, 0, 0, 0.5], "red", "0.0", "C5", "jaune"], "conversion"),
(4)    ]
(4)    @pytest.mark.parametrize('c_case, re_key', params_test_scatter_c)
(4)    def test_scatter_c(self, c_case, re_key):
(8)        def get_next_color():
(12)           return 'blue'  # currently unused
```

```
2107: (8)                   xsize = 4
2108: (8)                   REGEXP = {
2109: (12)                      "shape": "^'c' argument has [0-9]+ elements",  # shape mismatch
2110: (12)                      "conversion": "^'c' argument must be a color",  # bad vals
2111: (12)                      }
2112: (8)                   assert_context = (
2113: (12)                      pytest.raises(ValueError, match=REGEXP[re_key])
2114: (12)                      if re_key is not None
2115: (12)                      else pytest.warns(match="argument looks like a single numeric
RGB")
2116: (12)                      if isinstance(c_case, list) and len(c_case) == 3
2117: (12)                      else contextlib.nullcontext()
2118: (8)                   )
2119: (8)                   with assert_context:
2120: (12)                      mpl.axes.Axes._parse_scatter_color_args(
2121: (16)                          c=c_case, edgecolors="black", kwargs={}, xsize=xsize,
2122: (16)                          get_next_color_func=get_next_color)
2123: (4)               @mpl.style.context('default')
2124: (4)               @check_figures_equal(extensions=["png"])
2125: (4)               def test_scatter_single_color_c(self, fig_test, fig_ref):
2126: (8)                   rgb = [[1, 0.5, 0.05]]
2127: (8)                   rgba = [[1, 0.5, 0.05, .5]]
2128: (8)                   ax_ref = fig_ref.subplots()
2129: (8)                   ax_ref.scatter(np.ones(3), range(3), color=rgb)
2130: (8)                   ax_ref.scatter(np.ones(4)*2, range(4), color=rgba)
2131: (8)                   ax_test = fig_test.subplots()
2132: (8)                   ax_test.scatter(np.ones(3), range(3), c=rgb)
2133: (8)                   ax_test.scatter(np.ones(4)*2, range(4), c=rgba)
2134: (4)               def test_scatter_linewidths(self):
2135: (8)                   x = np.arange(5)
2136: (8)                   fig, ax = plt.subplots()
2137: (8)                   for i in range(3):
2138: (12)                      pc = ax.scatter(x, np.full(5, i), c=f'C{i}', marker='x', s=100,
2139: (28)                                      linewidths=i + 1)
2140: (12)                      assert pc.get_linewidths() == i + 1
2141: (8)                   pc = ax.scatter(x, np.full(5, 3), c='C3', marker='x', s=100,
2142: (24)                                  linewidths=[*range(1, 5), None])
2143: (8)                   assert_array_equal(pc.get_linewidths(),
2144: (27)                                     [*range(1, 5), mpl.rcParams['lines.linewidth']])
2145: (4)               def test_scatter_singular_plural_arguments(self):
2146: (8)                   with pytest.raises(TypeError,
2147: (27)                                     match="Got both 'linewidth' and 'linewidths',\
2148: (1)            which are aliases of one another"):
2149: (12)                      plt.scatter([1, 2, 3], [1, 2, 3], linewidths=[0.5, 0.4, 0.3],
linewidth=0.2)
2150: (8)                   with pytest.raises(TypeError,
2151: (27)                                     match="Got both 'edgecolor' and 'edgecolors',\
2152: (1)            which are aliases of one another"):
2153: (12)                      plt.scatter([1, 2, 3], [1, 2, 3],
2154: (24)                                  edgecolors=["#ffffff", "#000000", "#f0f0f0"],
2155: (26)                                    edgecolor="#ffffff")
2156: (8)                   with pytest.raises(TypeError,
2157: (27)                                     match="Got both 'facecolors' and 'facecolor',\
2158: (1)            which are aliases of one another"):
2159: (12)                      plt.scatter([1, 2, 3], [1, 2, 3],
2160: (24)                                  facecolors=["#ffffff", "#000000", "#f0f0f0"],
2161: (28)                                      facecolor="#ffffff")
2162: (0)            def _params(c=None, xsize=2, *, edgecolors=None, **kwargs):
2163: (4)               return (c, edgecolors, kwargs if kwargs is not None else {}, xsize)
2164: (0)            _result = namedtuple('_result', 'c, colors')
2165: (0)            @pytest.mark.parametrize(
2166: (4)               'params, expected_result',
2167: (4)               [(_params(),
2168: (6)                 _result(c='b', colors=np.array([[0, 0, 1, 1]]))),
2169: (5)                (_params(c='r'),
2170: (6)                 _result(c='r', colors=np.array([[1, 0, 0, 1]]))),
2171: (5)                (_params(c='r', colors='b'),
2172: (6)                 _result(c='r', colors=np.array([[1, 0, 0, 1]]))),
2173: (5)                (_params(color='b'),
```

```
 2174: (6)                 _result(c='b', colors=np.array([[0, 0, 1, 1]]))),
 2175: (5)                (_params(color=['b', 'g']),
 2176: (6)                 _result(c=['b', 'g'], colors=np.array([[0, 0, 1, 1], [0, .5, 0, 1]]))),
 2177: (5)                ])
 2178: (0)            def test_parse_scatter_color_args(params, expected_result):
 2179: (4)                def get_next_color():
 2180: (8)                    return 'blue'  # currently unused
 2181: (4)                c, colors, _edgecolors = mpl.axes.Axes._parse_scatter_color_args(
 2182: (8)                    *params, get_next_color_func=get_next_color)
 2183: (4)                assert c == expected_result.c
 2184: (4)                assert_allclose(colors, expected_result.colors)
 2185: (0)            del _params
 2186: (0)            del _result
 2187: (0)            @pytest.mark.parametrize(
 2188: (4)                'kwargs, expected_edgecolors',
 2189: (4)                [(dict(), None),
 2190: (5)                 (dict(c='b'), None),
 2191: (5)                 (dict(edgecolors='r'), 'r'),
 2192: (5)                 (dict(edgecolors=['r', 'g']), ['r', 'g']),
 2193: (5)                 (dict(edgecolor='r'), 'r'),
 2194: (5)                 (dict(edgecolors='face'), 'face'),
 2195: (5)                 (dict(edgecolors='none'), 'none'),
 2196: (5)                 (dict(edgecolor='r', edgecolors='g'), 'r'),
 2197: (5)                 (dict(c='b', edgecolor='r', edgecolors='g'), 'r'),
 2198: (5)                 (dict(color='r'), 'r'),
 2199: (5)                 (dict(color='r', edgecolor='g'), 'g'),
 2200: (5)                 ])
 2201: (0)            def test_parse_scatter_color_args_edgecolors(kwargs, expected_edgecolors):
 2202: (4)                def get_next_color():
 2203: (8)                    return 'blue'  # currently unused
 2204: (4)                c = kwargs.pop('c', None)
 2205: (4)                edgecolors = kwargs.pop('edgecolors', None)
 2206: (4)                _, _, result_edgecolors = \
 2207: (8)                    mpl.axes.Axes._parse_scatter_color_args(
 2208: (12)                       c, edgecolors, kwargs, xsize=2,
get_next_color_func=get_next_color)
 2209: (4)                assert result_edgecolors == expected_edgecolors
 2210: (0)            def test_parse_scatter_color_args_error():
 2211: (4)                def get_next_color():
 2212: (8)                    return 'blue'  # currently unused
 2213: (4)                with pytest.raises(ValueError,
 2214: (23)                                  match="RGBA values should be within 0-1 range"):
 2215: (8)                    c = np.array([[0.1, 0.2, 0.7], [0.2, 0.4, 1.4]])  # value > 1
 2216: (8)                    mpl.axes.Axes._parse_scatter_color_args(
 2217: (12)                       c, None, kwargs={}, xsize=2, get_next_color_func=get_next_color)
 2218: (0)            def test_as_mpl_axes_api():
 2219: (4)                class Polar:
 2220: (8)                    def __init__(self):
 2221: (12)                       self.theta_offset = 0
 2222: (8)                    def _as_mpl_axes(self):
 2223: (12)                       return PolarAxes, {'theta_offset': self.theta_offset}
 2224: (4)                prj = Polar()
 2225: (4)                prj2 = Polar()
 2226: (4)                prj2.theta_offset = np.pi
 2227: (4)                ax = plt.axes((0, 0, 1, 1), projection=prj)
 2228: (4)                assert type(ax) is PolarAxes
 2229: (4)                plt.close()
 2230: (4)                ax = plt.subplot(121, projection=prj)
 2231: (4)                assert type(ax) is PolarAxes
 2232: (4)                plt.close()
 2233: (0)            def test_pyplot_axes():
 2234: (4)                fig1, ax1 = plt.subplots()
 2235: (4)                fig2, ax2 = plt.subplots()
 2236: (4)                plt.sca(ax1)
 2237: (4)                assert ax1 is plt.gca()
 2238: (4)                assert fig1 is plt.gcf()
 2239: (4)                plt.close(fig1)
 2240: (4)                plt.close(fig2)
 2241: (0)            def test_log_scales():
```

```
(4)             fig, ax = plt.subplots()
(4)             ax.plot(np.log(np.linspace(0.1, 100)))
(4)             ax.set_yscale('log', base=5.5)
(4)             ax.invert_yaxis()
(4)             ax.set_xscale('log', base=9.0)
(4)             xticks, yticks = [
(8)                 [(t.get_loc(), t.label1.get_text()) for t in axis._update_ticks()]
(8)                 for axis in [ax.xaxis, ax.yaxis]
(4)             ]
(4)             assert xticks == [
(8)                 (1.0, '$\\mathdefault{9^{0}}$'),
(8)                 (9.0, '$\\mathdefault{9^{1}}$'),
(8)                 (81.0, '$\\mathdefault{9^{2}}$'),
(8)                 (2.0, ''),
(8)                 (3.0, ''),
(8)                 (4.0, ''),
(8)                 (5.0, ''),
(8)                 (6.0, ''),
(8)                 (7.0, ''),
(8)                 (8.0, ''),
(8)                 (18.0, ''),
(8)                 (27.0, ''),
(8)                 (36.0, ''),
(8)                 (45.0, ''),
(8)                 (54.0, ''),
(8)                 (63.0, ''),
(8)                 (72.0, ''),
(4)             ]
(4)             assert yticks == [
(8)                 (0.18181818181818182, '$\\mathdefault{5.5^{-1}}$'),
(8)                 (1.0, '$\\mathdefault{5.5^{0}}$'),
(8)                 (5.5, '$\\mathdefault{5.5^{1}}$'),
(8)                 (0.36363636363636365, ''),
(8)                 (0.5454545454545454, ''),
(8)                 (0.7272727272727273, ''),
(8)                 (0.9090909090909092, ''),
(8)                 (2.0, ''),
(8)                 (3.0, ''),
(8)                 (4.0, ''),
(8)                 (5.0, ''),
(4)             ]
(0)         def test_log_scales_no_data():
(4)             _, ax = plt.subplots()
(4)             ax.set(xscale="log", yscale="log")
(4)             ax.xaxis.set_major_locator(mticker.MultipleLocator(1))
(4)             assert ax.get_xlim() == ax.get_ylim() == (1, 10)
(0)         def test_log_scales_invalid():
(4)             fig, ax = plt.subplots()
(4)             ax.set_xscale('log')
(4)             with pytest.warns(UserWarning, match='Attempt to set non-positive'):
(8)                 ax.set_xlim(-1, 10)
(4)             ax.set_yscale('log')
(4)             with pytest.warns(UserWarning, match='Attempt to set non-positive'):
(8)                 ax.set_ylim(-1, 10)
(0)         @image_comparison(['stackplot_test_image', 'stackplot_test_image'],
(18)                          tol=0.031 if platform.machine() == 'arm64' else 0)
(0)         def test_stackplot():
(4)             fig = plt.figure()
(4)             x = np.linspace(0, 10, 10)
(4)             y1 = 1.0 * x
(4)             y2 = 2.0 * x + 1
(4)             y3 = 3.0 * x + 2
(4)             ax = fig.add_subplot(1, 1, 1)
(4)             ax.stackplot(x, y1, y2, y3)
(4)             ax.set_xlim((0, 10))
(4)             ax.set_ylim((0, 70))
(4)             data = {"x": x, "y1": y1, "y2": y2, "y3": y3}
(4)             fig, ax = plt.subplots()
(4)             ax.stackplot("x", "y1", "y2", "y3", data=data, colors=["C0", "C1", "C2"])
```

```
2311: (4)               ax.set_xlim((0, 10))
2312: (4)               ax.set_ylim((0, 70))
2313: (0)           @image_comparison(['stackplot_test_baseline'], remove_text=True)
2314: (0)           def test_stackplot_baseline():
2315: (4)               np.random.seed(0)
2316: (4)               def layers(n, m):
2317: (8)                   a = np.zeros((m, n))
2318: (8)                   for i in range(n):
2319: (12)                      for j in range(5):
2320: (16)                          x = 1 / (.1 + np.random.random())
2321: (16)                          y = 2 * np.random.random() - .5
2322: (16)                          z = 10 / (.1 + np.random.random())
2323: (16)                          a[:, i] += x * np.exp(-((np.arange(m) / m - y) * z) ** 2)
2324: (8)                   return a
2325: (4)               d = layers(3, 100)
2326: (4)               d[50, :] = 0  # test for fixed weighted wiggle (issue #6313)
2327: (4)               fig, axs = plt.subplots(2, 2)
2328: (4)               axs[0, 0].stackplot(range(100), d.T, baseline='zero')
2329: (4)               axs[0, 1].stackplot(range(100), d.T, baseline='sym')
2330: (4)               axs[1, 0].stackplot(range(100), d.T, baseline='wiggle')
2331: (4)               axs[1, 1].stackplot(range(100), d.T, baseline='weighted_wiggle')
2332: (0)           @check_figures_equal()
2333: (0)           def test_stackplot_hatching(fig_ref, fig_test):
2334: (4)               x = np.linspace(0, 10, 10)
2335: (4)               y1 = 1.0 * x
2336: (4)               y2 = 2.0 * x + 1
2337: (4)               y3 = 3.0 * x + 2
2338: (4)               ax_test = fig_test.subplots()
2339: (4)               ax_test.stackplot(x, y1, y2, y3, hatch=["x", "//", "\\\\"], colors=
["white"])
2340: (4)               ax_test.set_xlim((0, 10))
2341: (4)               ax_test.set_ylim((0, 70))
2342: (4)               stack_baseline = np.zeros(len(x))
2343: (4)               ax_ref = fig_ref.subplots()
2344: (4)               ax_ref.fill_between(x, stack_baseline, y1, hatch="x", facecolor="white")
2345: (4)               ax_ref.fill_between(x, y1, y1+y2, hatch="//", facecolor="white")
2346: (4)               ax_ref.fill_between(x, y1+y2, y1+y2+y3, hatch="\\\\", facecolor="white")
2347: (4)               ax_ref.set_xlim((0, 10))
2348: (4)               ax_ref.set_ylim((0, 70))
2349: (0)           def _bxp_test_helper(
2350: (8)                   stats_kwargs={}, transform_stats=lambda s: s, bxp_kwargs={}):
2351: (4)               np.random.seed(937)
2352: (4)               logstats = mpl.cbook.boxplot_stats(
2353: (8)                   np.random.lognormal(mean=1.25, sigma=1., size=(37, 4)),
**stats_kwargs)
2354: (4)               fig, ax = plt.subplots()
2355: (4)               if bxp_kwargs.get('vert', True):
2356: (8)                   ax.set_yscale('log')
2357: (4)               else:
2358: (8)                   ax.set_xscale('log')
2359: (4)               if not bxp_kwargs.get('patch_artist', False):
2360: (8)                   mpl.rcParams['boxplot.boxprops.linewidth'] = \
2361: (12)                      mpl.rcParams['lines.linewidth']
2362: (4)               ax.bxp(transform_stats(logstats), **bxp_kwargs)
2363: (0)           @image_comparison(['bxp_baseline.png'],
2364: (18)                            savefig_kwarg={'dpi': 40},
2365: (18)                            style='default')
2366: (0)           def test_bxp_baseline():
2367: (4)               _bxp_test_helper()
2368: (0)           @image_comparison(['bxp_rangewhis.png'],
2369: (18)                            savefig_kwarg={'dpi': 40},
2370: (18)                            style='default')
2371: (0)           def test_bxp_rangewhis():
2372: (4)               _bxp_test_helper(stats_kwargs=dict(whis=[0, 100]))
2373: (0)           @image_comparison(['bxp_percentilewhis.png'],
2374: (18)                            savefig_kwarg={'dpi': 40},
2375: (18)                            style='default')
2376: (0)           def test_bxp_percentilewhis():
2377: (4)               _bxp_test_helper(stats_kwargs=dict(whis=[5, 95]))
```

```
(0)           @image_comparison(['bxp_with_xlabels.png'],
(18)                            savefig_kwarg={'dpi': 40},
(18)                            style='default')
(0)           def test_bxp_with_xlabels():
(4)               def transform(stats):
(8)                   for s, label in zip(stats, list('ABCD')):
(12)                      s['label'] = label
(8)                   return stats
(4)               _bxp_test_helper(transform_stats=transform)
(0)           @image_comparison(['bxp_horizontal.png'],
(18)                            remove_text=True,
(18)                            savefig_kwarg={'dpi': 40},
(18)                            style='default',
(18)                            tol=0.1)
(0)           def test_bxp_horizontal():
(4)               _bxp_test_helper(bxp_kwargs=dict(vert=False))
(0)           @image_comparison(['bxp_with_ylabels.png'],
(18)                            savefig_kwarg={'dpi': 40},
(18)                            style='default',
(18)                            tol=0.1)
(0)           def test_bxp_with_ylabels():
(4)               def transform(stats):
(8)                   for s, label in zip(stats, list('ABCD')):
(12)                      s['label'] = label
(8)                   return stats
(4)               _bxp_test_helper(transform_stats=transform, bxp_kwargs=dict(vert=False))
(0)           @image_comparison(['bxp_patchartist.png'],
(18)                            remove_text=True,
(18)                            savefig_kwarg={'dpi': 40},
(18)                            style='default')
(0)           def test_bxp_patchartist():
(4)               _bxp_test_helper(bxp_kwargs=dict(patch_artist=True))
(0)           @image_comparison(['bxp_custompatchartist.png'],
(18)                            remove_text=True,
(18)                            savefig_kwarg={'dpi': 100},
(18)                            style='default')
(0)           def test_bxp_custompatchartist():
(4)               _bxp_test_helper(bxp_kwargs=dict(
(8)                   patch_artist=True,
(8)                   boxprops=dict(facecolor='yellow', edgecolor='green', ls=':')))
(0)           @image_comparison(['bxp_customoutlier.png'],
(18)                            remove_text=True,
(18)                            savefig_kwarg={'dpi': 40},
(18)                            style='default')
(0)           def test_bxp_customoutlier():
(4)               _bxp_test_helper(bxp_kwargs=dict(
(8)                   flierprops=dict(linestyle='none', marker='d', mfc='g')))
(0)           @image_comparison(['bxp_withmean_custompoint.png'],
(18)                            remove_text=True,
(18)                            savefig_kwarg={'dpi': 40},
(18)                            style='default')
(0)           def test_bxp_showcustommean():
(4)               _bxp_test_helper(bxp_kwargs=dict(
(8)                   showmeans=True,
(8)                   meanprops=dict(linestyle='none', marker='d', mfc='green'),
(4)               ))
(0)           @image_comparison(['bxp_custombox.png'],
(18)                            remove_text=True,
(18)                            savefig_kwarg={'dpi': 40},
(18)                            style='default')
(0)           def test_bxp_custombox():
(4)               _bxp_test_helper(bxp_kwargs=dict(
(8)                   boxprops=dict(linestyle='--', color='b', lw=3)))
(0)           @image_comparison(['bxp_custommedian.png'],
(18)                            remove_text=True,
(18)                            savefig_kwarg={'dpi': 40},
(18)                            style='default')
(0)           def test_bxp_custommedian():
(4)               _bxp_test_helper(bxp_kwargs=dict(
```

```
(8)                 medianprops=dict(linestyle='--', color='b', lw=3)))
(0)         @image_comparison(['bxp_customcap.png'],
(18)                          remove_text=True,
(18)                          savefig_kwarg={'dpi': 40},
(18)                          style='default')
(0)         def test_bxp_customcap():
(4)             _bxp_test_helper(bxp_kwargs=dict(
(8)                 capprops=dict(linestyle='--', color='g', lw=3)))
(0)         @image_comparison(['bxp_customwhisker.png'],
(18)                          remove_text=True,
(18)                          savefig_kwarg={'dpi': 40},
(18)                          style='default')
(0)         def test_bxp_customwhisker():
(4)             _bxp_test_helper(bxp_kwargs=dict(
(8)                 whiskerprops=dict(linestyle='-', color='m', lw=3)))
(0)         @check_figures_equal()
(0)         def test_boxplot_median_bound_by_box(fig_test, fig_ref):
(4)             data = np.arange(3)
(4)             medianprops_test = {"linewidth": 12}
(4)             medianprops_ref = {**medianprops_test, "solid_capstyle": "butt"}
(4)             fig_test.subplots().boxplot(data,  medianprops=medianprops_test)
(4)             fig_ref.subplots().boxplot(data, medianprops=medianprops_ref)
(0)         @image_comparison(['bxp_withnotch.png'],
(18)                          remove_text=True,
(18)                          savefig_kwarg={'dpi': 40},
(18)                          style='default')
(0)         def test_bxp_shownotches():
(4)             _bxp_test_helper(bxp_kwargs=dict(shownotches=True))
(0)         @image_comparison(['bxp_nocaps.png'],
(18)                          remove_text=True,
(18)                          savefig_kwarg={'dpi': 40},
(18)                          style='default')
(0)         def test_bxp_nocaps():
(4)             _bxp_test_helper(bxp_kwargs=dict(showcaps=False))
(0)         @image_comparison(['bxp_nobox.png'],
(18)                          remove_text=True,
(18)                          savefig_kwarg={'dpi': 40},
(18)                          style='default')
(0)         def test_bxp_nobox():
(4)             _bxp_test_helper(bxp_kwargs=dict(showbox=False))
(0)         @image_comparison(['bxp_no_flier_stats.png'],
(18)                          remove_text=True,
(18)                          savefig_kwarg={'dpi': 40},
(18)                          style='default')
(0)         def test_bxp_no_flier_stats():
(4)             def transform(stats):
(8)                 for s in stats:
(12)                    s.pop('fliers', None)
(8)                 return stats
(4)             _bxp_test_helper(transform_stats=transform,
(21)                             bxp_kwargs=dict(showfliers=False))
(0)         @image_comparison(['bxp_withmean_point.png'],
(18)                          remove_text=True,
(18)                          savefig_kwarg={'dpi': 40},
(18)                          style='default')
(0)         def test_bxp_showmean():
(4)             _bxp_test_helper(bxp_kwargs=dict(showmeans=True, meanline=False))
(0)         @image_comparison(['bxp_withmean_line.png'],
(18)                          remove_text=True,
(18)                          savefig_kwarg={'dpi': 40},
(18)                          style='default')
(0)         def test_bxp_showmeanasline():
(4)             _bxp_test_helper(bxp_kwargs=dict(showmeans=True, meanline=True))
(0)         @image_comparison(['bxp_scalarwidth.png'],
(18)                          remove_text=True,
(18)                          savefig_kwarg={'dpi': 40},
(18)                          style='default')
(0)         def test_bxp_scalarwidth():
(4)             _bxp_test_helper(bxp_kwargs=dict(widths=.25))
```

```
(0)           @image_comparison(['bxp_customwidths.png'],
(18)                            remove_text=True,
(18)                            savefig_kwarg={'dpi': 40},
(18)                            style='default')
(0)           def test_bxp_customwidths():
(4)               _bxp_test_helper(bxp_kwargs=dict(widths=[0.10, 0.25, 0.65, 0.85]))
(0)           @image_comparison(['bxp_custompositions.png'],
(18)                            remove_text=True,
(18)                            savefig_kwarg={'dpi': 40},
(18)                            style='default')
(0)           def test_bxp_custompositions():
(4)               _bxp_test_helper(bxp_kwargs=dict(positions=[1, 5, 6, 7]))
(0)           def test_bxp_bad_widths():
(4)               with pytest.raises(ValueError):
(8)                   _bxp_test_helper(bxp_kwargs=dict(widths=[1]))
(0)           def test_bxp_bad_positions():
(4)               with pytest.raises(ValueError):
(8)                   _bxp_test_helper(bxp_kwargs=dict(positions=[2, 3]))
(0)           @image_comparison(['bxp_custom_capwidths.png'],
(18)                            savefig_kwarg={'dpi': 40},
(18)                            style='default')
(0)           def test_bxp_custom_capwidths():
(4)               _bxp_test_helper(bxp_kwargs=dict(capwidths=[0.0, 0.1, 0.5, 1.0]))
(0)           @image_comparison(['bxp_custom_capwidth.png'],
(18)                            savefig_kwarg={'dpi': 40},
(18)                            style='default')
(0)           def test_bxp_custom_capwidth():
(4)               _bxp_test_helper(bxp_kwargs=dict(capwidths=0.6))
(0)           def test_bxp_bad_capwidths():
(4)               with pytest.raises(ValueError):
(8)                   _bxp_test_helper(bxp_kwargs=dict(capwidths=[1]))
(0)           @image_comparison(['boxplot', 'boxplot'], tol=1.28, style='default')
(0)           def test_boxplot():
(4)               np.random.seed(937)
(4)               x = np.linspace(-7, 7, 140)
(4)               x = np.hstack([-25, x, 25])
(4)               fig, ax = plt.subplots()
(4)               ax.boxplot([x, x], bootstrap=10000, notch=1)
(4)               ax.set_ylim((-30, 30))
(4)               data = {"x": [x, x]}
(4)               fig, ax = plt.subplots()
(4)               ax.boxplot("x", bootstrap=10000, notch=1, data=data)
(4)               ax.set_ylim((-30, 30))
(0)           @check_figures_equal(extensions=["png"])
(0)           def test_boxplot_masked(fig_test, fig_ref):
(4)               x_orig = np.linspace(-1, 1, 200)
(4)               ax = fig_test.subplots()
(4)               x = x_orig[x_orig >= 0]
(4)               ax.boxplot(x)
(4)               x = np.ma.masked_less(x_orig, 0)
(4)               ax = fig_ref.subplots()
(4)               ax.boxplot(x)
(0)           @image_comparison(['boxplot_custom_capwidths.png'],
(18)                            savefig_kwarg={'dpi': 40}, style='default')
(0)           def test_boxplot_custom_capwidths():
(4)               x = np.linspace(-7, 7, 140)
(4)               x = np.hstack([-25, x, 25])
(4)               fig, ax = plt.subplots()
(4)               ax.boxplot([x, x], notch=1, capwidths=[0.01, 0.2])
(0)           @image_comparison(['boxplot_sym2.png'], remove_text=True, style='default')
(0)           def test_boxplot_sym2():
(4)               np.random.seed(937)
(4)               x = np.linspace(-7, 7, 140)
(4)               x = np.hstack([-25, x, 25])
(4)               fig, [ax1, ax2] = plt.subplots(1, 2)
(4)               ax1.boxplot([x, x], bootstrap=10000, sym='^')
(4)               ax1.set_ylim((-30, 30))
(4)               ax2.boxplot([x, x], bootstrap=10000, sym='g')
(4)               ax2.set_ylim((-30, 30))
```

```
(0)  @image_comparison(['boxplot_sym.png'],
(18)                   remove_text=True,
(18)                   savefig_kwarg={'dpi': 40},
(18)                   style='default')
(0)  def test_boxplot_sym():
(4)      x = np.linspace(-7, 7, 140)
(4)      x = np.hstack([-25, x, 25])
(4)      fig, ax = plt.subplots()
(4)      ax.boxplot([x, x], sym='gs')
(4)      ax.set_ylim((-30, 30))
(0)  @image_comparison(['boxplot_autorange_false_whiskers.png',
(19)                    'boxplot_autorange_true_whiskers.png'],
(18)                   style='default')
(0)  def test_boxplot_autorange_whiskers():
(4)      np.random.seed(937)
(4)      x = np.ones(140)
(4)      x = np.hstack([0, x, 2])
(4)      fig1, ax1 = plt.subplots()
(4)      ax1.boxplot([x, x], bootstrap=10000, notch=1)
(4)      ax1.set_ylim((-5, 5))
(4)      fig2, ax2 = plt.subplots()
(4)      ax2.boxplot([x, x], bootstrap=10000, notch=1, autorange=True)
(4)      ax2.set_ylim((-5, 5))
(0)  def _rc_test_bxp_helper(ax, rc_dict):
(4)      x = np.linspace(-7, 7, 140)
(4)      x = np.hstack([-25, x, 25])
(4)      with matplotlib.rc_context(rc_dict):
(8)          ax.boxplot([x, x])
(4)      return ax
(0)  @image_comparison(['boxplot_rc_parameters'],
(18)                   savefig_kwarg={'dpi': 100}, remove_text=True,
(18)                   tol=1, style='default')
(0)  def test_boxplot_rc_parameters():
(4)      np.random.seed(937)
(4)      fig, ax = plt.subplots(3)
(4)      rc_axis0 = {
(8)          'boxplot.notch': True,
(8)          'boxplot.whiskers': [5, 95],
(8)          'boxplot.bootstrap': 10000,
(8)          'boxplot.flierprops.color': 'b',
(8)          'boxplot.flierprops.marker': 'o',
(8)          'boxplot.flierprops.markerfacecolor': 'g',
(8)          'boxplot.flierprops.markeredgecolor': 'b',
(8)          'boxplot.flierprops.markersize': 5,
(8)          'boxplot.flierprops.linestyle': '--',
(8)          'boxplot.flierprops.linewidth': 2.0,
(8)          'boxplot.boxprops.color': 'r',
(8)          'boxplot.boxprops.linewidth': 2.0,
(8)          'boxplot.boxprops.linestyle': '--',
(8)          'boxplot.capprops.color': 'c',
(8)          'boxplot.capprops.linewidth': 2.0,
(8)          'boxplot.capprops.linestyle': '--',
(8)          'boxplot.medianprops.color': 'k',
(8)          'boxplot.medianprops.linewidth': 2.0,
(8)          'boxplot.medianprops.linestyle': '--',
(4)      }
(4)      rc_axis1 = {
(8)          'boxplot.vertical': False,
(8)          'boxplot.whiskers': [0, 100],
(8)          'boxplot.patchartist': True,
(4)      }
(4)      rc_axis2 = {
(8)          'boxplot.whiskers': 2.0,
(8)          'boxplot.showcaps': False,
(8)          'boxplot.showbox': False,
(8)          'boxplot.showfliers': False,
(8)          'boxplot.showmeans': True,
(8)          'boxplot.meanline': True,
(8)          'boxplot.meanprops.color': 'c',
```

```
2654: (8)                    'boxplot.meanprops.linewidth': 2.0,
2655: (8)                    'boxplot.meanprops.linestyle': '--',
2656: (8)                    'boxplot.whiskerprops.color': 'r',
2657: (8)                    'boxplot.whiskerprops.linewidth': 2.0,
2658: (8)                    'boxplot.whiskerprops.linestyle': '-.',
2659: (4)                }
2660: (4)                dict_list = [rc_axis0, rc_axis1, rc_axis2]
2661: (4)                for axis, rc_axis in zip(ax, dict_list):
2662: (8)                    _rc_test_bxp_helper(axis, rc_axis)
2663: (4)                assert (matplotlib.patches.PathPatch in
2664: (12)                       [type(t) for t in ax[1].get_children()])
2665: (0)            @image_comparison(['boxplot_with_CIarray.png'],
2666: (18)                             remove_text=True, savefig_kwarg={'dpi': 40},
style='default')
2667: (0)            def test_boxplot_with_CIarray():
2668: (4)                np.random.seed(937)
2669: (4)                x = np.linspace(-7, 7, 140)
2670: (4)                x = np.hstack([-25, x, 25])
2671: (4)                fig, ax = plt.subplots()
2672: (4)                CIs = np.array([[-1.5, 3.], [-1., 3.5]])
2673: (4)                ax.boxplot([x, x], bootstrap=10000, usermedians=[None, 1.0],
2674: (15)                          conf_intervals=CIs, notch=1)
2675: (4)                ax.set_ylim((-30, 30))
2676: (0)            @image_comparison(['boxplot_no_inverted_whisker.png'],
2677: (18)                             remove_text=True, savefig_kwarg={'dpi': 40},
style='default')
2678: (0)            def test_boxplot_no_weird_whisker():
2679: (4)                x = np.array([3, 9000, 150, 88, 350, 200000, 1400, 960],
2680: (17)                            dtype=np.float64)
2681: (4)                ax1 = plt.axes()
2682: (4)                ax1.boxplot(x)
2683: (4)                ax1.set_yscale('log')
2684: (4)                ax1.yaxis.grid(False, which='minor')
2685: (4)                ax1.xaxis.grid(False)
2686: (0)            def test_boxplot_bad_medians():
2687: (4)                x = np.linspace(-7, 7, 140)
2688: (4)                x = np.hstack([-25, x, 25])
2689: (4)                fig, ax = plt.subplots()
2690: (4)                with pytest.raises(ValueError):
2691: (8)                    ax.boxplot(x, usermedians=[1, 2])
2692: (4)                with pytest.raises(ValueError):
2693: (8)                    ax.boxplot([x, x], usermedians=[[1, 2], [1, 2]])
2694: (0)            def test_boxplot_bad_ci():
2695: (4)                x = np.linspace(-7, 7, 140)
2696: (4)                x = np.hstack([-25, x, 25])
2697: (4)                fig, ax = plt.subplots()
2698: (4)                with pytest.raises(ValueError):
2699: (8)                    ax.boxplot([x, x], conf_intervals=[[1, 2]])
2700: (4)                with pytest.raises(ValueError):
2701: (8)                    ax.boxplot([x, x], conf_intervals=[[1, 2], [1]])
2702: (0)            def test_boxplot_zorder():
2703: (4)                x = np.arange(10)
2704: (4)                fix, ax = plt.subplots()
2705: (4)                assert ax.boxplot(x)['boxes'][0].get_zorder() == 2
2706: (4)                assert ax.boxplot(x, zorder=10)['boxes'][0].get_zorder() == 10
2707: (0)            def test_boxplot_marker_behavior():
2708: (4)                plt.rcParams['lines.marker'] = 's'
2709: (4)                plt.rcParams['boxplot.flierprops.marker'] = 'o'
2710: (4)                plt.rcParams['boxplot.meanprops.marker'] = '^'
2711: (4)                fig, ax = plt.subplots()
2712: (4)                test_data = np.arange(100)
2713: (4)                test_data[-1] = 150  # a flier point
2714: (4)                bxp_handle = ax.boxplot(test_data, showmeans=True)
2715: (4)                for bxp_lines in ['whiskers', 'caps', 'boxes', 'medians']:
2716: (8)                    for each_line in bxp_handle[bxp_lines]:
2717: (12)                       assert each_line.get_marker() == ''
2718: (4)                assert bxp_handle['fliers'][0].get_marker() == 'o'
2719: (4)                assert bxp_handle['means'][0].get_marker() == '^'
2720: (0)            @image_comparison(['boxplot_mod_artists_after_plotting.png'],
```

```
 (18)                            remove_text=True, savefig_kwarg={'dpi': 40},
style='default')
 (0)           def test_boxplot_mod_artist_after_plotting():
 (4)               x = [0.15, 0.11, 0.06, 0.06, 0.12, 0.56, -0.56]
 (4)               fig, ax = plt.subplots()
 (4)               bp = ax.boxplot(x, sym="o")
 (4)               for key in bp:
 (8)                   for obj in bp[key]:
 (12)                      obj.set_color('green')
 (0)           @image_comparison(['violinplot_vert_baseline.png',
 (19)                             'violinplot_vert_baseline.png'])
 (0)           def test_vert_violinplot_baseline():
 (4)               np.random.seed(414213562)
 (4)               data = [np.random.normal(size=100) for _ in range(4)]
 (4)               ax = plt.axes()
 (4)               ax.violinplot(data, positions=range(4), showmeans=False,
showextrema=False,
 (18)                            showmedians=False)
 (4)               data = {"d": data}
 (4)               fig, ax = plt.subplots()
 (4)               ax.violinplot("d", positions=range(4), showmeans=False, showextrema=False,
 (18)                            showmedians=False, data=data)
 (0)           @image_comparison(['violinplot_vert_showmeans.png'])
 (0)           def test_vert_violinplot_showmeans():
 (4)               ax = plt.axes()
 (4)               np.random.seed(732050807)
 (4)               data = [np.random.normal(size=100) for _ in range(4)]
 (4)               ax.violinplot(data, positions=range(4), showmeans=True, showextrema=False,
 (18)                            showmedians=False)
 (0)           @image_comparison(['violinplot_vert_showextrema.png'])
 (0)           def test_vert_violinplot_showextrema():
 (4)               ax = plt.axes()
 (4)               np.random.seed(236067977)
 (4)               data = [np.random.normal(size=100) for _ in range(4)]
 (4)               ax.violinplot(data, positions=range(4), showmeans=False, showextrema=True,
 (18)                            showmedians=False)
 (0)           @image_comparison(['violinplot_vert_showmedians.png'])
 (0)           def test_vert_violinplot_showmedians():
 (4)               ax = plt.axes()
 (4)               np.random.seed(645751311)
 (4)               data = [np.random.normal(size=100) for _ in range(4)]
 (4)               ax.violinplot(data, positions=range(4), showmeans=False,
showextrema=False,
 (18)                            showmedians=True)
 (0)           @image_comparison(['violinplot_vert_showall.png'])
 (0)           def test_vert_violinplot_showall():
 (4)               ax = plt.axes()
 (4)               np.random.seed(316624790)
 (4)               data = [np.random.normal(size=100) for _ in range(4)]
 (4)               ax.violinplot(data, positions=range(4), showmeans=True, showextrema=True,
 (18)                            showmedians=True,
 (18)                            quantiles=[[0.1, 0.9], [0.2, 0.8], [0.3, 0.7], [0.4, 0.6]])
 (0)           @image_comparison(['violinplot_vert_custompoints_10.png'])
 (0)           def test_vert_violinplot_custompoints_10():
 (4)               ax = plt.axes()
 (4)               np.random.seed(605551275)
 (4)               data = [np.random.normal(size=100) for _ in range(4)]
 (4)               ax.violinplot(data, positions=range(4), showmeans=False,
showextrema=False,
 (18)                            showmedians=False, points=10)
 (0)           @image_comparison(['violinplot_vert_custompoints_200.png'])
 (0)           def test_vert_violinplot_custompoints_200():
 (4)               ax = plt.axes()
 (4)               np.random.seed(123105625)
 (4)               data = [np.random.normal(size=100) for _ in range(4)]
 (4)               ax.violinplot(data, positions=range(4), showmeans=False,
showextrema=False,
 (18)                            showmedians=False, points=200)
 (0)           @image_comparison(['violinplot_horiz_baseline.png'])
```

```
(0)  def test_horiz_violinplot_baseline():
(4)      ax = plt.axes()
(4)      np.random.seed(358898943)
(4)      data = [np.random.normal(size=100) for _ in range(4)]
(4)      ax.violinplot(data, positions=range(4), vert=False, showmeans=False,
(18)                   showextrema=False, showmedians=False)
(0)  @image_comparison(['violinplot_horiz_showmedians.png'])
(0)  def test_horiz_violinplot_showmedians():
(4)      ax = plt.axes()
(4)      np.random.seed(795831523)
(4)      data = [np.random.normal(size=100) for _ in range(4)]
(4)      ax.violinplot(data, positions=range(4), vert=False, showmeans=False,
(18)                   showextrema=False, showmedians=True)
(0)  @image_comparison(['violinplot_horiz_showmeans.png'])
(0)  def test_horiz_violinplot_showmeans():
(4)      ax = plt.axes()
(4)      np.random.seed(385164807)
(4)      data = [np.random.normal(size=100) for _ in range(4)]
(4)      ax.violinplot(data, positions=range(4), vert=False, showmeans=True,
(18)                   showextrema=False, showmedians=False)
(0)  @image_comparison(['violinplot_horiz_showextrema.png'])
(0)  def test_horiz_violinplot_showextrema():
(4)      ax = plt.axes()
(4)      np.random.seed(567764362)
(4)      data = [np.random.normal(size=100) for _ in range(4)]
(4)      ax.violinplot(data, positions=range(4), vert=False, showmeans=False,
(18)                   showextrema=True, showmedians=False)
(0)  @image_comparison(['violinplot_horiz_showall.png'])
(0)  def test_horiz_violinplot_showall():
(4)      ax = plt.axes()
(4)      np.random.seed(82762530)
(4)      data = [np.random.normal(size=100) for _ in range(4)]
(4)      ax.violinplot(data, positions=range(4), vert=False, showmeans=True,
(18)                   showextrema=True, showmedians=True,
(18)                   quantiles=[[0.1, 0.9], [0.2, 0.8], [0.3, 0.7], [0.4, 0.6]])
(0)  @image_comparison(['violinplot_horiz_custompoints_10.png'])
(0)  def test_horiz_violinplot_custompoints_10():
(4)      ax = plt.axes()
(4)      np.random.seed(403124237)
(4)      data = [np.random.normal(size=100) for _ in range(4)]
(4)      ax.violinplot(data, positions=range(4), vert=False, showmeans=False,
(18)                   showextrema=False, showmedians=False, points=10)
(0)  @image_comparison(['violinplot_horiz_custompoints_200.png'])
(0)  def test_horiz_violinplot_custompoints_200():
(4)      ax = plt.axes()
(4)      np.random.seed(557438524)
(4)      data = [np.random.normal(size=100) for _ in range(4)]
(4)      ax.violinplot(data, positions=range(4), vert=False, showmeans=False,
(18)                   showextrema=False, showmedians=False, points=200)
(0)  @image_comparison(['violinplot_sides.png'], remove_text=True, style='mpl20')
(0)  def test_violinplot_sides():
(4)      ax = plt.axes()
(4)      np.random.seed(19680801)
(4)      data = [np.random.normal(size=100)]
(4)      for pos, side in zip([0, -0.5, 0.5], ['both', 'low', 'high']):
(8)          ax.violinplot(data, positions=[pos], vert=False, showmeans=False,
(22)                       showextrema=True, showmedians=True, side=side)
(4)      for pos, side in zip([4, 3.5, 4.5], ['both', 'low', 'high']):
(8)          ax.violinplot(data, positions=[pos], vert=True, showmeans=False,
(22)                       showextrema=True, showmedians=True, side=side)
(0)  def test_violinplot_bad_positions():
(4)      ax = plt.axes()
(4)      np.random.seed(855654600)
(4)      data = [np.random.normal(size=100) for _ in range(4)]
(4)      with pytest.raises(ValueError):
(8)          ax.violinplot(data, positions=range(5))
(0)  def test_violinplot_bad_widths():
(4)      ax = plt.axes()
(4)      np.random.seed(280109889)
```

```
2854: (4)                data = [np.random.normal(size=100) for _ in range(4)]
2855: (4)                with pytest.raises(ValueError):
2856: (8)                    ax.violinplot(data, positions=range(4), widths=[1, 2, 3])
2857: (0)            def test_violinplot_bad_quantiles():
2858: (4)                ax = plt.axes()
2859: (4)                np.random.seed(544003745)
2860: (4)                data = [np.random.normal(size=100)]
2861: (4)                with pytest.raises(ValueError):
2862: (8)                    ax.violinplot(data, quantiles=[[0.1, 0.2], [0.5, 0.7]])
2863: (0)            def test_violinplot_outofrange_quantiles():
2864: (4)                ax = plt.axes()
2865: (4)                np.random.seed(888194417)
2866: (4)                data = [np.random.normal(size=100)]
2867: (4)                with pytest.raises(ValueError):
2868: (8)                    ax.violinplot(data, quantiles=[[0.1, 0.2, 0.3, 1.05]])
2869: (4)                with pytest.raises(ValueError):
2870: (8)                    ax.violinplot(data, quantiles=[[-0.05, 0.2, 0.3, 0.75]])
2871: (0)            @check_figures_equal(extensions=["png"])
2872: (0)            def test_violinplot_single_list_quantiles(fig_test, fig_ref):
2873: (4)                np.random.seed(110433579)
2874: (4)                data = [np.random.normal(size=100)]
2875: (4)                ax = fig_test.subplots()
2876: (4)                ax.violinplot(data, quantiles=[0.1, 0.3, 0.9])
2877: (4)                ax = fig_ref.subplots()
2878: (4)                ax.violinplot(data, quantiles=[[0.1, 0.3, 0.9]])
2879: (0)            @check_figures_equal(extensions=["png"])
2880: (0)            def test_violinplot_pandas_series(fig_test, fig_ref, pd):
2881: (4)                np.random.seed(110433579)
2882: (4)                s1 = pd.Series(np.random.normal(size=7), index=[9, 8, 7, 6, 5, 4, 3])
2883: (4)                s2 = pd.Series(np.random.normal(size=9), index=list('ABCDEFGHI'))
2884: (4)                s3 = pd.Series(np.random.normal(size=11))
2885: (4)                fig_test.subplots().violinplot([s1, s2, s3])
2886: (4)                fig_ref.subplots().violinplot([s1.values, s2.values, s3.values])
2887: (0)            def test_manage_xticks():
2888: (4)                _, ax = plt.subplots()
2889: (4)                ax.set_xlim(0, 4)
2890: (4)                old_xlim = ax.get_xlim()
2891: (4)                np.random.seed(0)
2892: (4)                y1 = np.random.normal(10, 3, 20)
2893: (4)                y2 = np.random.normal(3, 1, 20)
2894: (4)                ax.boxplot([y1, y2], positions=[1, 2], manage_ticks=False)
2895: (4)                new_xlim = ax.get_xlim()
2896: (4)                assert_array_equal(old_xlim, new_xlim)
2897: (0)            def test_boxplot_not_single():
2898: (4)                fig, ax = plt.subplots()
2899: (4)                ax.boxplot(np.random.rand(100), positions=[3])
2900: (4)                ax.boxplot(np.random.rand(100), positions=[5])
2901: (4)                fig.canvas.draw()
2902: (4)                assert ax.get_xlim() == (2.5, 5.5)
2903: (4)                assert list(ax.get_xticks()) == [3, 5]
2904: (4)                assert [t.get_text() for t in ax.get_xticklabels()] == ["3", "5"]
2905: (0)            def test_tick_space_size_0():
2906: (4)                plt.plot([0, 1], [0, 1])
2907: (4)                matplotlib.rcParams.update({'font.size': 0})
2908: (4)                b = io.BytesIO()
2909: (4)                plt.savefig(b, dpi=80, format='raw')
2910: (0)            @image_comparison(['errorbar_basic', 'errorbar_mixed', 'errorbar_basic'])
2911: (0)            def test_errorbar():
2912: (4)                x = np.arange(0.1, 4, 0.5, dtype=np.longdouble)
2913: (4)                y = np.exp(-x)
2914: (4)                yerr = 0.1 + 0.2*np.sqrt(x)
2915: (4)                xerr = 0.1 + yerr
2916: (4)                fig = plt.figure()
2917: (4)                ax = fig.gca()
2918: (4)                ax.errorbar(x, y, xerr=0.2, yerr=0.4)
2919: (4)                ax.set_title("Simplest errorbars, 0.2 in x, 0.4 in y")
2920: (4)                fig, axs = plt.subplots(nrows=2, ncols=2, sharex=True)
2921: (4)                ax = axs[0, 0]
2922: (4)                ax.errorbar(x, y, yerr=yerr, fmt='o')
```

```
(4)             ax.set_title('Vert. symmetric')
(4)             ax.locator_params(nbins=4)
(4)             ax = axs[0, 1]
(4)             ax.errorbar(x, y, xerr=xerr, fmt='o', alpha=0.4)
(4)             ax.set_title('Hor. symmetric w/ alpha')
(4)             ax = axs[1, 0]
(4)             ax.errorbar(x, y, yerr=[yerr, 2*yerr], xerr=[xerr, 2*xerr], fmt='--o')
(4)             ax.set_title('H, V asymmetric')
(4)             ax = axs[1, 1]
(4)             ax.set_yscale('log')
(4)             ylower = np.maximum(1e-2, y - yerr)
(4)             yerr_lower = y - ylower
(4)             ax.errorbar(x, y, yerr=[yerr_lower, 2*yerr], xerr=xerr,
(16)                        fmt='o', ecolor='g', capthick=2)
(4)             ax.set_title('Mixed sym., log y')
(4)             ax.set_ylim(1e-2, 1e1)
(4)             fig.suptitle('Variable errorbars')
(4)             data = {"x": x, "y": y}
(4)             fig = plt.figure()
(4)             ax = fig.gca()
(4)             ax.errorbar("x", "y", xerr=0.2, yerr=0.4, data=data)
(4)             ax.set_title("Simplest errorbars, 0.2 in x, 0.4 in y")
(0)         @image_comparison(['mixed_errorbar_polar_caps'], extensions=['png'],
(18)                          remove_text=True)
(0)         def test_mixed_errorbar_polar_caps():
(4)             """
(4)             Mix several polar errorbar use cases in a single test figure.
(4)             It is advisable to position individual points off the grid. If there are
(4)             problems with reproducibility of this test, consider removing grid.
(4)             """
(4)             fig = plt.figure()
(4)             ax = plt.subplot(111, projection='polar')
(4)             th_sym = [1, 2, 3]
(4)             r_sym = [0.9]*3
(4)             ax.errorbar(th_sym, r_sym, xerr=0.35, yerr=0.2, fmt="o")
(4)             th_long = [np.pi/2 + .1, np.pi + .1]
(4)             r_long = [1.8, 2.2]
(4)             ax.errorbar(th_long, r_long, xerr=0.8 * np.pi, yerr=0.15, fmt="o")
(4)             th_asym = [4*np.pi/3 + .1, 5*np.pi/3 + .1, 2*np.pi-0.1]
(4)             r_asym = [1.1]*3
(4)             xerr = [[.3, .3, .2], [.2, .3, .3]]
(4)             yerr = [[.35, .5, .5], [.5, .35, .5]]
(4)             ax.errorbar(th_asym, r_asym, xerr=xerr, yerr=yerr, fmt="o")
(4)             th_over = [2.1]
(4)             r_over = [3.1]
(4)             ax.errorbar(th_over, r_over, xerr=10, yerr=.2, fmt="o")
(0)         def test_errorbar_colorcycle():
(4)             f, ax = plt.subplots()
(4)             x = np.arange(10)
(4)             y = 2*x
(4)             e1, _, _ = ax.errorbar(x, y, c=None)
(4)             e2, _, _ = ax.errorbar(x, 2*y, c=None)
(4)             ln1, = ax.plot(x, 4*y)
(4)             assert mcolors.to_rgba(e1.get_color()) == mcolors.to_rgba('C0')
(4)             assert mcolors.to_rgba(e2.get_color()) == mcolors.to_rgba('C1')
(4)             assert mcolors.to_rgba(ln1.get_color()) == mcolors.to_rgba('C2')
(0)         @check_figures_equal()
(0)         def test_errorbar_cycle_ecolor(fig_test, fig_ref):
(4)             x = np.arange(0.1, 4, 0.5)
(4)             y = [np.exp(-x+n) for n in range(4)]
(4)             axt = fig_test.subplots()
(4)             axr = fig_ref.subplots()
(4)             for yi, color in zip(y, ['C0', 'C1', 'C2', 'C3']):
(8)                 axt.errorbar(x, yi, yerr=(yi * 0.25), linestyle='-',
(21)                             marker='o', ecolor='black')
(8)                 axr.errorbar(x, yi, yerr=(yi * 0.25), linestyle='-',
(21)                             marker='o', color=color, ecolor='black')
(0)         def test_errorbar_shape():
(4)             fig = plt.figure()
```

```
(4)                 ax = fig.gca()
(4)                 x = np.arange(0.1, 4, 0.5)
(4)                 y = np.exp(-x)
(4)                 yerr1 = 0.1 + 0.2*np.sqrt(x)
(4)                 yerr = np.vstack((yerr1, 2*yerr1)).T
(4)                 xerr = 0.1 + yerr
(4)                 with pytest.raises(ValueError):
(8)                     ax.errorbar(x, y, yerr=yerr, fmt='o')
(4)                 with pytest.raises(ValueError):
(8)                     ax.errorbar(x, y, xerr=xerr, fmt='o')
(4)                 with pytest.raises(ValueError):
(8)                     ax.errorbar(x, y, yerr=yerr, xerr=xerr, fmt='o')
(0)             @image_comparison(['errorbar_limits'])
(0)             def test_errorbar_limits():
(4)                 x = np.arange(0.5, 5.5, 0.5)
(4)                 y = np.exp(-x)
(4)                 xerr = 0.1
(4)                 yerr = 0.2
(4)                 ls = 'dotted'
(4)                 fig, ax = plt.subplots()
(4)                 ax.errorbar(x, y, xerr=xerr, yerr=yerr, ls=ls, color='blue')
(4)                 uplims = np.zeros_like(x)
(4)                 uplims[[1, 5, 9]] = True
(4)                 ax.errorbar(x, y+0.5, xerr=xerr, yerr=yerr, uplims=uplims, ls=ls,
(16)                            color='green')
(4)                 lolims = np.zeros_like(x)
(4)                 lolims[[2, 4, 8]] = True
(4)                 ax.errorbar(x, y+1.0, xerr=xerr, yerr=yerr, lolims=lolims, ls=ls,
(16)                            color='red')
(4)                 ax.errorbar(x, y+1.5, marker='o', ms=8, xerr=xerr, yerr=yerr,
(16)                            lolims=lolims, uplims=uplims, ls=ls, color='magenta')
(4)                 xerr = 0.2
(4)                 yerr = np.full_like(x, 0.2)
(4)                 yerr[[3, 6]] = 0.3
(4)                 xlolims = lolims
(4)                 xuplims = uplims
(4)                 lolims = np.zeros_like(x)
(4)                 uplims = np.zeros_like(x)
(4)                 lolims[[6]] = True
(4)                 uplims[[3]] = True
(4)                 ax.errorbar(x, y+2.1, marker='o', ms=8, xerr=xerr, yerr=yerr,
(16)                            xlolims=xlolims, xuplims=xuplims, uplims=uplims,
(16)                            lolims=lolims, ls='none', mec='blue', capsize=0,
(16)                            color='cyan')
(4)                 ax.set_xlim((0, 5.5))
(4)                 ax.set_title('Errorbar upper and lower limits')
(0)             def test_errorbar_nonefmt():
(4)                 x = np.arange(5)
(4)                 y = np.arange(5)
(4)                 plotline, _, barlines = plt.errorbar(x, y, xerr=1, yerr=1, fmt='none')
(4)                 assert plotline is None
(4)                 for errbar in barlines:
(8)                     assert np.all(errbar.get_color() == mcolors.to_rgba('C0'))
(0)             def test_errorbar_line_specific_kwargs():
(4)                 x = np.arange(5)
(4)                 y = np.arange(5)
(4)                 plotline, _, _ = plt.errorbar(x, y, xerr=1, yerr=1, ls='None',
(34)                                              marker='s', fillstyle='full',
(34)                                              drawstyle='steps-mid',
(34)                                              dash_capstyle='round',
(34)                                              dash_joinstyle='miter',
(34)                                              solid_capstyle='butt',
(34)                                              solid_joinstyle='bevel')
(4)                 assert plotline.get_fillstyle() == 'full'
(4)                 assert plotline.get_drawstyle() == 'steps-mid'
(0)             @check_figures_equal(extensions=['png'])
(0)             def test_errorbar_with_prop_cycle(fig_test, fig_ref):
(4)                 ax = fig_ref.subplots()
(4)                 ax.errorbar(x=[2, 4, 10], y=[0, 1, 2], yerr=0.5,
```

```
(16)                            ls='--', marker='s', mfc='k')
(4)                 ax.errorbar(x=[2, 4, 10], y=[2, 3, 4], yerr=0.5, color='tab:green',
(16)                            ls=':', marker='s', mfc='y')
(4)                 ax.errorbar(x=[2, 4, 10], y=[4, 5, 6], yerr=0.5, fmt='tab:blue',
(16)                            ls='-.', marker='o', mfc='c')
(4)                 ax.set_xlim(1, 11)
(4)                 _cycle = cycler(ls=['--', ':', '-.'], marker=['s', 's', 'o'],
(20)                                mfc=['k', 'y', 'c'], color=['b', 'g', 'r'])
(4)                 plt.rc("axes", prop_cycle=_cycle)
(4)                 ax = fig_test.subplots()
(4)                 ax.errorbar(x=[2, 4, 10], y=[0, 1, 2], yerr=0.5)
(4)                 ax.errorbar(x=[2, 4, 10], y=[2, 3, 4], yerr=0.5, color='tab:green')
(4)                 ax.errorbar(x=[2, 4, 10], y=[4, 5, 6], yerr=0.5, fmt='tab:blue')
(4)                 ax.set_xlim(1, 11)
(0)             def test_errorbar_every_invalid():
(4)                 x = np.linspace(0, 1, 15)
(4)                 y = x * (1-x)
(4)                 yerr = y/6
(4)                 ax = plt.figure().subplots()
(4)                 with pytest.raises(ValueError, match='not a tuple of two integers'):
(8)                     ax.errorbar(x, y, yerr, errorevery=(1, 2, 3))
(4)                 with pytest.raises(ValueError, match='not a tuple of two integers'):
(8)                     ax.errorbar(x, y, yerr, errorevery=(1.3, 3))
(4)                 with pytest.raises(ValueError, match='not a valid NumPy fancy index'):
(8)                     ax.errorbar(x, y, yerr, errorevery=[False, True])
(4)                 with pytest.raises(ValueError, match='not a recognized value'):
(8)                     ax.errorbar(x, y, yerr, errorevery='foobar')
(0)             def test_xerr_yerr_not_negative():
(4)                 ax = plt.figure().subplots()
(4)                 with pytest.raises(ValueError,
(23)                                   match="'xerr' must not contain negative values"):
(8)                     ax.errorbar(x=[0], y=[0], xerr=[[-0.5], [1]], yerr=[[-0.5], [1]])
(4)                 with pytest.raises(ValueError,
(23)                                   match="'xerr' must not contain negative values"):
(8)                     ax.errorbar(x=[0], y=[0], xerr=[[-0.5], [1]])
(4)                 with pytest.raises(ValueError,
(23)                                   match="'yerr' must not contain negative values"):
(8)                     ax.errorbar(x=[0], y=[0], yerr=[[-0.5], [1]])
(4)                 with pytest.raises(ValueError,
(23)                                   match="'yerr' must not contain negative values"):
(8)                     x = np.arange(5)
(8)                     y = [datetime.datetime(2021, 9, i * 2 + 1) for i in x]
(8)                     ax.errorbar(x=x,
(20)                                y=y,
(20)                                yerr=datetime.timedelta(days=-10))
(0)             def test_xerr_yerr_not_none():
(4)                 ax = plt.figure().subplots()
(4)                 with pytest.raises(ValueError,
(23)                                   match="'xerr' must not contain None"):
(8)                     ax.errorbar(x=[0], y=[0], xerr=[[None], [1]], yerr=[[None], [1]])
(4)                 with pytest.raises(ValueError,
(23)                                   match="'xerr' must not contain None"):
(8)                     ax.errorbar(x=[0], y=[0], xerr=[[None], [1]])
(4)                 with pytest.raises(ValueError,
(23)                                   match="'yerr' must not contain None"):
(8)                     ax.errorbar(x=[0], y=[0], yerr=[[None], [1]])
(0)             @check_figures_equal()
(0)             def test_errorbar_every(fig_test, fig_ref):
(4)                 x = np.linspace(0, 1, 15)
(4)                 y = x * (1-x)
(4)                 yerr = y/6
(4)                 ax_ref = fig_ref.subplots()
(4)                 ax_test = fig_test.subplots()
(4)                 for color, shift in zip('rgbk', [0, 0, 2, 7]):
(8)                     y += .02
(8)                     ax_test.errorbar(x, y, yerr, errorevery=(shift, 4),
(25)                                     capsize=4, c=color)
(8)                     ax_ref.plot(x, y, c=color, zorder=2.1)
(8)                     ax_ref.errorbar(x[shift::4], y[shift::4], yerr[shift::4],
```

```
 3130: (24)                                capsize=4, c=color, fmt='none')
 3131: (4)                ax_test.errorbar(x, y + 0.1, yerr, markevery=(1, 4), capsize=4, fmt='o')
 3132: (4)                ax_ref.plot(x[1::4], y[1::4] + 0.1, 'o', zorder=2.1)
 3133: (4)                ax_ref.errorbar(x, y + 0.1, yerr, capsize=4, fmt='none')
 3134: (4)                ax_test.errorbar(x, y + 0.2, yerr, errorevery=slice(2, None, 3),
 3135: (21)                                 markevery=slice(2, None, 3),
 3136: (21)                                 capsize=4, c='C0', fmt='o')
 3137: (4)                ax_ref.plot(x[2::3], y[2::3] + 0.2, 'o', c='C0', zorder=2.1)
 3138: (4)                ax_ref.errorbar(x[2::3], y[2::3] + 0.2, yerr[2::3],
 3139: (20)                               capsize=4, c='C0', fmt='none')
 3140: (4)                ax_test.errorbar(x, y + 0.2, yerr, errorevery=[False, True, False] * 5,
 3141: (21)                                 markevery=[False, True, False] * 5,
 3142: (21)                                 capsize=4, c='C1', fmt='o')
 3143: (4)                ax_ref.plot(x[1::3], y[1::3] + 0.2, 'o', c='C1', zorder=2.1)
 3144: (4)                ax_ref.errorbar(x[1::3], y[1::3] + 0.2, yerr[1::3],
 3145: (20)                               capsize=4, c='C1', fmt='none')
 3146: (0)            @pytest.mark.parametrize('elinewidth', [[1, 2, 3],
 3147: (40)                                                   np.array([1, 2, 3]),
 3148: (40)                                                   1])
 3149: (0)            def test_errorbar_linewidth_type(elinewidth):
 3150: (4)                plt.errorbar([1, 2, 3], [1, 2, 3], yerr=[1, 2, 3], elinewidth=elinewidth)
 3151: (0)            @check_figures_equal(extensions=["png"])
 3152: (0)            def test_errorbar_nan(fig_test, fig_ref):
 3153: (4)                ax = fig_test.add_subplot()
 3154: (4)                xs = range(5)
 3155: (4)                ys = np.array([1, 2, np.nan, np.nan, 3])
 3156: (4)                es = np.array([4, 5, np.nan, np.nan, 6])
 3157: (4)                ax.errorbar(xs, ys, es)
 3158: (4)                ax = fig_ref.add_subplot()
 3159: (4)                ax.errorbar([0, 1], [1, 2], [4, 5])
 3160: (4)                ax.errorbar([4], [3], [6], fmt="C0")
 3161: (0)            @image_comparison(['hist_stacked_stepfilled', 'hist_stacked_stepfilled'])
 3162: (0)            def test_hist_stacked_stepfilled():
 3163: (4)                d1 = np.linspace(1, 3, 20)
 3164: (4)                d2 = np.linspace(0, 10, 50)
 3165: (4)                fig, ax = plt.subplots()
 3166: (4)                ax.hist((d1, d2), histtype="stepfilled", stacked=True)
 3167: (4)                data = {"x": (d1, d2)}
 3168: (4)                fig, ax = plt.subplots()
 3169: (4)                ax.hist("x", histtype="stepfilled", stacked=True, data=data)
 3170: (0)            @image_comparison(['hist_offset'])
 3171: (0)            def test_hist_offset():
 3172: (4)                d1 = np.linspace(0, 10, 50)
 3173: (4)                d2 = np.linspace(1, 3, 20)
 3174: (4)                fig, ax = plt.subplots()
 3175: (4)                ax.hist(d1, bottom=5)
 3176: (4)                ax.hist(d2, bottom=15)
 3177: (0)            @image_comparison(['hist_step.png'], remove_text=True)
 3178: (0)            def test_hist_step():
 3179: (4)                d1 = np.linspace(1, 3, 20)
 3180: (4)                fig, ax = plt.subplots()
 3181: (4)                ax.hist(d1, histtype="step")
 3182: (4)                ax.set_ylim(0, 10)
 3183: (4)                ax.set_xlim(-1, 5)
 3184: (0)            @image_comparison(['hist_step_horiz.png'])
 3185: (0)            def test_hist_step_horiz():
 3186: (4)                d1 = np.linspace(0, 10, 50)
 3187: (4)                d2 = np.linspace(1, 3, 20)
 3188: (4)                fig, ax = plt.subplots()
 3189: (4)                ax.hist((d1, d2), histtype="step", orientation="horizontal")
 3190: (0)            @image_comparison(['hist_stacked_weights'])
 3191: (0)            def test_hist_stacked_weighted():
 3192: (4)                d1 = np.linspace(0, 10, 50)
 3193: (4)                d2 = np.linspace(1, 3, 20)
 3194: (4)                w1 = np.linspace(0.01, 3.5, 50)
 3195: (4)                w2 = np.linspace(0.05, 2., 20)
 3196: (4)                fig, ax = plt.subplots()
 3197: (4)                ax.hist((d1, d2), weights=(w1, w2), histtype="stepfilled", stacked=True)
 3198: (0)            @image_comparison(['stem.png'], style='mpl20', remove_text=True)
```

```
(0)          def test_stem():
(4)              x = np.linspace(0.1, 2 * np.pi, 100)
(4)              fig, ax = plt.subplots()
(4)              ax.stem(x, np.cos(x),
(12)                     linefmt='C2-.', markerfmt='k+', basefmt='C1-.', label=' ')
(4)              ax.legend()
(0)          def test_stem_args():
(4)              """Test that stem() correctly identifies x and y values."""
(4)              def _assert_equal(stem_container, expected):
(8)                  x, y = map(list, stem_container.markerline.get_data())
(8)                  assert x == expected[0]
(8)                  assert y == expected[1]
(4)              fig, ax = plt.subplots()
(4)              x = [1, 3, 5]
(4)              y = [9, 8, 7]
(4)              _assert_equal(ax.stem(y), expected=([0, 1, 2], y))
(4)              _assert_equal(ax.stem(x, y), expected=(x, y))
(4)              _assert_equal(ax.stem(x, y, linefmt='r--'), expected=(x, y))
(4)              _assert_equal(ax.stem(x, y, 'r--'), expected=(x, y))
(4)              _assert_equal(ax.stem(x, y, linefmt='r--', basefmt='b--'), expected=(x,
y))
(4)              _assert_equal(ax.stem(y, linefmt='r--'), expected=([0, 1, 2], y))
(4)              _assert_equal(ax.stem(y, 'r--'), expected=([0, 1, 2], y))
(0)          def test_stem_markerfmt():
(4)              """Test that stem(..., markerfmt=...) produces the intended markers."""
(4)              def _assert_equal(stem_container, linecolor=None, markercolor=None,
(22)                               marker=None):
(8)                  """
(8)                  Check that the given StemContainer has the properties listed as
(8)                  keyword-arguments.
(8)                  """
(8)                  if linecolor is not None:
(12)                     assert mcolors.same_color(
(16)                         stem_container.stemlines.get_color(),
(16)                         linecolor)
(8)                  if markercolor is not None:
(12)                     assert mcolors.same_color(
(16)                         stem_container.markerline.get_color(),
(16)                         markercolor)
(8)                  if marker is not None:
(12)                     assert stem_container.markerline.get_marker() == marker
(8)                  assert stem_container.markerline.get_linestyle() == 'None'
(4)              fig, ax = plt.subplots()
(4)              x = [1, 3, 5]
(4)              y = [9, 8, 7]
(4)              _assert_equal(ax.stem(x, y), markercolor='C0', marker='o')
(4)              _assert_equal(ax.stem(x, y, markerfmt='x'), markercolor='C0', marker='x')
(4)              _assert_equal(ax.stem(x, y, markerfmt='rx'), markercolor='r', marker='x')
(4)              _assert_equal(
(8)                  ax.stem(x, y, 'r'),  # marker color follows linefmt if not given
(8)                  linecolor='r', markercolor='r', marker='o')
(4)              _assert_equal(
(8)                  ax.stem(x, y, 'rx'),  # the marker is currently not taken from linefmt
(8)                  linecolor='r', markercolor='r', marker='o')
(4)              _assert_equal(
(8)                  ax.stem(x, y, 'r', markerfmt='x'),  # only marker type specified
(8)                  linecolor='r', markercolor='r', marker='x')
(4)              _assert_equal(
(8)                  ax.stem(x, y, 'r', markerfmt='g'),  # only marker color specified
(8)                  linecolor='r', markercolor='g', marker='o')
(4)              _assert_equal(
(8)                  ax.stem(x, y, 'r', markerfmt='gx'),  # marker type and color specified
(8)                  linecolor='r', markercolor='g', marker='x')
(4)              _assert_equal(
(8)                  ax.stem(x, y, 'r', markerfmt=' '),  # markerfmt=' ' for no marker
(8)                  linecolor='r', markercolor='r', marker='None')
(4)              _assert_equal(
(8)                  ax.stem(x, y, 'r', markerfmt=''),  # markerfmt='' for no marker
(8)                  linecolor='r', markercolor='r', marker='None')
```

```
(4)                 _assert_equal(
(8)                     ax.stem(x, y, linefmt='r'),
(8)                     linecolor='r', markercolor='r', marker='o')
(4)                 _assert_equal(
(8)                     ax.stem(x, y, linefmt='r', markerfmt='x'),
(8)                     linecolor='r', markercolor='r', marker='x')
(4)                 _assert_equal(
(8)                     ax.stem(x, y, linefmt='r', markerfmt='gx'),
(8)                     linecolor='r', markercolor='g', marker='x')
(0)             def test_stem_dates():
(4)                 fig, ax = plt.subplots(1, 1)
(4)                 xs = [dateutil.parser.parse("2013-9-28 11:00:00"),
(10)                      dateutil.parser.parse("2013-9-28 12:00:00")]
(4)                 ys = [100, 200]
(4)                 ax.stem(xs, ys)
(0)             @image_comparison(['stem_orientation.png'], style='mpl20', remove_text=True)
(0)             def test_stem_orientation():
(4)                 x = np.linspace(0.1, 2*np.pi, 50)
(4)                 fig, ax = plt.subplots()
(4)                 ax.stem(x, np.cos(x),
(12)                        linefmt='C2-.', markerfmt='kx', basefmt='C1-.',
(12)                        orientation='horizontal')
(0)             @image_comparison(['hist_stacked_stepfilled_alpha'])
(0)             def test_hist_stacked_stepfilled_alpha():
(4)                 d1 = np.linspace(1, 3, 20)
(4)                 d2 = np.linspace(0, 10, 50)
(4)                 fig, ax = plt.subplots()
(4)                 ax.hist((d1, d2), histtype="stepfilled", stacked=True, alpha=0.5)
(0)             @image_comparison(['hist_stacked_step'])
(0)             def test_hist_stacked_step():
(4)                 d1 = np.linspace(1, 3, 20)
(4)                 d2 = np.linspace(0, 10, 50)
(4)                 fig, ax = plt.subplots()
(4)                 ax.hist((d1, d2), histtype="step", stacked=True)
(0)             @image_comparison(['hist_stacked_normed'])
(0)             def test_hist_stacked_density():
(4)                 d1 = np.linspace(1, 3, 20)
(4)                 d2 = np.linspace(0, 10, 50)
(4)                 fig, ax = plt.subplots()
(4)                 ax.hist((d1, d2), stacked=True, density=True)
(0)             @image_comparison(['hist_step_bottom.png'], remove_text=True)
(0)             def test_hist_step_bottom():
(4)                 d1 = np.linspace(1, 3, 20)
(4)                 fig, ax = plt.subplots()
(4)                 ax.hist(d1, bottom=np.arange(10), histtype="stepfilled")
(0)             def test_hist_step_geometry():
(4)                 bins = [0, 1, 2, 3]
(4)                 data = [0, 0, 1, 1, 1, 2]
(4)                 top = [[0, 0], [0, 2], [1, 2], [1, 3], [2, 3], [2, 1], [3, 1], [3, 0]]
(4)                 bottom = [[2, 0], [2, 0], [1, 0], [1, 0], [0, 0]]
(4)                 for histtype, xy in [('step', top), ('stepfilled', top + bottom)]:
(8)                     _, _, (polygon, ) = plt.hist(data, bins=bins, histtype=histtype)
(8)                     assert_array_equal(polygon.get_xy(), xy)
(0)             def test_hist_step_bottom_geometry():
(4)                 bins = [0, 1, 2, 3]
(4)                 data = [0, 0, 1, 1, 1, 2]
(4)                 top = [[0, 1], [0, 3], [1, 3], [1, 5], [2, 5], [2, 2.5], [3, 2.5], [3,
1.5]]
(4)                 bottom = [[2, 1.5], [2, 2], [1, 2], [1, 1], [0, 1]]
(4)                 for histtype, xy in [('step', top), ('stepfilled', top + bottom)]:
(8)                     _, _, (polygon, ) = plt.hist(data, bins=bins, bottom=[1, 2, 1.5],
(37)                                                  histtype=histtype)
(8)                     assert_array_equal(polygon.get_xy(), xy)
(0)             def test_hist_stacked_step_geometry():
(4)                 bins = [0, 1, 2, 3]
(4)                 data_1 = [0, 0, 1, 1, 1, 2]
(4)                 data_2 = [0, 1, 2]
(4)                 tops = [
(8)                     [[0, 0], [0, 2], [1, 2], [1, 3], [2, 3], [2, 1], [3, 1], [3, 0]],
```

```
3335: (8)                   [[0, 2], [0, 3], [1, 3], [1, 4], [2, 4], [2, 2], [3, 2], [3, 1]],
3336: (4)               ]
3337: (4)               bottoms = [
3338: (8)                   [[2, 0], [2, 0], [1, 0], [1, 0], [0, 0]],
3339: (8)                   [[2, 1], [2, 3], [1, 3], [1, 2], [0, 2]],
3340: (4)               ]
3341: (4)               combined = [t + b for t, b in zip(tops, bottoms)]
3342: (4)               for histtype, xy in [('step', tops), ('stepfilled', combined)]:
3343: (8)                   _, _, patches = plt.hist([data_1, data_2], bins=bins, stacked=True,
3344: (33)                                            histtype=histtype)
3345: (8)                   assert len(patches) == 2
3346: (8)                   polygon, = patches[0]
3347: (8)                   assert_array_equal(polygon.get_xy(), xy[0])
3348: (8)                   polygon, = patches[1]
3349: (8)                   assert_array_equal(polygon.get_xy(), xy[1])
3350: (0)           def test_hist_stacked_step_bottom_geometry():
3351: (4)               bins = [0, 1, 2, 3]
3352: (4)               data_1 = [0, 0, 1, 1, 1, 2]
3353: (4)               data_2 = [0, 1, 2]
3354: (4)               tops = [
3355: (8)                   [[0, 1], [0, 3], [1, 3], [1, 5], [2, 5], [2, 2.5], [3, 2.5], [3,
1.5]],
3356: (8)                   [[0, 3], [0, 4], [1, 4], [1, 6], [2, 6], [2, 3.5], [3, 3.5], [3,
2.5]],
3357: (4)               ]
3358: (4)               bottoms = [
3359: (8)                   [[2, 1.5], [2, 2], [1, 2], [1, 1], [0, 1]],
3360: (8)                   [[2, 2.5], [2, 5], [1, 5], [1, 3], [0, 3]],
3361: (4)               ]
3362: (4)               combined = [t + b for t, b in zip(tops, bottoms)]
3363: (4)               for histtype, xy in [('step', tops), ('stepfilled', combined)]:
3364: (8)                   _, _, patches = plt.hist([data_1, data_2], bins=bins, stacked=True,
3365: (33)                                            bottom=[1, 2, 1.5], histtype=histtype)
3366: (8)                   assert len(patches) == 2
3367: (8)                   polygon, = patches[0]
3368: (8)                   assert_array_equal(polygon.get_xy(), xy[0])
3369: (8)                   polygon, = patches[1]
3370: (8)                   assert_array_equal(polygon.get_xy(), xy[1])
3371: (0)           @image_comparison(['hist_stacked_bar'])
3372: (0)           def test_hist_stacked_bar():
3373: (4)               d = [[100, 100, 100, 100, 200, 320, 450, 80, 20, 600, 310, 800],
3374: (9)                    [20, 23, 50, 11, 100, 420], [120, 120, 120, 140, 140, 150, 180],
3375: (9)                    [60, 60, 60, 60, 300, 300, 5, 5, 5, 5, 10, 300],
3376: (9)                    [555, 555, 555, 30, 30, 30, 30, 30, 100, 100, 100, 100, 30, 30],
3377: (9)                    [30, 30, 30, 30, 400, 400, 400, 400, 400, 400, 400, 400]]
3378: (4)               colors = [(0.5759849696758961, 1.0, 0.0), (0.0, 1.0, 0.350624650815206),
3379: (14)                        (0.0, 1.0, 0.6549834156005998), (0.0, 0.6569064625276622, 1.0),
3380: (14)                        (0.28302699607823545, 0.0, 1.0), (0.6849123462299822, 0.0, 1.0)]
3381: (4)               labels = ['green', 'orange', ' yellow', 'magenta', 'black']
3382: (4)               fig, ax = plt.subplots()
3383: (4)               ax.hist(d, bins=10, histtype='barstacked', align='mid', color=colors,
3384: (12)                      label=labels)
3385: (4)               ax.legend(loc='upper right', bbox_to_anchor=(1.0, 1.0), ncols=1)
3386: (0)           def test_hist_barstacked_bottom_unchanged():
3387: (4)               b = np.array([10, 20])
3388: (4)               plt.hist([[0, 1], [0, 1]], 2, histtype="barstacked", bottom=b)
3389: (4)               assert b.tolist() == [10, 20]
3390: (0)           def test_hist_emptydata():
3391: (4)               fig, ax = plt.subplots()
3392: (4)               ax.hist([[], range(10), range(10)], histtype="step")
3393: (0)           def test_hist_labels():
3394: (4)               fig, ax = plt.subplots()
3395: (4)               _, _, bars = ax.hist([0, 1], label=0)
3396: (4)               assert bars[0].get_label() == '0'
3397: (4)               _, _, bars = ax.hist([0, 1], label=[0])
3398: (4)               assert bars[0].get_label() == '0'
3399: (4)               _, _, bars = ax.hist([0, 1], label=None)
3400: (4)               assert bars[0].get_label() == '_nolegend_'
3401: (4)               _, _, bars = ax.hist([0, 1], label='0')
```

```
(4)             assert bars[0].get_label() == '0'
(4)             _, _, bars = ax.hist([0, 1], label='00')
(4)             assert bars[0].get_label() == '00'
(0)         @image_comparison(['transparent_markers'], remove_text=True)
(0)         def test_transparent_markers():
(4)             np.random.seed(0)
(4)             data = np.random.random(50)
(4)             fig, ax = plt.subplots()
(4)             ax.plot(data, 'D', mfc='none', markersize=100)
(0)         @image_comparison(['rgba_markers'], remove_text=True)
(0)         def test_rgba_markers():
(4)             fig, axs = plt.subplots(ncols=2)
(4)             rcolors = [(1, 0, 0, 1), (1, 0, 0, 0.5)]
(4)             bcolors = [(0, 0, 1, 1), (0, 0, 1, 0.5)]
(4)             alphas = [None, 0.2]
(4)             kw = dict(ms=100, mew=20)
(4)             for i, alpha in enumerate(alphas):
(8)                 for j, rcolor in enumerate(rcolors):
(12)                    for k, bcolor in enumerate(bcolors):
(16)                        axs[i].plot(j+1, k+1, 'o', mfc=bcolor, mec=rcolor,
(28)                                    alpha=alpha, **kw)
(16)                        axs[i].plot(j+1, k+3, 'x', mec=rcolor, alpha=alpha, **kw)
(4)             for ax in axs:
(8)                 ax.axis([-1, 4, 0, 5])
(0)         @image_comparison(['mollweide_grid'], remove_text=True)
(0)         def test_mollweide_grid():
(4)             fig = plt.figure()
(4)             ax = fig.add_subplot(projection='mollweide')
(4)             ax.grid()
(0)         def test_mollweide_forward_inverse_closure():
(4)             fig = plt.figure()
(4)             ax = fig.add_subplot(projection='mollweide')
(4)             lon = np.linspace(-np.pi, np.pi, 360)
(4)             lat = np.linspace(-np.pi / 2.0, np.pi / 2.0, 180)[1:-1]
(4)             lon, lat = np.meshgrid(lon, lat)
(4)             ll = np.vstack((lon.flatten(), lat.flatten())).T
(4)             xy = ax.transProjection.transform(ll)
(4)             ll2 = ax.transProjection.inverted().transform(xy)
(4)             np.testing.assert_array_almost_equal(ll, ll2, 3)
(0)         def test_mollweide_inverse_forward_closure():
(4)             fig = plt.figure()
(4)             ax = fig.add_subplot(projection='mollweide')
(4)             x = np.linspace(0, 1, 500)
(4)             x, y = np.meshgrid(x, x)
(4)             xy = np.vstack((x.flatten(), y.flatten())).T
(4)             ll = ax.transProjection.inverted().transform(xy)
(4)             xy2 = ax.transProjection.transform(ll)
(4)             np.testing.assert_array_almost_equal(xy, xy2, 3)
(0)         @image_comparison(['test_alpha'], remove_text=True)
(0)         def test_alpha():
(4)             np.random.seed(0)
(4)             data = np.random.random(50)
(4)             fig, ax = plt.subplots()
(4)             ax.plot(data, '-D', color=[1, 0, 0], mfc=[1, 0, 0, .5],
(12)                    markersize=20, lw=10)
(4)             ax.plot(data + 2, '-D', color=[1, 0, 0, .5], mfc=[1, 0, 0, .5],
(12)                    markersize=20, lw=10,
(12)                    alpha=1)
(4)             ax.plot(data + 4, '-D', color=[1, 0, 0], mfc=[1, 0, 0],
(12)                    markersize=20, lw=10,
(12)                    alpha=.5)
(4)             ax.plot(data + 6, '-D', color=[1, 0, 0, .5], mfc=[1, 0, 0, .5],
(12)                    markersize=20, lw=10)
(4)             ax.plot(data + 8, '-D', color=[1, 0, 0, .5], mfc=[1, 0, 0],
(12)                    markersize=20, lw=10)
(0)         @image_comparison(['eventplot', 'eventplot'], remove_text=True)
(0)         def test_eventplot():
(4)             np.random.seed(0)
(4)             data1 = np.random.random([32, 20]).tolist()
```

```
(4)             data2 = np.random.random([6, 20]).tolist()
(4)             data = data1 + data2
(4)             num_datasets = len(data)
(4)             colors1 = [[0, 1, .7]] * len(data1)
(4)             colors2 = [[1, 0, 0],
(15)                       [0, 1, 0],
(15)                       [0, 0, 1],
(15)                       [1, .75, 0],
(15)                       [1, 0, 1],
(15)                       [0, 1, 1]]
(4)             colors = colors1 + colors2
(4)             lineoffsets1 = 12 + np.arange(0, len(data1)) * .33
(4)             lineoffsets2 = [-15, -3, 1, 1.5, 6, 10]
(4)             lineoffsets = lineoffsets1.tolist() + lineoffsets2
(4)             linelengths1 = [.33] * len(data1)
(4)             linelengths2 = [5, 2, 1, 1, 3, 1.5]
(4)             linelengths = linelengths1 + linelengths2
(4)             fig = plt.figure()
(4)             axobj = fig.add_subplot()
(4)             colls = axobj.eventplot(data, colors=colors, lineoffsets=lineoffsets,
(28)                                     linelengths=linelengths)
(4)             num_collections = len(colls)
(4)             assert num_collections == num_datasets
(4)             data = {"pos": data, "c": colors, "lo": lineoffsets, "ll": linelengths}
(4)             fig = plt.figure()
(4)             axobj = fig.add_subplot()
(4)             colls = axobj.eventplot("pos", colors="c", lineoffsets="lo",
(28)                                     linelengths="ll", data=data)
(4)             num_collections = len(colls)
(4)             assert num_collections == num_datasets
(0)         @image_comparison(['test_eventplot_defaults.png'], remove_text=True)
(0)         def test_eventplot_defaults():
(4)             """
(4)             test that eventplot produces the correct output given the default params
(4)             (see bug #3728)
(4)             """
(4)             np.random.seed(0)
(4)             data1 = np.random.random([32, 20]).tolist()
(4)             data2 = np.random.random([6, 20]).tolist()
(4)             data = data1 + data2
(4)             fig = plt.figure()
(4)             axobj = fig.add_subplot()
(4)             axobj.eventplot(data)
(0)         @pytest.mark.parametrize(('colors'), [
(4)             ('0.5',),  # string color with multiple characters: not OK before #8193
fix
(4)             ('tab:orange', 'tab:pink', 'tab:cyan', 'bLacK'),  # case-insensitive
(4)             ('red', (0, 1, 0), None, (1, 0, 1, 0.5)),  # a tricky case mixing types
(0)         ])
(0)         def test_eventplot_colors(colors):
(4)             """Test the *colors* parameter of eventplot. Inspired by issue #8193."""
(4)             data = [[0], [1], [2], [3]]  # 4 successive events of different nature
(4)             expected = [c if c is not None else 'C0' for c in colors]
(4)             if len(expected) == 1:
(8)                 expected = expected[0]
(4)             expected = np.broadcast_to(mcolors.to_rgba_array(expected), (len(data),
4))
(4)             fig, ax = plt.subplots()
(4)             if len(colors) == 1:  # tuple with a single string (like '0.5' or 'rgbk')
(8)                 colors = colors[0]
(4)             collections = ax.eventplot(data, colors=colors)
(4)             for coll, color in zip(collections, expected):
(8)                 assert_allclose(coll.get_color(), color)
(0)         def test_eventplot_alpha():
(4)             fig, ax = plt.subplots()
(4)             collections = ax.eventplot([[0, 2, 4], [1, 3, 5, 7]], alpha=0.7)
(4)             assert collections[0].get_alpha() == 0.7
(4)             assert collections[1].get_alpha() == 0.7
(4)             collections = ax.eventplot([[0, 2, 4], [1, 3, 5, 7]], alpha=[0.5, 0.7])
```

```
(4)             assert collections[0].get_alpha() == 0.5
(4)             assert collections[1].get_alpha() == 0.7
(4)             with pytest.raises(ValueError, match="alpha and positions are unequal"):
(8)                 ax.eventplot([[0, 2, 4], [1, 3, 5, 7]], alpha=[0.5, 0.7, 0.9])
(4)             with pytest.raises(ValueError, match="alpha and positions are unequal"):
(8)                 ax.eventplot([0, 2, 4], alpha=[0.5, 0.7])
(0)         @image_comparison(['test_eventplot_problem_kwargs.png'], remove_text=True)
(0)         def test_eventplot_problem_kwargs(recwarn):
(4)             """
(4)             test that 'singular' versions of LineCollection props raise an
(4)             MatplotlibDeprecationWarning rather than overriding the 'plural' versions
(4)             (e.g., to prevent 'color' from overriding 'colors', see issue #4297)
(4)             """
(4)             np.random.seed(0)
(4)             data1 = np.random.random([20]).tolist()
(4)             data2 = np.random.random([10]).tolist()
(4)             data = [data1, data2]
(4)             fig = plt.figure()
(4)             axobj = fig.add_subplot()
(4)             axobj.eventplot(data,
(20)                            colors=['r', 'b'],
(20)                            color=['c', 'm'],
(20)                            linewidths=[2, 1],
(20)                            linewidth=[1, 2],
(20)                            linestyles=['solid', 'dashed'],
(20)                            linestyle=['dashdot', 'dotted'])
(4)             assert len(recwarn) == 3
(4)             assert all(issubclass(wi.category, mpl.MatplotlibDeprecationWarning)
(15)                       for wi in recwarn)
(0)         def test_empty_eventplot():
(4)             fig, ax = plt.subplots(1, 1)
(4)             ax.eventplot([[]], colors=[(0.0, 0.0, 0.0, 0.0)])
(4)             plt.draw()
(0)         @pytest.mark.parametrize('data', [[[]], [[], [0, 1]], [[0, 1], []]])
(0)         @pytest.mark.parametrize('orientation', [None, 'vertical', 'horizontal'])
(0)         def test_eventplot_orientation(data, orientation):
(4)             """Introduced when fixing issue #6412."""
(4)             opts = {} if orientation is None else {'orientation': orientation}
(4)             fig, ax = plt.subplots(1, 1)
(4)             ax.eventplot(data, **opts)
(4)             plt.draw()
(0)         @check_figures_equal(extensions=['png'])
(0)         def test_eventplot_units_list(fig_test, fig_ref):
(4)             ts_1 = [datetime.datetime(2021, 1, 1), datetime.datetime(2021, 1, 2),
(12)                    datetime.datetime(2021, 1, 3)]
(4)             ts_2 = [datetime.datetime(2021, 1, 15), datetime.datetime(2021, 1, 16)]
(4)             ax = fig_ref.subplots()
(4)             ax.eventplot(ts_1, lineoffsets=0)
(4)             ax.eventplot(ts_2, lineoffsets=1)
(4)             ax = fig_test.subplots()
(4)             ax.eventplot([ts_1, ts_2])
(0)         @image_comparison(['marker_styles.png'], remove_text=True)
(0)         def test_marker_styles():
(4)             fig, ax = plt.subplots()
(4)             markers = sorted(matplotlib.markers.MarkerStyle.markers,
(21)                             key=lambda x: str(type(x))+str(x))
(4)             markers.remove('none')
(4)             markers = ['none', *markers]
(4)             for y, marker in enumerate(markers):
(8)                 ax.plot((y % 2)*5 + np.arange(10)*10, np.ones(10)*10*y, linestyle='',
(16)                        marker=marker, markersize=10+y/5, label=marker)
(0)         @image_comparison(['rc_markerfill.png'],
(18)                          tol=0.037 if platform.machine() == 'arm64' else 0)
(0)         def test_markers_fillstyle_rcparams():
(4)             fig, ax = plt.subplots()
(4)             x = np.arange(7)
(4)             for idx, (style, marker) in enumerate(
(12)                    [('top', 's'), ('bottom', 'o'), ('none', '^')]):
(8)                 matplotlib.rcParams['markers.fillstyle'] = style
```

```
(8)                 ax.plot(x+idx, marker=marker)
(0)         @image_comparison(['vertex_markers.png'], remove_text=True)
(0)         def test_vertex_markers():
(4)             data = list(range(10))
(4)             marker_as_tuple = ((-1, -1), (1, -1), (1, 1), (-1, 1))
(4)             marker_as_list = [(-1, -1), (1, -1), (1, 1), (-1, 1)]
(4)             fig, ax = plt.subplots()
(4)             ax.plot(data, linestyle='', marker=marker_as_tuple, mfc='k')
(4)             ax.plot(data[::-1], linestyle='', marker=marker_as_list, mfc='b')
(4)             ax.set_xlim([-1, 10])
(4)             ax.set_ylim([-1, 10])
(0)         @image_comparison(['vline_hline_zorder', 'errorbar_zorder'],
(18)                          tol=0 if platform.machine() == 'x86_64' else 0.026)
(0)         def test_eb_line_zorder():
(4)             x = list(range(10))
(4)             fig = plt.figure()
(4)             ax = fig.gca()
(4)             ax.plot(x, lw=10, zorder=5)
(4)             ax.axhline(1, color='red', lw=10, zorder=1)
(4)             ax.axhline(5, color='green', lw=10, zorder=10)
(4)             ax.axvline(7, color='m', lw=10, zorder=7)
(4)             ax.axvline(2, color='k', lw=10, zorder=3)
(4)             ax.set_title("axvline and axhline zorder test")
(4)             fig = plt.figure()
(4)             ax = fig.gca()
(4)             x = list(range(10))
(4)             y = np.zeros(10)
(4)             yerr = list(range(10))
(4)             ax.errorbar(x, y, yerr=yerr, zorder=5, lw=5, color='r')
(4)             for j in range(10):
(8)                 ax.axhline(j, lw=5, color='k', zorder=j)
(8)                 ax.axhline(-j, lw=5, color='k', zorder=j)
(4)             ax.set_title("errorbar zorder test")
(0)         @check_figures_equal()
(0)         def test_axline_loglog(fig_test, fig_ref):
(4)             ax = fig_test.subplots()
(4)             ax.set(xlim=(0.1, 10), ylim=(1e-3, 1))
(4)             ax.loglog([.3, .6], [.3, .6], ".-")
(4)             ax.axline((1, 1e-3), (10, 1e-2), c="k")
(4)             ax = fig_ref.subplots()
(4)             ax.set(xlim=(0.1, 10), ylim=(1e-3, 1))
(4)             ax.loglog([.3, .6], [.3, .6], ".-")
(4)             ax.loglog([1, 10], [1e-3, 1e-2], c="k")
(0)         @check_figures_equal()
(0)         def test_axline(fig_test, fig_ref):
(4)             ax = fig_test.subplots()
(4)             ax.set(xlim=(-1, 1), ylim=(-1, 1))
(4)             ax.axline((0, 0), (1, 1))
(4)             ax.axline((0, 0), (1, 0), color='C1')
(4)             ax.axline((0, 0.5), (1, 0.5), color='C2')
(4)             ax.axline((-0.7, -0.5), slope=0, color='C3')
(4)             ax.axline((1, -0.5), slope=-0.5, color='C4')
(4)             ax.axline((-0.5, 1), slope=float('inf'), color='C5')
(4)             ax = fig_ref.subplots()
(4)             ax.set(xlim=(-1, 1), ylim=(-1, 1))
(4)             ax.plot([-1, 1], [-1, 1])
(4)             ax.axhline(0, color='C1')
(4)             ax.axhline(0.5, color='C2')
(4)             ax.axhline(-0.5, color='C3')
(4)             ax.plot([-1, 1], [0.5, -0.5], color='C4')
(4)             ax.axvline(-0.5, color='C5')
(0)         @check_figures_equal()
(0)         def test_axline_transaxes(fig_test, fig_ref):
(4)             ax = fig_test.subplots()
(4)             ax.set(xlim=(-1, 1), ylim=(-1, 1))
(4)             ax.axline((0, 0), slope=1, transform=ax.transAxes)
(4)             ax.axline((1, 0.5), slope=1, color='C1', transform=ax.transAxes)
(4)             ax.axline((0.5, 0.5), slope=0, color='C2', transform=ax.transAxes)
(4)             ax.axline((0.5, 0), (0.5, 1), color='C3', transform=ax.transAxes)
```

```
(4)             ax = fig_ref.subplots()
(4)             ax.set(xlim=(-1, 1), ylim=(-1, 1))
(4)             ax.plot([-1, 1], [-1, 1])
(4)             ax.plot([0, 1], [-1, 0], color='C1')
(4)             ax.plot([-1, 1], [0, 0], color='C2')
(4)             ax.plot([0, 0], [-1, 1], color='C3')
(0)         @check_figures_equal()
(0)         def test_axline_transaxes_panzoom(fig_test, fig_ref):
(4)             ax = fig_test.subplots()
(4)             ax.set(xlim=(-1, 1), ylim=(-1, 1))
(4)             ax.axline((0, 0), slope=1, transform=ax.transAxes)
(4)             ax.axline((0.5, 0.5), slope=2, color='C1', transform=ax.transAxes)
(4)             ax.axline((0.5, 0.5), slope=0, color='C2', transform=ax.transAxes)
(4)             ax.set(xlim=(0, 5), ylim=(0, 10))
(4)             fig_test.set_size_inches(3, 3)
(4)             ax = fig_ref.subplots()
(4)             ax.set(xlim=(0, 5), ylim=(0, 10))
(4)             fig_ref.set_size_inches(3, 3)
(4)             ax.plot([0, 5], [0, 5])
(4)             ax.plot([0, 5], [0, 10], color='C1')
(4)             ax.plot([0, 5], [5, 5], color='C2')
(0)         def test_axline_args():
(4)             """Exactly one of *xy2* and *slope* must be specified."""
(4)             fig, ax = plt.subplots()
(4)             with pytest.raises(TypeError):
(8)                 ax.axline((0, 0))  # missing second parameter
(4)             with pytest.raises(TypeError):
(8)                 ax.axline((0, 0), (1, 1), slope=1)  # redundant parameters
(4)             ax.set_xscale('log')
(4)             with pytest.raises(TypeError):
(8)                 ax.axline((0, 0), slope=1)
(4)             ax.set_xscale('linear')
(4)             ax.set_yscale('log')
(4)             with pytest.raises(TypeError):
(8)                 ax.axline((0, 0), slope=1)
(4)             ax.set_yscale('linear')
(4)             with pytest.raises(ValueError):
(8)                 ax.axline((0, 0), (0, 0))  # two identical points are not allowed
(8)                 plt.draw()
(0)         @image_comparison(['vlines_basic', 'vlines_with_nan', 'vlines_masked'],
(18)                           extensions=['png'])
(0)         def test_vlines():
(4)             x1 = [2, 3, 4, 5, 7]
(4)             y1 = [2, -6, 3, 8, 2]
(4)             fig1, ax1 = plt.subplots()
(4)             ax1.vlines(x1, 0, y1, colors='g', linewidth=5)
(4)             x2 = [2, 3, 4, 5, 6, 7]
(4)             y2 = [2, -6, 3, 8, np.nan, 2]
(4)             fig2, (ax2, ax3, ax4) = plt.subplots(nrows=3, figsize=(4, 8))
(4)             ax2.vlines(x2, 0, y2, colors='g', linewidth=5)
(4)             x3 = [2, 3, 4, 5, 6, 7]
(4)             y3 = [np.nan, 2, -6, 3, 8, 2]
(4)             ax3.vlines(x3, 0, y3, colors='r', linewidth=3, linestyle='--')
(4)             x4 = [2, 3, 4, 5, 6, 7]
(4)             y4 = [np.nan, 2, -6, 3, 8, np.nan]
(4)             ax4.vlines(x4, 0, y4, colors='k', linewidth=2)
(4)             for ax in [ax1, ax2, ax3, ax4]:
(8)                 ax.set_xlim(0, 10)
(4)             assert ax1.get_ylim() == ax2.get_ylim()
(4)             assert ax1.get_ylim() == ax3.get_ylim()
(4)             assert ax1.get_ylim() == ax4.get_ylim()
(4)             fig3, ax5 = plt.subplots()
(4)             x5 = np.ma.masked_equal([2, 4, 6, 8, 10, 12], 8)
(4)             ymin5 = np.ma.masked_equal([0, 1, -1, 0, 2, 1], 2)
(4)             ymax5 = np.ma.masked_equal([13, 14, 15, 16, 17, 18], 18)
(4)             ax5.vlines(x5, ymin5, ymax5, colors='k', linewidth=2)
(4)             ax5.set_xlim(0, 15)
(0)         def test_vlines_default():
(4)             fig, ax = plt.subplots()
```

```
(4)             with mpl.rc_context({'lines.color': 'red'}):
(8)                 lines = ax.vlines(0.5, 0, 1)
(8)                 assert mpl.colors.same_color(lines.get_color(), 'red')
(0)         @image_comparison(['hlines_basic', 'hlines_with_nan', 'hlines_masked'],
(18)                          extensions=['png'])
(0)         def test_hlines():
(4)             y1 = [2, 3, 4, 5, 7]
(4)             x1 = [2, -6, 3, 8, 2]
(4)             fig1, ax1 = plt.subplots()
(4)             ax1.hlines(y1, 0, x1, colors='g', linewidth=5)
(4)             y2 = [2, 3, 4, 5, 6, 7]
(4)             x2 = [2, -6, 3, 8, np.nan, 2]
(4)             fig2, (ax2, ax3, ax4) = plt.subplots(nrows=3, figsize=(4, 8))
(4)             ax2.hlines(y2, 0, x2, colors='g', linewidth=5)
(4)             y3 = [2, 3, 4, 5, 6, 7]
(4)             x3 = [np.nan, 2, -6, 3, 8, 2]
(4)             ax3.hlines(y3, 0, x3, colors='r', linewidth=3, linestyle='--')
(4)             y4 = [2, 3, 4, 5, 6, 7]
(4)             x4 = [np.nan, 2, -6, 3, 8, np.nan]
(4)             ax4.hlines(y4, 0, x4, colors='k', linewidth=2)
(4)             for ax in [ax1, ax2, ax3, ax4]:
(8)                 ax.set_ylim(0, 10)
(4)             assert ax1.get_xlim() == ax2.get_xlim()
(4)             assert ax1.get_xlim() == ax3.get_xlim()
(4)             assert ax1.get_xlim() == ax4.get_xlim()
(4)             fig3, ax5 = plt.subplots()
(4)             y5 = np.ma.masked_equal([2, 4, 6, 8, 10, 12], 8)
(4)             xmin5 = np.ma.masked_equal([0, 1, -1, 0, 2, 1], 2)
(4)             xmax5 = np.ma.masked_equal([13, 14, 15, 16, 17, 18], 18)
(4)             ax5.hlines(y5, xmin5, xmax5, colors='k', linewidth=2)
(4)             ax5.set_ylim(0, 15)
(0)         def test_hlines_default():
(4)             fig, ax = plt.subplots()
(4)             with mpl.rc_context({'lines.color': 'red'}):
(8)                 lines = ax.hlines(0.5, 0, 1)
(8)                 assert mpl.colors.same_color(lines.get_color(), 'red')
(0)         @pytest.mark.parametrize('data', [[1, 2, 3, np.nan, 5],
(34)                                           np.ma.masked_equal([1, 2, 3, 4, 5], 4)])
(0)         @check_figures_equal(extensions=["png"])
(0)         def test_lines_with_colors(fig_test, fig_ref, data):
(4)             test_colors = ['red', 'green', 'blue', 'purple', 'orange']
(4)             fig_test.add_subplot(2, 1, 1).vlines(data, 0, 1,
(41)                                                  colors=test_colors, linewidth=5)
(4)             fig_test.add_subplot(2, 1, 2).hlines(data, 0, 1,
(41)                                                  colors=test_colors, linewidth=5)
(4)             expect_xy = [1, 2, 3, 5]
(4)             expect_color = ['red', 'green', 'blue', 'orange']
(4)             fig_ref.add_subplot(2, 1, 1).vlines(expect_xy, 0, 1,
(40)                                                colors=expect_color, linewidth=5)
(4)             fig_ref.add_subplot(2, 1, 2).hlines(expect_xy, 0, 1,
(40)                                                colors=expect_color, linewidth=5)
(0)         @image_comparison(['vlines_hlines_blended_transform'],
(18)                          extensions=['png'], style='mpl20')
(0)         def test_vlines_hlines_blended_transform():
(4)             t = np.arange(5.0, 10.0, 0.1)
(4)             s = np.exp(-t) + np.sin(2 * np.pi * t) + 10
(4)             fig, (hax, vax) = plt.subplots(2, 1, figsize=(6, 6))
(4)             hax.plot(t, s, '^')
(4)             hax.hlines([10, 9], xmin=0, xmax=0.5,
(15)                       transform=hax.get_yaxis_transform(), colors='r')
(4)             vax.plot(t, s, '^')
(4)             vax.vlines([6, 7], ymin=0, ymax=0.15, transform=vax.get_xaxis_transform(),
(15)                       colors='r')
(0)         @image_comparison(['step_linestyle', 'step_linestyle'], remove_text=True,
(18)                          tol=0.2)
(0)         def test_step_linestyle():
(4)             x = y = np.arange(10)
(4)             fig, ax_lst = plt.subplots(2, 2)
(4)             ax_lst = ax_lst.flatten()
```

```
3814: (4)               ln_styles = ['-', '--', '-.', ':']
3815: (4)               for ax, ls in zip(ax_lst, ln_styles):
3816: (8)                   ax.step(x, y, lw=5, linestyle=ls, where='pre')
3817: (8)                   ax.step(x, y + 1, lw=5, linestyle=ls, where='mid')
3818: (8)                   ax.step(x, y + 2, lw=5, linestyle=ls, where='post')
3819: (8)                   ax.set_xlim([-1, 5])
3820: (8)                   ax.set_ylim([-1, 7])
3821: (4)               data = {"X": x, "Y0": y, "Y1": y+1, "Y2": y+2}
3822: (4)               fig, ax_lst = plt.subplots(2, 2)
3823: (4)               ax_lst = ax_lst.flatten()
3824: (4)               ln_styles = ['-', '--', '-.', ':']
3825: (4)               for ax, ls in zip(ax_lst, ln_styles):
3826: (8)                   ax.step("X", "Y0", lw=5, linestyle=ls, where='pre', data=data)
3827: (8)                   ax.step("X", "Y1", lw=5, linestyle=ls, where='mid', data=data)
3828: (8)                   ax.step("X", "Y2", lw=5, linestyle=ls, where='post', data=data)
3829: (8)                   ax.set_xlim([-1, 5])
3830: (8)                   ax.set_ylim([-1, 7])
3831: (0)           @image_comparison(['mixed_collection'], remove_text=True)
3832: (0)           def test_mixed_collection():
3833: (4)               fig, ax = plt.subplots()
3834: (4)               c = mpatches.Circle((8, 8), radius=4, facecolor='none', edgecolor='green')
3835: (4)               p1 = mpl.collections.PatchCollection([c], match_original=True)
3836: (4)               p1.set_offsets([[0, 0], [24, 24]])
3837: (4)               p1.set_linewidths([1, 5])
3838: (4)               p2 = mpl.collections.PatchCollection([c], match_original=True)
3839: (4)               p2.set_offsets([[48, 0], [-32, -16]])
3840: (4)               p2.set_linewidths([1, 5])
3841: (4)               p2.set_edgecolors([[0, 0, 0.1, 1.0], [0, 0, 0.1, 0.5]])
3842: (4)               ax.patch.set_color('0.5')
3843: (4)               ax.add_collection(p1)
3844: (4)               ax.add_collection(p2)
3845: (4)               ax.set_xlim(0, 16)
3846: (4)               ax.set_ylim(0, 16)
3847: (0)           def test_subplot_key_hash():
3848: (4)               ax = plt.subplot(np.int32(5), np.int64(1), 1)
3849: (4)               ax.twinx()
3850: (4)               assert ax.get_subplotspec().get_geometry() == (5, 1, 0, 0)
3851: (0)           @image_comparison(
3852: (4)               ["specgram_freqs.png", "specgram_freqs_linear.png",
3853: (5)                "specgram_noise.png", "specgram_noise_linear.png"],
3854: (4)               remove_text=True, tol=0.07, style="default")
3855: (0)           def test_specgram():
3856: (4)               """Test axes.specgram in default (psd) mode."""
3857: (4)               matplotlib.rcParams['image.interpolation'] = 'nearest'
3858: (4)               n = 1000
3859: (4)               Fs = 10.
3860: (4)               fstims = [[Fs/4, Fs/5, Fs/11], [Fs/4.7, Fs/5.6, Fs/11.9]]
3861: (4)               NFFT_freqs = int(10 * Fs / np.min(fstims))
3862: (4)               x = np.arange(0, n, 1/Fs)
3863: (4)               y_freqs = np.concatenate(
3864: (8)                   np.sin(2 * np.pi * np.multiply.outer(fstims, x)).sum(axis=1))
3865: (4)               NFFT_noise = int(10 * Fs / 11)
3866: (4)               np.random.seed(0)
3867: (4)               y_noise = np.concatenate([np.random.standard_normal(n),
np.random.rand(n)])
3868: (4)               all_sides = ["default", "onesided", "twosided"]
3869: (4)               for y, NFFT in [(y_freqs, NFFT_freqs), (y_noise, NFFT_noise)]:
3870: (8)                   noverlap = NFFT // 2
3871: (8)                   pad_to = int(2 ** np.ceil(np.log2(NFFT)))
3872: (8)                   for ax, sides in zip(plt.figure().subplots(3), all_sides):
3873: (12)                      ax.specgram(y, NFFT=NFFT, Fs=Fs, noverlap=noverlap,
3874: (24)                                  pad_to=pad_to, sides=sides)
3875: (8)                   for ax, sides in zip(plt.figure().subplots(3), all_sides):
3876: (12)                      ax.specgram(y, NFFT=NFFT, Fs=Fs, noverlap=noverlap,
3877: (24)                                  pad_to=pad_to, sides=sides,
3878: (24)                                  scale="linear", norm=matplotlib.colors.LogNorm())
3879: (0)           @image_comparison(
3880: (4)               ["specgram_magnitude_freqs.png", "specgram_magnitude_freqs_linear.png",
3881: (5)                "specgram_magnitude_noise.png", "specgram_magnitude_noise_linear.png"],
```

```
3882: (4)               remove_text=True, tol=0.07, style="default")
3883: (0)           def test_specgram_magnitude():
3884: (4)               """Test axes.specgram in magnitude mode."""
3885: (4)               matplotlib.rcParams['image.interpolation'] = 'nearest'
3886: (4)               n = 1000
3887: (4)               Fs = 10.
3888: (4)               fstims = [[Fs/4, Fs/5, Fs/11], [Fs/4.7, Fs/5.6, Fs/11.9]]
3889: (4)               NFFT_freqs = int(100 * Fs / np.min(fstims))
3890: (4)               x = np.arange(0, n, 1/Fs)
3891: (4)               y = np.sin(2 * np.pi * np.multiply.outer(fstims, x)).sum(axis=1)
3892: (4)               y[:, -1] = 1
3893: (4)               y_freqs = np.hstack(y)
3894: (4)               NFFT_noise = int(10 * Fs / 11)
3895: (4)               np.random.seed(0)
3896: (4)               y_noise = np.concatenate([np.random.standard_normal(n),
np.random.rand(n)])
3897: (4)               all_sides = ["default", "onesided", "twosided"]
3898: (4)               for y, NFFT in [(y_freqs, NFFT_freqs), (y_noise, NFFT_noise)]:
3899: (8)                   noverlap = NFFT // 2
3900: (8)                   pad_to = int(2 ** np.ceil(np.log2(NFFT)))
3901: (8)                   for ax, sides in zip(plt.figure().subplots(3), all_sides):
3902: (12)                      ax.specgram(y, NFFT=NFFT, Fs=Fs, noverlap=noverlap,
3903: (24)                                  pad_to=pad_to, sides=sides, mode="magnitude")
3904: (8)                   for ax, sides in zip(plt.figure().subplots(3), all_sides):
3905: (12)                      ax.specgram(y, NFFT=NFFT, Fs=Fs, noverlap=noverlap,
3906: (24)                                  pad_to=pad_to, sides=sides, mode="magnitude",
3907: (24)                                  scale="linear", norm=matplotlib.colors.LogNorm())
3908: (0)           @image_comparison(
3909: (4)               ["specgram_angle_freqs.png", "specgram_phase_freqs.png",
3910: (5)                "specgram_angle_noise.png", "specgram_phase_noise.png"],
3911: (4)               remove_text=True, tol=0.07, style="default")
3912: (0)           def test_specgram_angle():
3913: (4)               """Test axes.specgram in angle and phase modes."""
3914: (4)               matplotlib.rcParams['image.interpolation'] = 'nearest'
3915: (4)               n = 1000
3916: (4)               Fs = 10.
3917: (4)               fstims = [[Fs/4, Fs/5, Fs/11], [Fs/4.7, Fs/5.6, Fs/11.9]]
3918: (4)               NFFT_freqs = int(10 * Fs / np.min(fstims))
3919: (4)               x = np.arange(0, n, 1/Fs)
3920: (4)               y = np.sin(2 * np.pi * np.multiply.outer(fstims, x)).sum(axis=1)
3921: (4)               y[:, -1] = 1
3922: (4)               y_freqs = np.hstack(y)
3923: (4)               NFFT_noise = int(10 * Fs / 11)
3924: (4)               np.random.seed(0)
3925: (4)               y_noise = np.concatenate([np.random.standard_normal(n),
np.random.rand(n)])
3926: (4)               all_sides = ["default", "onesided", "twosided"]
3927: (4)               for y, NFFT in [(y_freqs, NFFT_freqs), (y_noise, NFFT_noise)]:
3928: (8)                   noverlap = NFFT // 2
3929: (8)                   pad_to = int(2 ** np.ceil(np.log2(NFFT)))
3930: (8)                   for mode in ["angle", "phase"]:
3931: (12)                      for ax, sides in zip(plt.figure().subplots(3), all_sides):
3932: (16)                          ax.specgram(y, NFFT=NFFT, Fs=Fs, noverlap=noverlap,
3933: (28)                                      pad_to=pad_to, sides=sides, mode=mode)
3934: (16)                          with pytest.raises(ValueError):
3935: (20)                              ax.specgram(y, NFFT=NFFT, Fs=Fs, noverlap=noverlap,
3936: (32)                                          pad_to=pad_to, sides=sides, mode=mode,
3937: (32)                                          scale="dB")
3938: (0)           def test_specgram_fs_none():
3939: (4)               """Test axes.specgram when Fs is None, should not throw error."""
3940: (4)               spec, freqs, t, im = plt.specgram(np.ones(300), Fs=None, scale='linear')
3941: (4)               xmin, xmax, freq0, freq1 = im.get_extent()
3942: (4)               assert xmin == 32 and xmax == 96
3943: (0)           @check_figures_equal(extensions=["png"])
3944: (0)           def test_specgram_origin_rcparam(fig_test, fig_ref):
3945: (4)               """Test specgram ignores image.origin rcParam and uses origin 'upper'."""
3946: (4)               t = np.arange(500)
3947: (4)               signal = np.sin(t)
3948: (4)               plt.rcParams["image.origin"] = 'upper'
```

```
3949: (4)               fig_ref.subplots().specgram(signal)
3950: (4)               plt.rcParams["image.origin"] = 'lower'
3951: (4)               fig_test.subplots().specgram(signal)
3952: (0)           def test_specgram_origin_kwarg():
3953: (4)               """Ensure passing origin as a kwarg raises a TypeError."""
3954: (4)               t = np.arange(500)
3955: (4)               signal = np.sin(t)
3956: (4)               with pytest.raises(TypeError):
3957: (8)                   plt.specgram(signal, origin='lower')
3958: (0)           @image_comparison(
3959: (4)               ["psd_freqs.png", "csd_freqs.png", "psd_noise.png", "csd_noise.png"],
3960: (4)               remove_text=True, tol=0.002)
3961: (0)           def test_psd_csd():
3962: (4)               n = 10000
3963: (4)               Fs = 100.
3964: (4)               fstims = [[Fs/4, Fs/5, Fs/11], [Fs/4.7, Fs/5.6, Fs/11.9]]
3965: (4)               NFFT_freqs = int(1000 * Fs / np.min(fstims))
3966: (4)               x = np.arange(0, n, 1/Fs)
3967: (4)               ys_freqs = np.sin(2 * np.pi * np.multiply.outer(fstims, x)).sum(axis=1)
3968: (4)               NFFT_noise = int(1000 * Fs / 11)
3969: (4)               np.random.seed(0)
3970: (4)               ys_noise = [np.random.standard_normal(n), np.random.rand(n)]
3971: (4)               all_kwargs = [{"sides": "default"},
3972: (18)                            {"sides": "onesided", "return_line": False},
3973: (18)                            {"sides": "twosided", "return_line": True}]
3974: (4)               for ys, NFFT in [(ys_freqs, NFFT_freqs), (ys_noise, NFFT_noise)]:
3975: (8)                   noverlap = NFFT // 2
3976: (8)                   pad_to = int(2 ** np.ceil(np.log2(NFFT)))
3977: (8)                   for ax, kwargs in zip(plt.figure().subplots(3), all_kwargs):
3978: (12)                      ret = ax.psd(np.concatenate(ys), NFFT=NFFT, Fs=Fs,
3979: (25)                                   noverlap=noverlap, pad_to=pad_to, **kwargs)
3980: (12)                      assert len(ret) == 2 + kwargs.get("return_line", False)
3981: (12)                      ax.set(xlabel="", ylabel="")
3982: (8)                   for ax, kwargs in zip(plt.figure().subplots(3), all_kwargs):
3983: (12)                      ret = ax.csd(*ys, NFFT=NFFT, Fs=Fs,
3984: (25)                                   noverlap=noverlap, pad_to=pad_to, **kwargs)
3985: (12)                      assert len(ret) == 2 + kwargs.get("return_line", False)
3986: (12)                      ax.set(xlabel="", ylabel="")
3987: (0)           @image_comparison(
3988: (4)               ["magnitude_spectrum_freqs_linear.png",
3989: (5)                "magnitude_spectrum_freqs_dB.png",
3990: (5)                "angle_spectrum_freqs.png",
3991: (5)                "phase_spectrum_freqs.png",
3992: (5)                "magnitude_spectrum_noise_linear.png",
3993: (5)                "magnitude_spectrum_noise_dB.png",
3994: (5)                "angle_spectrum_noise.png",
3995: (5)                "phase_spectrum_noise.png"],
3996: (4)               remove_text=True)
3997: (0)           def test_spectrum():
3998: (4)               n = 10000
3999: (4)               Fs = 100.
4000: (4)               fstims1 = [Fs/4, Fs/5, Fs/11]
4001: (4)               NFFT = int(1000 * Fs / min(fstims1))
4002: (4)               pad_to = int(2 ** np.ceil(np.log2(NFFT)))
4003: (4)               x = np.arange(0, n, 1/Fs)
4004: (4)               y_freqs = ((np.sin(2 * np.pi * np.outer(x, fstims1)) * 10**np.arange(3))
4005: (15)                         .sum(axis=1))
4006: (4)               np.random.seed(0)
4007: (4)               y_noise = np.hstack([np.random.standard_normal(n), np.random.rand(n)]) -
.5
4008: (4)               all_sides = ["default", "onesided", "twosided"]
4009: (4)               kwargs = {"Fs": Fs, "pad_to": pad_to}
4010: (4)               for y in [y_freqs, y_noise]:
4011: (8)                   for ax, sides in zip(plt.figure().subplots(3), all_sides):
4012: (12)                      spec, freqs, line = ax.magnitude_spectrum(y, sides=sides,
**kwargs)
4013: (12)                      ax.set(xlabel="", ylabel="")
4014: (8)                   for ax, sides in zip(plt.figure().subplots(3), all_sides):
4015: (12)                      spec, freqs, line = ax.magnitude_spectrum(y, sides=sides,
```

```
**kwargs,
(54)                                                                scale="dB")
(12)                        ax.set(xlabel="", ylabel="")
(8)                     for ax, sides in zip(plt.figure().subplots(3), all_sides):
(12)                        spec, freqs, line = ax.angle_spectrum(y, sides=sides, **kwargs)
(12)                        ax.set(xlabel="", ylabel="")
(8)                     for ax, sides in zip(plt.figure().subplots(3), all_sides):
(12)                        spec, freqs, line = ax.phase_spectrum(y, sides=sides, **kwargs)
(12)                        ax.set(xlabel="", ylabel="")
(0)             def test_psd_csd_edge_cases():
(4)                 axs = plt.figure().subplots(2)
(4)                 for ax in axs:
(8)                     ax.yaxis.set(inverted=True)
(4)                 with np.errstate(divide="ignore"):
(8)                     axs[0].psd(np.zeros(5))
(8)                     axs[1].csd(np.zeros(5), np.zeros(5))
(0)             @check_figures_equal(extensions=['png'])
(0)             def test_twin_remove(fig_test, fig_ref):
(4)                 ax_test = fig_test.add_subplot()
(4)                 ax_twinx = ax_test.twinx()
(4)                 ax_twiny = ax_test.twiny()
(4)                 ax_twinx.remove()
(4)                 ax_twiny.remove()
(4)                 ax_ref = fig_ref.add_subplot()
(4)                 ax_ref.xaxis.tick_bottom()
(4)                 ax_ref.yaxis.tick_left()
(0)             @image_comparison(['twin_spines.png'], remove_text=True,
(18)                            tol=0.022 if platform.machine() == 'arm64' else 0)
(0)             def test_twin_spines():
(4)                 def make_patch_spines_invisible(ax):
(8)                     ax.set_frame_on(True)
(8)                     ax.patch.set_visible(False)
(8)                     ax.spines[:].set_visible(False)
(4)                 fig = plt.figure(figsize=(4, 3))
(4)                 fig.subplots_adjust(right=0.75)
(4)                 host = fig.add_subplot()
(4)                 par1 = host.twinx()
(4)                 par2 = host.twinx()
(4)                 par2.spines.right.set_position(("axes", 1.2))
(4)                 make_patch_spines_invisible(par2)
(4)                 par2.spines.right.set_visible(True)
(4)                 p1, = host.plot([0, 1, 2], [0, 1, 2], "b-")
(4)                 p2, = par1.plot([0, 1, 2], [0, 3, 2], "r-")
(4)                 p3, = par2.plot([0, 1, 2], [50, 30, 15], "g-")
(4)                 host.set_xlim(0, 2)
(4)                 host.set_ylim(0, 2)
(4)                 par1.set_ylim(0, 4)
(4)                 par2.set_ylim(1, 65)
(4)                 host.yaxis.label.set_color(p1.get_color())
(4)                 par1.yaxis.label.set_color(p2.get_color())
(4)                 par2.yaxis.label.set_color(p3.get_color())
(4)                 tkw = dict(size=4, width=1.5)
(4)                 host.tick_params(axis='y', colors=p1.get_color(), **tkw)
(4)                 par1.tick_params(axis='y', colors=p2.get_color(), **tkw)
(4)                 par2.tick_params(axis='y', colors=p3.get_color(), **tkw)
(4)                 host.tick_params(axis='x', **tkw)
(0)             @image_comparison(['twin_spines_on_top.png', 'twin_spines_on_top.png'],
(18)                            remove_text=True)
(0)             def test_twin_spines_on_top():
(4)                 matplotlib.rcParams['axes.linewidth'] = 48.0
(4)                 matplotlib.rcParams['lines.linewidth'] = 48.0
(4)                 fig = plt.figure()
(4)                 ax1 = fig.add_subplot(1, 1, 1)
(4)                 data = np.array([[1000, 1100, 1200, 1250],
(21)                                 [310, 301, 360, 400]])
(4)                 ax2 = ax1.twinx()
(4)                 ax1.plot(data[0], data[1]/1E3, color='#BEAED4')
(4)                 ax1.fill_between(data[0], data[1]/1E3, color='#BEAED4', alpha=.8)
(4)                 ax2.plot(data[0], data[1]/1E3, color='#7FC97F')
```

```
4084: (4)               ax2.fill_between(data[0], data[1]/1E3, color='#7FC97F', alpha=.5)
4085: (4)               data = {"i": data[0], "j": data[1]/1E3}
4086: (4)               fig = plt.figure()
4087: (4)               ax1 = fig.add_subplot(1, 1, 1)
4088: (4)               ax2 = ax1.twinx()
4089: (4)               ax1.plot("i", "j", color='#BEAED4', data=data)
4090: (4)               ax1.fill_between("i", "j", color='#BEAED4', alpha=.8, data=data)
4091: (4)               ax2.plot("i", "j", color='#7FC97F', data=data)
4092: (4)               ax2.fill_between("i", "j", color='#7FC97F', alpha=.5, data=data)
4093: (0)           @pytest.mark.parametrize("grid_which, major_visible, minor_visible", [
4094: (4)               ("both", True, True),
4095: (4)               ("major", True, False),
4096: (4)               ("minor", False, True),
4097: (0)           ])
4098: (0)           def test_rcparam_grid_minor(grid_which, major_visible, minor_visible):
4099: (4)               mpl.rcParams.update({"axes.grid": True, "axes.grid.which": grid_which})
4100: (4)               fig, ax = plt.subplots()
4101: (4)               fig.canvas.draw()
4102: (4)               assert all(tick.gridline.get_visible() == major_visible
4103: (15)                          for tick in ax.xaxis.majorTicks)
4104: (4)               assert all(tick.gridline.get_visible() == minor_visible
4105: (15)                          for tick in ax.xaxis.minorTicks)
4106: (0)           def test_grid():
4107: (4)               fig, ax = plt.subplots()
4108: (4)               ax.grid()
4109: (4)               fig.canvas.draw()
4110: (4)               assert ax.xaxis.majorTicks[0].gridline.get_visible()
4111: (4)               ax.grid(visible=False)
4112: (4)               fig.canvas.draw()
4113: (4)               assert not ax.xaxis.majorTicks[0].gridline.get_visible()
4114: (4)               ax.grid(visible=True)
4115: (4)               fig.canvas.draw()
4116: (4)               assert ax.xaxis.majorTicks[0].gridline.get_visible()
4117: (4)               ax.grid()
4118: (4)               fig.canvas.draw()
4119: (4)               assert not ax.xaxis.majorTicks[0].gridline.get_visible()
4120: (0)           def test_reset_grid():
4121: (4)               fig, ax = plt.subplots()
4122: (4)               ax.tick_params(reset=True, which='major', labelsize=10)
4123: (4)               assert not ax.xaxis.majorTicks[0].gridline.get_visible()
4124: (4)               ax.grid(color='red')  # enables grid
4125: (4)               assert ax.xaxis.majorTicks[0].gridline.get_visible()
4126: (4)               with plt.rc_context({'axes.grid': True}):
4127: (8)                   ax.clear()
4128: (8)                   ax.tick_params(reset=True, which='major', labelsize=10)
4129: (8)                   assert ax.xaxis.majorTicks[0].gridline.get_visible()
4130: (0)           @check_figures_equal(extensions=['png'])
4131: (0)           def test_reset_ticks(fig_test, fig_ref):
4132: (4)               for fig in [fig_ref, fig_test]:
4133: (8)                   ax = fig.add_subplot()
4134: (8)                   ax.grid(True)
4135: (8)                   ax.tick_params(
4136: (12)                      direction='in', length=10, width=5, color='C0', pad=12,
4137: (12)                      labelsize=14, labelcolor='C1', labelrotation=45,
4138: (12)                      grid_color='C2', grid_alpha=0.8, grid_linewidth=3,
4139: (12)                      grid_linestyle='--')
4140: (8)                   fig.draw_without_rendering()
4141: (4)               for ax in fig_test.axes:
4142: (8)                   ax.xaxis.reset_ticks()
4143: (8)                   ax.yaxis.reset_ticks()
4144: (0)           @mpl.style.context('mpl20')
4145: (0)           def test_context_ticks():
4146: (4)               with plt.rc_context({
4147: (12)                      'xtick.direction': 'in', 'xtick.major.size': 30,
'xtick.major.width': 5,
4148: (12)                      'xtick.color': 'C0', 'xtick.major.pad': 12,
4149: (12)                      'xtick.bottom': True, 'xtick.top': True,
4150: (12)                      'xtick.labelsize': 14, 'xtick.labelcolor': 'C1'}):
4151: (8)                   fig, ax = plt.subplots()
```

```
4152: (4)               fig.draw_without_rendering()
4153: (4)               first_tick = ax.xaxis.majorTicks[0]
4154: (4)               for tick in ax.xaxis.majorTicks[1:]:
4155: (8)                   assert tick._size == first_tick._size
4156: (8)                   assert tick._width == first_tick._width
4157: (8)                   assert tick._base_pad == first_tick._base_pad
4158: (8)                   assert tick._labelrotation == first_tick._labelrotation
4159: (8)                   assert tick._zorder == first_tick._zorder
4160: (8)                   assert tick._tickdir == first_tick._tickdir
4161: (0)           def test_vline_limit():
4162: (4)               fig = plt.figure()
4163: (4)               ax = fig.gca()
4164: (4)               ax.axvline(0.5)
4165: (4)               ax.plot([-0.1, 0, 0.2, 0.1])
4166: (4)               assert_allclose(ax.get_ylim(), (-.1, .2))
4167: (0)           @pytest.mark.parametrize('fv, fh, args', [[plt.axvline, plt.axhline, (1,)],
4168: (42)                                                     [plt.axvspan, plt.axhspan, (1, 1)]])
4169: (0)           def test_axline_minmax(fv, fh, args):
4170: (4)               bad_lim = matplotlib.dates.num2date(1)
4171: (4)               with pytest.raises(ValueError, match='ymin must be a single scalar
value'):
4172: (8)                   fv(*args, ymin=bad_lim, ymax=1)
4173: (4)               with pytest.raises(ValueError, match='ymax must be a single scalar
value'):
4174: (8)                   fv(*args, ymin=1, ymax=bad_lim)
4175: (4)               with pytest.raises(ValueError, match='xmin must be a single scalar
value'):
4176: (8)                   fh(*args, xmin=bad_lim, xmax=1)
4177: (4)               with pytest.raises(ValueError, match='xmax must be a single scalar
value'):
4178: (8)                   fh(*args, xmin=1, xmax=bad_lim)
4179: (0)           def test_empty_shared_subplots():
4180: (4)               fig, axs = plt.subplots(nrows=1, ncols=2, sharex=True, sharey=True)
4181: (4)               axs[0].plot([1, 2, 3], [2, 4, 6])
4182: (4)               x0, x1 = axs[1].get_xlim()
4183: (4)               y0, y1 = axs[1].get_ylim()
4184: (4)               assert x0 <= 1
4185: (4)               assert x1 >= 3
4186: (4)               assert y0 <= 2
4187: (4)               assert y1 >= 6
4188: (0)           def test_shared_with_aspect_1():
4189: (4)               for adjustable in ['box', 'datalim']:
4190: (8)                   fig, axs = plt.subplots(nrows=2, sharex=True)
4191: (8)                   axs[0].set_aspect(2, adjustable=adjustable, share=True)
4192: (8)                   assert axs[1].get_aspect() == 2
4193: (8)                   assert axs[1].get_adjustable() == adjustable
4194: (8)                   fig, axs = plt.subplots(nrows=2, sharex=True)
4195: (8)                   axs[0].set_aspect(2, adjustable=adjustable)
4196: (8)                   assert axs[1].get_aspect() == 'auto'
4197: (0)           def test_shared_with_aspect_2():
4198: (4)               fig, axs = plt.subplots(nrows=2, sharex=True, sharey=True)
4199: (4)               axs[0].set_aspect(2, share=True)
4200: (4)               axs[0].plot([1, 2], [3, 4])
4201: (4)               axs[1].plot([3, 4], [1, 2])
4202: (4)               plt.draw()  # Trigger apply_aspect().
4203: (4)               assert axs[0].get_xlim() == axs[1].get_xlim()
4204: (4)               assert axs[0].get_ylim() == axs[1].get_ylim()
4205: (0)           def test_shared_with_aspect_3():
4206: (4)               for adjustable in ['box', 'datalim']:
4207: (8)                   fig, axs = plt.subplots(nrows=2, sharey=True)
4208: (8)                   axs[0].set_aspect(2, adjustable=adjustable)
4209: (8)                   axs[1].set_aspect(0.5, adjustable=adjustable)
4210: (8)                   axs[0].plot([1, 2], [3, 4])
4211: (8)                   axs[1].plot([3, 4], [1, 2])
4212: (8)                   plt.draw()  # Trigger apply_aspect().
4213: (8)                   assert axs[0].get_xlim() != axs[1].get_xlim()
4214: (8)                   assert axs[0].get_ylim() == axs[1].get_ylim()
4215: (8)                   fig_aspect = fig.bbox_inches.height / fig.bbox_inches.width
4216: (8)                   for ax in axs:
```

```
(12)                    p = ax.get_position()
(12)                    box_aspect = p.height / p.width
(12)                    lim_aspect = ax.viewLim.height / ax.viewLim.width
(12)                    expected = fig_aspect * box_aspect / lim_aspect
(12)                    assert round(expected, 4) == round(ax.get_aspect(), 4)
(0)         def test_shared_aspect_error():
(4)             fig, axes = plt.subplots(1, 2, sharex=True, sharey=True)
(4)             axes[0].axis("equal")
(4)             with pytest.raises(RuntimeError, match=r"set_aspect\(..., adjustable="):
(8)                 fig.draw_without_rendering()
(0)         @pytest.mark.parametrize('err, args, kwargs, match',
(25)                                ((TypeError, (1, 2), {},
(27)                                  r"axis\(\) takes from 0 to 1 positional arguments "
(27)                                  "but 2 were given"),
(26)                                 (ValueError, ('foo', ), {},
(27)                                  "Unrecognized string 'foo' to axis; try 'on' or "
(27)                                  "'off'"),
(26)                                 (TypeError, ([1, 2], ), {},
(27)                                  "The first argument to axis*"),
(26)                                 (TypeError, tuple(), {'foo': None},
(27)                                  r"axis\(\) got an unexpected keyword argument "
(27)                                  "'foo'"),
(26)                                 ))
(0)         def test_axis_errors(err, args, kwargs, match):
(4)             with pytest.raises(err, match=match):
(8)                 plt.axis(*args, **kwargs)
(0)         def test_axis_method_errors():
(4)             ax = plt.gca()
(4)             with pytest.raises(ValueError, match="unknown value for which: 'foo'"):
(8)                 ax.get_xaxis_transform('foo')
(4)             with pytest.raises(ValueError, match="unknown value for which: 'foo'"):
(8)                 ax.get_yaxis_transform('foo')
(4)             with pytest.raises(TypeError, match="Cannot supply both positional and"):
(8)                 ax.set_prop_cycle('foo', label='bar')
(4)             with pytest.raises(ValueError, match="argument must be among"):
(8)                 ax.set_anchor('foo')
(4)             with pytest.raises(ValueError, match="scilimits must be a sequence"):
(8)                 ax.ticklabel_format(scilimits=1)
(4)             with pytest.raises(TypeError, match="Specifying 'loc' is disallowed"):
(8)                 ax.set_xlabel('foo', loc='left', x=1)
(4)             with pytest.raises(TypeError, match="Specifying 'loc' is disallowed"):
(8)                 ax.set_ylabel('foo', loc='top', y=1)
(4)             with pytest.raises(TypeError, match="Cannot pass both 'left'"):
(8)                 ax.set_xlim(left=0, xmin=1)
(4)             with pytest.raises(TypeError, match="Cannot pass both 'right'"):
(8)                 ax.set_xlim(right=0, xmax=1)
(4)             with pytest.raises(TypeError, match="Cannot pass both 'bottom'"):
(8)                 ax.set_ylim(bottom=0, ymin=1)
(4)             with pytest.raises(TypeError, match="Cannot pass both 'top'"):
(8)                 ax.set_ylim(top=0, ymax=1)
(0)         @pytest.mark.parametrize('twin', ('x', 'y'))
(0)         def test_twin_with_aspect(twin):
(4)             fig, ax = plt.subplots()
(4)             ax_twin = getattr(ax, f'twin{twin}')()
(4)             ax.set_aspect(5)
(4)             ax_twin.set_aspect(2)
(4)             assert_array_equal(ax.bbox.extents,
(23)                               ax_twin.bbox.extents)
(0)         def test_relim_visible_only():
(4)             x1 = (0., 10.)
(4)             y1 = (0., 10.)
(4)             x2 = (-10., 20.)
(4)             y2 = (-10., 30.)
(4)             fig = matplotlib.figure.Figure()
(4)             ax = fig.add_subplot()
(4)             ax.plot(x1, y1)
(4)             assert ax.get_xlim() == x1
(4)             assert ax.get_ylim() == y1
(4)             line, = ax.plot(x2, y2)
```

```
4286: (4)               assert ax.get_xlim() == x2
4287: (4)               assert ax.get_ylim() == y2
4288: (4)               line.set_visible(False)
4289: (4)               assert ax.get_xlim() == x2
4290: (4)               assert ax.get_ylim() == y2
4291: (4)               ax.relim(visible_only=True)
4292: (4)               ax.autoscale_view()
4293: (4)               assert ax.get_xlim() == x1
4294: (4)               assert ax.get_ylim() == y1
4295: (0)           def test_text_labelsize():
4296: (4)               """
4297: (4)               tests for issue #1172
4298: (4)               """
4299: (4)               fig = plt.figure()
4300: (4)               ax = fig.gca()
4301: (4)               ax.tick_params(labelsize='large')
4302: (4)               ax.tick_params(direction='out')
4303: (0)           @image_comparison(['pie_default.png'], tol=0.01)
4304: (0)           def test_pie_default():
4305: (4)               labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
4306: (4)               sizes = [15, 30, 45, 10]
4307: (4)               colors = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral']
4308: (4)               explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')
4309: (4)               fig1, ax1 = plt.subplots(figsize=(8, 6))
4310: (4)               ax1.pie(sizes, explode=explode, labels=labels, colors=colors,
4311: (12)                      autopct='%1.1f%%', shadow=True, startangle=90)
4312: (0)           @image_comparison(['pie_linewidth_0', 'pie_linewidth_0', 'pie_linewidth_0'],
4313: (18)                        extensions=['png'], style='mpl20', tol=0.01)
4314: (0)           def test_pie_linewidth_0():
4315: (4)               labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
4316: (4)               sizes = [15, 30, 45, 10]
4317: (4)               colors = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral']
4318: (4)               explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')
4319: (4)               plt.pie(sizes, explode=explode, labels=labels, colors=colors,
4320: (12)                      autopct='%1.1f%%', shadow=True, startangle=90,
4321: (12)                      wedgeprops={'linewidth': 0})
4322: (4)               plt.axis('equal')
4323: (4)               data = {"l": labels, "s": sizes, "c": colors, "ex": explode}
4324: (4)               fig = plt.figure()
4325: (4)               ax = fig.gca()
4326: (4)               ax.pie("s", explode="ex", labels="l", colors="c",
4327: (11)                     autopct='%1.1f%%', shadow=True, startangle=90,
4328: (11)                     wedgeprops={'linewidth': 0}, data=data)
4329: (4)               ax.axis('equal')
4330: (4)               plt.figure()
4331: (4)               plt.pie("s", explode="ex", labels="l", colors="c",
4332: (12)                      autopct='%1.1f%%', shadow=True, startangle=90,
4333: (12)                      wedgeprops={'linewidth': 0}, data=data)
4334: (4)               plt.axis('equal')
4335: (0)           @image_comparison(['pie_center_radius.png'], style='mpl20', tol=0.01)
4336: (0)           def test_pie_center_radius():
4337: (4)               labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
4338: (4)               sizes = [15, 30, 45, 10]
4339: (4)               colors = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral']
4340: (4)               explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')
4341: (4)               plt.pie(sizes, explode=explode, labels=labels, colors=colors,
4342: (12)                      autopct='%1.1f%%', shadow=True, startangle=90,
4343: (12)                      wedgeprops={'linewidth': 0}, center=(1, 2), radius=1.5)
4344: (4)               plt.annotate("Center point", xy=(1, 2), xytext=(1, 1.3),
4345: (17)                           arrowprops=dict(arrowstyle="->",
4346: (33)                                           connectionstyle="arc3"),
4347: (17)                           bbox=dict(boxstyle="square", facecolor="lightgrey"))
4348: (4)               plt.axis('equal')
4349: (0)           @image_comparison(['pie_linewidth_2.png'], style='mpl20', tol=0.01)
4350: (0)           def test_pie_linewidth_2():
4351: (4)               labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
4352: (4)               sizes = [15, 30, 45, 10]
4353: (4)               colors = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral']
4354: (4)               explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')
```

```
4355: (4)                plt.pie(sizes, explode=explode, labels=labels, colors=colors,
4356: (12)                       autopct='%1.1f%%', shadow=True, startangle=90,
4357: (12)                       wedgeprops={'linewidth': 2})
4358: (4)                plt.axis('equal')
4359: (0)            @image_comparison(['pie_ccw_true.png'], style='mpl20', tol=0.01)
4360: (0)            def test_pie_ccw_true():
4361: (4)                labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
4362: (4)                sizes = [15, 30, 45, 10]
4363: (4)                colors = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral']
4364: (4)                explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')
4365: (4)                plt.pie(sizes, explode=explode, labels=labels, colors=colors,
4366: (12)                       autopct='%1.1f%%', shadow=True, startangle=90,
4367: (12)                       counterclock=True)
4368: (4)                plt.axis('equal')
4369: (0)            @image_comparison(['pie_frame_grid.png'], style='mpl20', tol=0.002)
4370: (0)            def test_pie_frame_grid():
4371: (4)                labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
4372: (4)                sizes = [15, 30, 45, 10]
4373: (4)                colors = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral']
4374: (4)                explode = (0, 0.1, 0, 0)
4375: (4)                plt.pie(sizes, explode=explode, labels=labels, colors=colors,
4376: (12)                       autopct='%1.1f%%', shadow=True, startangle=90,
4377: (12)                       wedgeprops={'linewidth': 0},
4378: (12)                       frame=True, center=(2, 2))
4379: (4)                plt.pie(sizes[::-1], explode=explode, labels=labels, colors=colors,
4380: (12)                       autopct='%1.1f%%', shadow=True, startangle=90,
4381: (12)                       wedgeprops={'linewidth': 0},
4382: (12)                       frame=True, center=(5, 2))
4383: (4)                plt.pie(sizes, explode=explode[::-1], labels=labels, colors=colors,
4384: (12)                       autopct='%1.1f%%', shadow=True, startangle=90,
4385: (12)                       wedgeprops={'linewidth': 0},
4386: (12)                       frame=True, center=(3, 5))
4387: (4)                plt.axis('equal')
4388: (0)            @image_comparison(['pie_rotatelabels_true.png'], style='mpl20', tol=0.009)
4389: (0)            def test_pie_rotatelabels_true():
4390: (4)                labels = 'Hogwarts', 'Frogs', 'Dogs', 'Logs'
4391: (4)                sizes = [15, 30, 45, 10]
4392: (4)                colors = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral']
4393: (4)                explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')
4394: (4)                plt.pie(sizes, explode=explode, labels=labels, colors=colors,
4395: (12)                       autopct='%1.1f%%', shadow=True, startangle=90,
4396: (12)                       rotatelabels=True)
4397: (4)                plt.axis('equal')
4398: (0)            @image_comparison(['pie_no_label.png'], tol=0.01)
4399: (0)            def test_pie_nolabel_but_legend():
4400: (4)                labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
4401: (4)                sizes = [15, 30, 45, 10]
4402: (4)                colors = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral']
4403: (4)                explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')
4404: (4)                plt.pie(sizes, explode=explode, labels=labels, colors=colors,
4405: (12)                       autopct='%1.1f%%', shadow=True, startangle=90, labeldistance=None,
4406: (12)                       rotatelabels=True)
4407: (4)                plt.axis('equal')
4408: (4)                plt.ylim(-1.2, 1.2)
4409: (4)                plt.legend()
4410: (0)            @image_comparison(['pie_shadow.png'], style='mpl20', tol=0.002)
4411: (0)            def test_pie_shadow():
4412: (4)                sizes = [15, 30, 45, 10]
4413: (4)                colors = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral']
4414: (4)                explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice
4415: (4)                _, axes = plt.subplots(2, 2)
4416: (4)                axes[0][0].pie(sizes, explode=explode, colors=colors,
4417: (19)                              shadow=True, startangle=90,
4418: (19)                              wedgeprops={'linewidth': 0})
4419: (4)                axes[0][1].pie(sizes, explode=explode, colors=colors,
4420: (19)                              shadow=False, startangle=90,
4421: (19)                              wedgeprops={'linewidth': 0})
4422: (4)                axes[1][0].pie(sizes, explode=explode, colors=colors,
4423: (19)                              shadow={'ox': -0.05, 'oy': -0.05, 'shade': 0.9,
```

```
'edgecolor': 'none'},
(19)                                startangle=90, wedgeprops={'linewidth': 0})
(4)                 axes[1][1].pie(sizes, explode=explode, colors=colors,
(19)                                shadow={'ox': 0.05, 'linewidth': 2, 'shade': 0.2},
(19)                                startangle=90, wedgeprops={'linewidth': 0})
(0)             def test_pie_textprops():
(4)                 data = [23, 34, 45]
(4)                 labels = ["Long name 1", "Long name 2", "Long name 3"]
(4)                 textprops = dict(horizontalalignment="center",
(21)                                 verticalalignment="top",
(21)                                 rotation=90,
(21)                                 rotation_mode="anchor",
(21)                                 size=12, color="red")
(4)                 _, texts, autopct = plt.gca().pie(data, labels=labels, autopct='%.2f',
(38)                                                  textprops=textprops)
(4)                 for labels in [texts, autopct]:
(8)                     for tx in labels:
(12)                        assert tx.get_ha() == textprops["horizontalalignment"]
(12)                        assert tx.get_va() == textprops["verticalalignment"]
(12)                        assert tx.get_rotation() == textprops["rotation"]
(12)                        assert tx.get_rotation_mode() == textprops["rotation_mode"]
(12)                        assert tx.get_size() == textprops["size"]
(12)                        assert tx.get_color() == textprops["color"]
(0)             def test_pie_get_negative_values():
(4)                 fig, ax = plt.subplots()
(4)                 with pytest.raises(ValueError):
(8)                     ax.pie([5, 5, -3], explode=[0, .1, .2])
(0)             def test_normalize_kwarg_pie():
(4)                 fig, ax = plt.subplots()
(4)                 x = [0.3, 0.3, 0.1]
(4)                 t1 = ax.pie(x=x, normalize=True)
(4)                 assert abs(t1[0][-1].theta2 - 360.) < 1e-3
(4)                 t2 = ax.pie(x=x, normalize=False)
(4)                 assert abs(t2[0][-1].theta2 - 360.) > 1e-3
(0)             @check_figures_equal()
(0)             def test_pie_hatch_single(fig_test, fig_ref):
(4)                 x = [0.3, 0.3, 0.1]
(4)                 hatch = '+'
(4)                 fig_test.subplots().pie(x, hatch=hatch)
(4)                 wedges, _ = fig_ref.subplots().pie(x)
(4)                 [w.set_hatch(hatch) for w in wedges]
(0)             @check_figures_equal()
(0)             def test_pie_hatch_multi(fig_test, fig_ref):
(4)                 x = [0.3, 0.3, 0.1]
(4)                 hatch = ['/', '+', '.']
(4)                 fig_test.subplots().pie(x, hatch=hatch)
(4)                 wedges, _ = fig_ref.subplots().pie(x)
(4)                 [w.set_hatch(hp) for w, hp in zip(wedges, hatch)]
(0)             @image_comparison(['set_get_ticklabels.png'],
(18)                              tol=0.025 if platform.machine() == 'arm64' else 0)
(0)             def test_set_get_ticklabels():
(4)                 fig, ax = plt.subplots(2)
(4)                 ha = ['normal', 'set_x/yticklabels']
(4)                 ax[0].plot(np.arange(10))
(4)                 ax[0].set_title(ha[0])
(4)                 ax[1].plot(np.arange(10))
(4)                 ax[1].set_title(ha[1])
(4)                 ax[0].set_xticks(range(10))
(4)                 ax[0].set_yticks(range(10))
(4)                 ax[0].set_xticklabels(['a', 'b', 'c', 'd'] + 6 * [''])
(4)                 ax[0].set_yticklabels(['11', '12', '13', '14'] + 6 * [''])
(4)                 ax[1].set_xticks(ax[0].get_xticks())
(4)                 ax[1].set_yticks(ax[0].get_yticks())
(4)                 ax[1].set_xticklabels(ax[0].get_xticklabels())
(4)                 ax[1].set_yticklabels(ax[0].get_yticklabels())
(0)             def test_set_ticks_kwargs_raise_error_without_labels():
(4)                 """
(4)                 When labels=None and any kwarg is passed, axis.set_ticks() raises a
(4)                 ValueError.
```

```
4492: (4)                """
4493: (4)                fig, ax = plt.subplots()
4494: (4)                ticks = [1, 2, 3]
4495: (4)                with pytest.raises(ValueError, match="Incorrect use of keyword argument
'alpha'"):
4496: (8)                    ax.xaxis.set_ticks(ticks, alpha=0.5)
4497: (0)            @check_figures_equal(extensions=["png"])
4498: (0)            def test_set_ticks_with_labels(fig_test, fig_ref):
4499: (4)                """
4500: (4)                Test that these two are identical::
4501: (8)                    set_xticks(ticks); set_xticklabels(labels, **kwargs)
4502: (8)                    set_xticks(ticks, labels, **kwargs)
4503: (4)                """
4504: (4)                ax = fig_ref.subplots()
4505: (4)                ax.set_xticks([1, 2, 4, 6])
4506: (4)                ax.set_xticklabels(['a', 'b', 'c', 'd'], fontweight='bold')
4507: (4)                ax.set_yticks([1, 3, 5])
4508: (4)                ax.set_yticks([2, 4], minor=True)
4509: (4)                ax.set_yticklabels(['A', 'B'], minor=True)
4510: (4)                ax = fig_test.subplots()
4511: (4)                ax.set_xticks([1, 2, 4, 6], ['a', 'b', 'c', 'd'], fontweight='bold')
4512: (4)                ax.set_yticks([1, 3, 5])
4513: (4)                ax.set_yticks([2, 4], ['A', 'B'], minor=True)
4514: (0)            def test_xticks_bad_args():
4515: (4)                ax = plt.figure().add_subplot()
4516: (4)                with pytest.raises(TypeError, match='must be a sequence'):
4517: (8)                    ax.set_xticks([2, 9], 3.1)
4518: (4)                with pytest.raises(ValueError, match='must be 1D'):
4519: (8)                    plt.xticks(np.arange(4).reshape((-1, 1)))
4520: (4)                with pytest.raises(ValueError, match='must be 1D'):
4521: (8)                    plt.xticks(np.arange(4).reshape((1, -1)))
4522: (4)                with pytest.raises(ValueError, match='must be 1D'):
4523: (8)                    plt.xticks(np.arange(4).reshape((-1, 1)), labels=range(4))
4524: (4)                with pytest.raises(ValueError, match='must be 1D'):
4525: (8)                    plt.xticks(np.arange(4).reshape((1, -1)), labels=range(4))
4526: (0)            def test_subsampled_ticklabels():
4527: (4)                fig, ax = plt.subplots()
4528: (4)                ax.plot(np.arange(10))
4529: (4)                ax.xaxis.set_ticks(np.arange(10) + 0.1)
4530: (4)                ax.locator_params(nbins=5)
4531: (4)                ax.xaxis.set_ticklabels([c for c in "bcdefghijk"])
4532: (4)                plt.draw()
4533: (4)                labels = [t.get_text() for t in ax.xaxis.get_ticklabels()]
4534: (4)                assert labels == ['b', 'd', 'f', 'h', 'j']
4535: (0)            def test_mismatched_ticklabels():
4536: (4)                fig, ax = plt.subplots()
4537: (4)                ax.plot(np.arange(10))
4538: (4)                ax.xaxis.set_ticks([1.5, 2.5])
4539: (4)                with pytest.raises(ValueError):
4540: (8)                    ax.xaxis.set_ticklabels(['a', 'b', 'c'])
4541: (0)            def test_empty_ticks_fixed_loc():
4542: (4)                fig, ax = plt.subplots()
4543: (4)                ax.bar([1, 2], [1, 2])
4544: (4)                ax.set_xticks([1, 2])
4545: (4)                ax.set_xticklabels([])
4546: (0)            @image_comparison(['retain_tick_visibility.png'])
4547: (0)            def test_retain_tick_visibility():
4548: (4)                fig, ax = plt.subplots()
4549: (4)                plt.plot([0, 1, 2], [0, -1, 4])
4550: (4)                plt.setp(ax.get_yticklabels(), visible=False)
4551: (4)                ax.tick_params(axis="y", which="both", length=0)
4552: (0)            def test_warn_too_few_labels():
4553: (4)                fig, ax = plt.subplots()
4554: (4)                with pytest.warns(
4555: (11)                       UserWarning,
4556: (11)                       match=r'set_ticklabels\(\) should only be used with a fixed
number'):
4557: (8)                    ax.set_xticklabels(['0', '0.1'])
4558: (4)                fig, ax = plt.subplots()
```

```
4559: (4)               ax.set_xticks([0, 0.5, 1])
4560: (4)               with pytest.raises(ValueError,
4561: (23)                                 match='The number of FixedLocator locations'):
4562: (8)                   ax.set_xticklabels(['0', '0.1'])
4563: (0)           def test_tick_label_update():
4564: (4)               fig, ax = plt.subplots()
4565: (4)               def formatter_func(x, pos):
4566: (8)                   return "unit value" if x == 1 else ""
4567: (4)               ax.xaxis.set_major_formatter(plt.FuncFormatter(formatter_func))
4568: (4)               ax.set_xticks([-1, 0, 1, 2, 3])
4569: (4)               ax.set_xlim(-0.5, 2.5)
4570: (4)               ax.figure.canvas.draw()
4571: (4)               tick_texts = [tick.get_text() for tick in ax.xaxis.get_ticklabels()]
4572: (4)               assert tick_texts == ["", "", "unit value", "", ""]
4573: (0)           @image_comparison(['o_marker_path_snap.png'], savefig_kwarg={'dpi': 72})
4574: (0)           def test_o_marker_path_snap():
4575: (4)               fig, ax = plt.subplots()
4576: (4)               ax.margins(.1)
4577: (4)               for ms in range(1, 15):
4578: (8)                   ax.plot([1, 2, ], np.ones(2) + ms, 'o', ms=ms)
4579: (4)               for ms in np.linspace(1, 10, 25):
4580: (8)                   ax.plot([3, 4, ], np.ones(2) + ms, 'o', ms=ms)
4581: (0)           def test_margins():
4582: (4)               data = [1, 10]
4583: (4)               xmin = 0.0
4584: (4)               xmax = len(data) - 1.0
4585: (4)               ymin = min(data)
4586: (4)               ymax = max(data)
4587: (4)               fig1, ax1 = plt.subplots(1, 1)
4588: (4)               ax1.plot(data)
4589: (4)               ax1.margins(1)
4590: (4)               assert ax1.margins() == (1, 1)
4591: (4)               assert ax1.get_xlim() == (xmin - (xmax - xmin) * 1,
4592: (30)                                        xmax + (xmax - xmin) * 1)
4593: (4)               assert ax1.get_ylim() == (ymin - (ymax - ymin) * 1,
4594: (30)                                        ymax + (ymax - ymin) * 1)
4595: (4)               fig2, ax2 = plt.subplots(1, 1)
4596: (4)               ax2.plot(data)
4597: (4)               ax2.margins(0.5, 2)
4598: (4)               assert ax2.margins() == (0.5, 2)
4599: (4)               assert ax2.get_xlim() == (xmin - (xmax - xmin) * 0.5,
4600: (30)                                        xmax + (xmax - xmin) * 0.5)
4601: (4)               assert ax2.get_ylim() == (ymin - (ymax - ymin) * 2,
4602: (30)                                        ymax + (ymax - ymin) * 2)
4603: (4)               fig3, ax3 = plt.subplots(1, 1)
4604: (4)               ax3.plot(data)
4605: (4)               ax3.margins(x=-0.2, y=0.5)
4606: (4)               assert ax3.margins() == (-0.2, 0.5)
4607: (4)               assert ax3.get_xlim() == (xmin - (xmax - xmin) * -0.2,
4608: (30)                                        xmax + (xmax - xmin) * -0.2)
4609: (4)               assert ax3.get_ylim() == (ymin - (ymax - ymin) * 0.5,
4610: (30)                                        ymax + (ymax - ymin) * 0.5)
4611: (0)           def test_margin_getters():
4612: (4)               fig = plt.figure()
4613: (4)               ax = fig.add_subplot()
4614: (4)               ax.margins(0.2, 0.3)
4615: (4)               assert ax.get_xmargin() == 0.2
4616: (4)               assert ax.get_ymargin() == 0.3
4617: (0)           def test_set_margin_updates_limits():
4618: (4)               mpl.style.use("default")
4619: (4)               fig, ax = plt.subplots()
4620: (4)               ax.plot([1, 2], [1, 2])
4621: (4)               ax.set(xscale="log", xmargin=0)
4622: (4)               assert ax.get_xlim() == (1, 2)
4623: (0)           @pytest.mark.parametrize('err, args, kwargs, match', (
4624: (8)                   (ValueError, (-1,), {}, r'margin must be greater than -0\.5'),
4625: (8)                   (ValueError, (1, -1), {}, r'margin must be greater than -0\.5'),
4626: (8)                   (ValueError, tuple(), {'x': -1}, r'margin must be greater than
-0\.5'),
```

```
4627: (8)                   (ValueError, tuple(), {'y': -1}, r'margin must be greater than
-0\.5'),
4628: (8)                   (TypeError, (1, ), {'x': 1, 'y': 1},
4629: (9)                    'Cannot pass both positional and keyword arguments for x and/or y'),
4630: (8)                   (TypeError, (1, ), {'x': 1},
4631: (9)                    'Cannot pass both positional and keyword arguments for x and/or y'),
4632: (8)                   (TypeError, (1, 1, 1), {}, 'Must pass a single positional argument'),
4633: (0)           ))
4634: (0)           def test_margins_errors(err, args, kwargs, match):
4635: (4)               with pytest.raises(err, match=match):
4636: (8)                   fig = plt.figure()
4637: (8)                   ax = fig.add_subplot()
4638: (8)                   ax.margins(*args, **kwargs)
4639: (0)           def test_length_one_hist():
4640: (4)               fig, ax = plt.subplots()
4641: (4)               ax.hist(1)
4642: (4)               ax.hist([1])
4643: (0)           def test_set_xy_bound():
4644: (4)               fig = plt.figure()
4645: (4)               ax = fig.add_subplot()
4646: (4)               ax.set_xbound(2.0, 3.0)
4647: (4)               assert ax.get_xbound() == (2.0, 3.0)
4648: (4)               assert ax.get_xlim() == (2.0, 3.0)
4649: (4)               ax.set_xbound(upper=4.0)
4650: (4)               assert ax.get_xbound() == (2.0, 4.0)
4651: (4)               assert ax.get_xlim() == (2.0, 4.0)
4652: (4)               ax.set_xbound(lower=3.0)
4653: (4)               assert ax.get_xbound() == (3.0, 4.0)
4654: (4)               assert ax.get_xlim() == (3.0, 4.0)
4655: (4)               ax.set_ybound(2.0, 3.0)
4656: (4)               assert ax.get_ybound() == (2.0, 3.0)
4657: (4)               assert ax.get_ylim() == (2.0, 3.0)
4658: (4)               ax.set_ybound(upper=4.0)
4659: (4)               assert ax.get_ybound() == (2.0, 4.0)
4660: (4)               assert ax.get_ylim() == (2.0, 4.0)
4661: (4)               ax.set_ybound(lower=3.0)
4662: (4)               assert ax.get_ybound() == (3.0, 4.0)
4663: (4)               assert ax.get_ylim() == (3.0, 4.0)
4664: (0)           def test_pathological_hexbin():
4665: (4)               mylist = [10] * 100
4666: (4)               fig, ax = plt.subplots(1, 1)
4667: (4)               ax.hexbin(mylist, mylist)
4668: (4)               fig.savefig(io.BytesIO())  # Check that no warning is emitted.
4669: (0)           def test_color_None():
4670: (4)               fig, ax = plt.subplots()
4671: (4)               ax.plot([1, 2], [1, 2], color=None)
4672: (0)           def test_color_alias():
4673: (4)               fig, ax = plt.subplots()
4674: (4)               line = ax.plot([0, 1], c='lime')[0]
4675: (4)               assert 'lime' == line.get_color()
4676: (0)           def test_numerical_hist_label():
4677: (4)               fig, ax = plt.subplots()
4678: (4)               ax.hist([range(15)] * 5, label=range(5))
4679: (4)               ax.legend()
4680: (0)           def test_unicode_hist_label():
4681: (4)               fig, ax = plt.subplots()
4682: (4)               a = (b'\xe5\xbe\x88\xe6\xbc\x82\xe4\xba\xae, ' +
4683: (9)                    b'r\xc3\xb6m\xc3\xa4n ch\xc3\xa4r\xc3\xa1ct\xc3\xa8rs')
4684: (4)               b = b'\xd7\xa9\xd7\x9c\xd7\x95\xd7\x9d'
4685: (4)               labels = [a.decode('utf-8'),
4686: (14)                         'hi aardvark',
4687: (14)                         b.decode('utf-8'),
4688: (14)                         ]
4689: (4)               ax.hist([range(15)] * 3, label=labels)
4690: (4)               ax.legend()
4691: (0)           def test_move_offsetlabel():
4692: (4)               data = np.random.random(10) * 1e-22
4693: (4)               fig, ax = plt.subplots()
4694: (4)               ax.plot(data)
```

```
(4)             fig.canvas.draw()
(4)             before = ax.yaxis.offsetText.get_position()
(4)             assert ax.yaxis.offsetText.get_horizontalalignment() == 'left'
(4)             ax.yaxis.tick_right()
(4)             fig.canvas.draw()
(4)             after = ax.yaxis.offsetText.get_position()
(4)             assert after[0] > before[0] and after[1] == before[1]
(4)             assert ax.yaxis.offsetText.get_horizontalalignment() == 'right'
(4)             fig, ax = plt.subplots()
(4)             ax.plot(data)
(4)             fig.canvas.draw()
(4)             before = ax.xaxis.offsetText.get_position()
(4)             assert ax.xaxis.offsetText.get_verticalalignment() == 'top'
(4)             ax.xaxis.tick_top()
(4)             fig.canvas.draw()
(4)             after = ax.xaxis.offsetText.get_position()
(4)             assert after[0] == before[0] and after[1] > before[1]
(4)             assert ax.xaxis.offsetText.get_verticalalignment() == 'bottom'
(0)         @image_comparison(['rc_spines.png'], savefig_kwarg={'dpi': 40})
(0)         def test_rc_spines():
(4)             rc_dict = {
(8)                 'axes.spines.left': False,
(8)                 'axes.spines.right': False,
(8)                 'axes.spines.top': False,
(8)                 'axes.spines.bottom': False}
(4)             with matplotlib.rc_context(rc_dict):
(8)                 plt.subplots()  # create a figure and axes with the spine properties
(0)         @image_comparison(['rc_grid.png'], savefig_kwarg={'dpi': 40})
(0)         def test_rc_grid():
(4)             fig = plt.figure()
(4)             rc_dict0 = {
(8)                 'axes.grid': True,
(8)                 'axes.grid.axis': 'both'
(4)             }
(4)             rc_dict1 = {
(8)                 'axes.grid': True,
(8)                 'axes.grid.axis': 'x'
(4)             }
(4)             rc_dict2 = {
(8)                 'axes.grid': True,
(8)                 'axes.grid.axis': 'y'
(4)             }
(4)             dict_list = [rc_dict0, rc_dict1, rc_dict2]
(4)             for i, rc_dict in enumerate(dict_list, 1):
(8)                 with matplotlib.rc_context(rc_dict):
(12)                    fig.add_subplot(3, 1, i)
(0)         def test_rc_tick():
(4)             d = {'xtick.bottom': False, 'xtick.top': True,
(9)                  'ytick.left': True, 'ytick.right': False}
(4)             with plt.rc_context(rc=d):
(8)                 fig = plt.figure()
(8)                 ax1 = fig.add_subplot(1, 1, 1)
(8)                 xax = ax1.xaxis
(8)                 yax = ax1.yaxis
(8)                 assert not xax._major_tick_kw['tick1On']
(8)                 assert xax._major_tick_kw['tick2On']
(8)                 assert not xax._minor_tick_kw['tick1On']
(8)                 assert xax._minor_tick_kw['tick2On']
(8)                 assert yax._major_tick_kw['tick1On']
(8)                 assert not yax._major_tick_kw['tick2On']
(8)                 assert yax._minor_tick_kw['tick1On']
(8)                 assert not yax._minor_tick_kw['tick2On']
(0)         def test_rc_major_minor_tick():
(4)             d = {'xtick.top': True, 'ytick.right': True,  # Enable all ticks
(9)                  'xtick.bottom': True, 'ytick.left': True,
(9)                  'xtick.minor.bottom': False, 'xtick.major.bottom': False,
(9)                  'ytick.major.left': False, 'ytick.minor.left': False}
(4)             with plt.rc_context(rc=d):
(8)                 fig = plt.figure()
```

```
(8)                 ax1 = fig.add_subplot(1, 1, 1)
(8)                 xax = ax1.xaxis
(8)                 yax = ax1.yaxis
(8)                 assert not xax._major_tick_kw['tick1On']
(8)                 assert xax._major_tick_kw['tick2On']
(8)                 assert not xax._minor_tick_kw['tick1On']
(8)                 assert xax._minor_tick_kw['tick2On']
(8)                 assert not yax._major_tick_kw['tick1On']
(8)                 assert yax._major_tick_kw['tick2On']
(8)                 assert not yax._minor_tick_kw['tick1On']
(8)                 assert yax._minor_tick_kw['tick2On']
(0)         def test_square_plot():
(4)             x = np.arange(4)
(4)             y = np.array([1., 3., 5., 7.])
(4)             fig, ax = plt.subplots()
(4)             ax.plot(x, y, 'mo')
(4)             ax.axis('square')
(4)             xlim, ylim = ax.get_xlim(), ax.get_ylim()
(4)             assert np.diff(xlim) == np.diff(ylim)
(4)             assert ax.get_aspect() == 1
(4)             assert_array_almost_equal(
(8)                 ax.get_position(original=True).extents, (0.125, 0.1, 0.9, 0.9))
(4)             assert_array_almost_equal(
(8)                 ax.get_position(original=False).extents, (0.2125, 0.1, 0.8125, 0.9))
(0)         def test_bad_plot_args():
(4)             with pytest.raises(ValueError):
(8)                 plt.plot(None)
(4)             with pytest.raises(ValueError):
(8)                 plt.plot(None, None)
(4)             with pytest.raises(ValueError):
(8)                 plt.plot(np.zeros((2, 2)), np.zeros((2, 3)))
(4)             with pytest.raises(ValueError):
(8)                 plt.plot((np.arange(5).reshape((1, -1)), np.arange(5).reshape(-1, 1)))
(0)         @pytest.mark.parametrize(
(4)             "xy, cls", [
(8)                 ((), mpl.image.AxesImage),  # (0, N)
(8)                 (((3, 7), (2, 6)), mpl.image.AxesImage),  # (xmin, xmax)
(8)                 ((range(5), range(4)), mpl.image.AxesImage),  # regular grid
(8)                 (([1, 2, 4, 8, 16], [0, 1, 2, 3]),  # irregular grid
(9)                  mpl.image.PcolorImage),
(8)                 ((np.random.random((4, 5)), np.random.random((4, 5))),  # 2D coords
(9)                  mpl.collections.QuadMesh),
(4)             ]
(0)         )
(0)         @pytest.mark.parametrize(
(4)             "data", [np.arange(12).reshape((3, 4)), np.random.rand(3, 4, 3)]
(0)         )
(0)         def test_pcolorfast(xy, data, cls):
(4)             fig, ax = plt.subplots()
(4)             assert type(ax.pcolorfast(*xy, data)) == cls
(0)         def test_pcolorfast_bad_dims():
(4)             fig, ax = plt.subplots()
(4)             with pytest.raises(
(12)                    TypeError, match=("the given X was 1D and the given Y was 2D")):
(8)                 ax.pcolorfast(np.empty(6), np.empty((4, 7)), np.empty((8, 8)))
(0)         def test_shared_scale():
(4)             fig, axs = plt.subplots(2, 2, sharex=True, sharey=True)
(4)             axs[0, 0].set_xscale("log")
(4)             axs[0, 0].set_yscale("log")
(4)             for ax in axs.flat:
(8)                 assert ax.get_yscale() == 'log'
(8)                 assert ax.get_xscale() == 'log'
(4)             axs[1, 1].set_xscale("linear")
(4)             axs[1, 1].set_yscale("linear")
(4)             for ax in axs.flat:
(8)                 assert ax.get_yscale() == 'linear'
(8)                 assert ax.get_xscale() == 'linear'
(0)         def test_shared_bool():
(4)             with pytest.raises(TypeError):
```

```
(8)                 plt.subplot(sharex=True)
(4)             with pytest.raises(TypeError):
(8)                 plt.subplot(sharey=True)
(0)         def test_violin_point_mass():
(4)             """Violin plot should handle point mass pdf gracefully."""
(4)             plt.violinplot(np.array([0, 0]))
(0)         def generate_errorbar_inputs():
(4)             base_xy = cycler('x', [np.arange(5)]) + cycler('y', [np.ones(5)])
(4)             err_cycler = cycler('err', [1,
(32)                                         [1, 1, 1, 1, 1],
(32)                                         [[1, 1, 1, 1, 1],
(33)                                          [1, 1, 1, 1, 1]],
(32)                                         np.ones(5),
(32)                                         np.ones((2, 5)),
(32)                                         None
(32)                                         ])
(4)             xerr_cy = cycler('xerr', err_cycler)
(4)             yerr_cy = cycler('yerr', err_cycler)
(4)             empty = ((cycler('x', [[]]) + cycler('y', [[]])) *
(13)                     cycler('xerr', [[], None]) * cycler('yerr', [[], None]))
(4)             xerr_only = base_xy * xerr_cy
(4)             yerr_only = base_xy * yerr_cy
(4)             both_err = base_xy * yerr_cy * xerr_cy
(4)             return [*xerr_only, *yerr_only, *both_err, *empty]
(0)         @pytest.mark.parametrize('kwargs', generate_errorbar_inputs())
(0)         def test_errorbar_inputs_shotgun(kwargs):
(4)             ax = plt.gca()
(4)             eb = ax.errorbar(**kwargs)
(4)             eb.remove()
(0)         @image_comparison(["dash_offset"], remove_text=True)
(0)         def test_dash_offset():
(4)             fig, ax = plt.subplots()
(4)             x = np.linspace(0, 10)
(4)             y = np.ones_like(x)
(4)             for j in range(0, 100, 2):
(8)                 ax.plot(x, j*y, ls=(j, (10, 10)), lw=5, color='k')
(0)         def test_title_pad():
(4)             fig, ax = plt.subplots()
(4)             ax.set_title('aardvark', pad=30.)
(4)             m = ax.titleOffsetTrans.get_matrix()
(4)             assert m[1, -1] == (30. / 72. * fig.dpi)
(4)             ax.set_title('aardvark', pad=0.)
(4)             m = ax.titleOffsetTrans.get_matrix()
(4)             assert m[1, -1] == 0.
(4)             ax.set_title('aardvark', pad=None)
(4)             m = ax.titleOffsetTrans.get_matrix()
(4)             assert m[1, -1] == (matplotlib.rcParams['axes.titlepad'] / 72. * fig.dpi)
(0)         def test_title_location_roundtrip():
(4)             fig, ax = plt.subplots()
(4)             plt.rcParams['axes.titlelocation'] = 'center'
(4)             ax.set_title('aardvark')
(4)             ax.set_title('left', loc='left')
(4)             ax.set_title('right', loc='right')
(4)             assert 'left' == ax.get_title(loc='left')
(4)             assert 'right' == ax.get_title(loc='right')
(4)             assert 'aardvark' == ax.get_title(loc='center')
(4)             with pytest.raises(ValueError):
(8)                 ax.get_title(loc='foo')
(4)             with pytest.raises(ValueError):
(8)                 ax.set_title('fail', loc='foo')
(0)         @pytest.mark.parametrize('sharex', [True, False])
(0)         def test_title_location_shared(sharex):
(4)             fig, axs = plt.subplots(2, 1, sharex=sharex)
(4)             axs[0].set_title('A', pad=-40)
(4)             axs[1].set_title('B', pad=-40)
(4)             fig.draw_without_rendering()
(4)             x, y1 = axs[0].title.get_position()
(4)             x, y2 = axs[1].title.get_position()
(4)             assert y1 == y2 == 1.0
```

```
(0)           @image_comparison(["loglog.png"], remove_text=True, tol=0.02)
(0)           def test_loglog():
(4)               fig, ax = plt.subplots()
(4)               x = np.arange(1, 11)
(4)               ax.loglog(x, x**3, lw=5)
(4)               ax.tick_params(length=25, width=2)
(4)               ax.tick_params(length=15, width=2, which='minor')
(0)           @image_comparison(["test_loglog_nonpos.png"], remove_text=True, style='mpl20',
(18)                            tol=0.029 if platform.machine() == 'arm64' else 0)
(0)           def test_loglog_nonpos():
(4)               fig, axs = plt.subplots(3, 3)
(4)               x = np.arange(1, 11)
(4)               y = x**3
(4)               y[7] = -3.
(4)               x[4] = -10
(4)               for (mcy, mcx), ax in zip(product(['mask', 'clip', ''], repeat=2),
(30)                                        axs.flat):
(8)                   if mcx == mcy:
(12)                      if mcx:
(16)                          ax.loglog(x, y**3, lw=2, nonpositive=mcx)
(12)                      else:
(16)                          ax.loglog(x, y**3, lw=2)
(8)                   else:
(12)                      ax.loglog(x, y**3, lw=2)
(12)                      if mcx:
(16)                          ax.set_xscale("log", nonpositive=mcx)
(12)                      if mcy:
(16)                          ax.set_yscale("log", nonpositive=mcy)
(0)           @mpl.style.context('default')
(0)           def test_axes_margins():
(4)               fig, ax = plt.subplots()
(4)               ax.plot([0, 1, 2, 3])
(4)               assert ax.get_ybound()[0] != 0
(4)               fig, ax = plt.subplots()
(4)               ax.bar([0, 1, 2, 3], [1, 1, 1, 1])
(4)               assert ax.get_ybound()[0] == 0
(4)               fig, ax = plt.subplots()
(4)               ax.barh([0, 1, 2, 3], [1, 1, 1, 1])
(4)               assert ax.get_xbound()[0] == 0
(4)               fig, ax = plt.subplots()
(4)               ax.pcolor(np.zeros((10, 10)))
(4)               assert ax.get_xbound() == (0, 10)
(4)               assert ax.get_ybound() == (0, 10)
(4)               fig, ax = plt.subplots()
(4)               ax.pcolorfast(np.zeros((10, 10)))
(4)               assert ax.get_xbound() == (0, 10)
(4)               assert ax.get_ybound() == (0, 10)
(4)               fig, ax = plt.subplots()
(4)               ax.hist(np.arange(10))
(4)               assert ax.get_ybound()[0] == 0
(4)               fig, ax = plt.subplots()
(4)               ax.imshow(np.zeros((10, 10)))
(4)               assert ax.get_xbound() == (-0.5, 9.5)
(4)               assert ax.get_ybound() == (-0.5, 9.5)
(0)           @pytest.fixture(params=['x', 'y'])
(0)           def shared_axis_remover(request):
(4)               def _helper_x(ax):
(8)                   ax2 = ax.twinx()
(8)                   ax2.remove()
(8)                   ax.set_xlim(0, 15)
(8)                   r = ax.xaxis.get_major_locator()()
(8)                   assert r[-1] > 14
(4)               def _helper_y(ax):
(8)                   ax2 = ax.twiny()
(8)                   ax2.remove()
(8)                   ax.set_ylim(0, 15)
(8)                   r = ax.yaxis.get_major_locator()()
(8)                   assert r[-1] > 14
(4)               return {"x": _helper_x, "y": _helper_y}[request.param]
```

```
4971: (0)           @pytest.fixture(params=['gca', 'subplots', 'subplots_shared', 'add_axes'])
4972: (0)           def shared_axes_generator(request):
4973: (4)               if request.param == 'gca':
4974: (8)                   fig = plt.figure()
4975: (8)                   ax = fig.gca()
4976: (4)               elif request.param == 'subplots':
4977: (8)                   fig, ax = plt.subplots()
4978: (4)               elif request.param == 'subplots_shared':
4979: (8)                   fig, ax_lst = plt.subplots(2, 2, sharex='all', sharey='all')
4980: (8)                   ax = ax_lst[0][0]
4981: (4)               elif request.param == 'add_axes':
4982: (8)                   fig = plt.figure()
4983: (8)                   ax = fig.add_axes([.1, .1, .8, .8])
4984: (4)               return fig, ax
4985: (0)           def test_remove_shared_axes(shared_axes_generator, shared_axis_remover):
4986: (4)               fig, ax = shared_axes_generator
4987: (4)               shared_axis_remover(ax)
4988: (0)           def test_remove_shared_axes_relim():
4989: (4)               fig, ax_lst = plt.subplots(2, 2, sharex='all', sharey='all')
4990: (4)               ax = ax_lst[0][0]
4991: (4)               orig_xlim = ax_lst[0][1].get_xlim()
4992: (4)               ax.remove()
4993: (4)               ax.set_xlim(0, 5)
4994: (4)               assert_array_equal(ax_lst[0][1].get_xlim(), orig_xlim)
4995: (0)           def test_shared_axes_autoscale():
4996: (4)               l = np.arange(-80, 90, 40)
4997: (4)               t = np.random.random_sample((l.size, l.size))
4998: (4)               fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, sharey=True)
4999: (4)               ax1.set_xlim(-1000, 1000)
5000: (4)               ax1.set_ylim(-1000, 1000)
5001: (4)               ax1.contour(l, l, t)
5002: (4)               ax2.contour(l, l, t)
5003: (4)               assert not ax1.get_autoscalex_on() and not ax2.get_autoscalex_on()
5004: (4)               assert not ax1.get_autoscaley_on() and not ax2.get_autoscaley_on()
5005: (4)               assert ax1.get_xlim() == ax2.get_xlim() == (-1000, 1000)
5006: (4)               assert ax1.get_ylim() == ax2.get_ylim() == (-1000, 1000)
5007: (0)           def test_adjust_numtick_aspect():
5008: (4)               fig, ax = plt.subplots()
5009: (4)               ax.yaxis.get_major_locator().set_params(nbins='auto')
5010: (4)               ax.set_xlim(0, 1000)
5011: (4)               ax.set_aspect('equal')
5012: (4)               fig.canvas.draw()
5013: (4)               assert len(ax.yaxis.get_major_locator()()) == 2
5014: (4)               ax.set_ylim(0, 1000)
5015: (4)               fig.canvas.draw()
5016: (4)               assert len(ax.yaxis.get_major_locator()()) > 2
5017: (0)           @mpl.style.context("default")
5018: (0)           def test_auto_numticks():
5019: (4)               axs = plt.figure().subplots(4, 4)
5020: (4)               for ax in axs.flat:  # Tiny, empty subplots have only 3 ticks.
5021: (8)                   assert [*ax.get_xticks()] == [*ax.get_yticks()] == [0, 0.5, 1]
5022: (0)           @mpl.style.context("default")
5023: (0)           def test_auto_numticks_log():
5024: (4)               fig, ax = plt.subplots()
5025: (4)               mpl.rcParams['axes.autolimit_mode'] = 'round_numbers'
5026: (4)               ax.loglog([1e-20, 1e5], [1e-16, 10])
5027: (4)               assert (np.log10(ax.get_xticks()) == np.arange(-26, 18, 4)).all()
5028: (4)               assert (np.log10(ax.get_yticks()) == np.arange(-20, 10, 3)).all()
5029: (0)           def test_broken_barh_empty():
5030: (4)               fig, ax = plt.subplots()
5031: (4)               ax.broken_barh([], (.1, .5))
5032: (0)           def test_broken_barh_timedelta():
5033: (4)               """Check that timedelta works as x, dx pair for this method."""
5034: (4)               fig, ax = plt.subplots()
5035: (4)               d0 = datetime.datetime(2018, 11, 9, 0, 0, 0)
5036: (4)               pp = ax.broken_barh([(d0, datetime.timedelta(hours=1))], [1, 2])
5037: (4)               assert pp.get_paths()[0].vertices[0, 0] == mdates.date2num(d0)
5038: (4)               assert pp.get_paths()[0].vertices[2, 0] == mdates.date2num(d0) + 1 / 24
5039: (0)           def test_pandas_pcolormesh(pd):
```

```
5040: (4)               time = pd.date_range('2000-01-01', periods=10)
5041: (4)               depth = np.arange(20)
5042: (4)               data = np.random.rand(19, 9)
5043: (4)               fig, ax = plt.subplots()
5044: (4)               ax.pcolormesh(time, depth, data)
5045: (0)           def test_pandas_indexing_dates(pd):
5046: (4)               dates = np.arange('2005-02', '2005-03', dtype='datetime64[D]')
5047: (4)               values = np.sin(range(len(dates)))
5048: (4)               df = pd.DataFrame({'dates': dates, 'values': values})
5049: (4)               ax = plt.gca()
5050: (4)               without_zero_index = df[np.array(df.index) % 2 == 1].copy()
5051: (4)               ax.plot('dates', 'values', data=without_zero_index)
5052: (0)           def test_pandas_errorbar_indexing(pd):
5053: (4)               df = pd.DataFrame(np.random.uniform(size=(5, 4)),
5054: (22)                                 columns=['x', 'y', 'xe', 'ye'],
5055: (22)                                 index=[1, 2, 3, 4, 5])
5056: (4)               fig, ax = plt.subplots()
5057: (4)               ax.errorbar('x', 'y', xerr='xe', yerr='ye', data=df)
5058: (0)           def test_pandas_index_shape(pd):
5059: (4)               df = pd.DataFrame({"XX": [4, 5, 6], "YY": [7, 1, 2]})
5060: (4)               fig, ax = plt.subplots()
5061: (4)               ax.plot(df.index, df['YY'])
5062: (0)           def test_pandas_indexing_hist(pd):
5063: (4)               ser_1 = pd.Series(data=[1, 2, 2, 3, 3, 4, 4, 4, 4, 5])
5064: (4)               ser_2 = ser_1.iloc[1:]
5065: (4)               fig, ax = plt.subplots()
5066: (4)               ax.hist(ser_2)
5067: (0)           def test_pandas_bar_align_center(pd):
5068: (4)               df = pd.DataFrame({'a': range(2), 'b': range(2)})
5069: (4)               fig, ax = plt.subplots(1)
5070: (4)               ax.bar(df.loc[df['a'] == 1, 'b'],
5071: (11)                     df.loc[df['a'] == 1, 'b'],
5072: (11)                     align='center')
5073: (4)               fig.canvas.draw()
5074: (0)           def test_axis_get_tick_params():
5075: (4)               axis = plt.subplot().yaxis
5076: (4)               initial_major_style_translated = {**axis.get_tick_params(which='major')}
5077: (4)               initial_minor_style_translated = {**axis.get_tick_params(which='minor')}
5078: (4)               translated_major_kw = axis._translate_tick_params(
5079: (8)                   axis._major_tick_kw, reverse=True
5080: (4)               )
5081: (4)               translated_minor_kw = axis._translate_tick_params(
5082: (8)                   axis._minor_tick_kw, reverse=True
5083: (4)               )
5084: (4)               assert translated_major_kw == initial_major_style_translated
5085: (4)               assert translated_minor_kw == initial_minor_style_translated
5086: (4)               axis.set_tick_params(labelsize=30, labelcolor='red',
5087: (25)                                   direction='out', which='both')
5088: (4)               new_major_style_translated = {**axis.get_tick_params(which='major')}
5089: (4)               new_minor_style_translated = {**axis.get_tick_params(which='minor')}
5090: (4)               new_major_style = axis._translate_tick_params(new_major_style_translated)
5091: (4)               new_minor_style = axis._translate_tick_params(new_minor_style_translated)
5092: (4)               assert initial_major_style_translated != new_major_style_translated
5093: (4)               assert axis._major_tick_kw == new_major_style
5094: (4)               assert initial_minor_style_translated != new_minor_style_translated
5095: (4)               assert axis._minor_tick_kw == new_minor_style
5096: (0)           def test_axis_set_tick_params_labelsize_labelcolor():
5097: (4)               axis_1 = plt.subplot()
5098: (4)               axis_1.yaxis.set_tick_params(labelsize=30, labelcolor='red',
5099: (33)                                           direction='out')
5100: (4)               assert axis_1.yaxis.majorTicks[0]._size == 4.0
5101: (4)               assert axis_1.yaxis.majorTicks[0].tick1line.get_color() == 'k'
5102: (4)               assert axis_1.yaxis.majorTicks[0].label1.get_size() == 30.0
5103: (4)               assert axis_1.yaxis.majorTicks[0].label1.get_color() == 'red'
5104: (0)           def test_axes_tick_params_gridlines():
5105: (4)               ax = plt.subplot()
5106: (4)               ax.tick_params(grid_color='b', grid_linewidth=5, grid_alpha=0.5,
5107: (19)                             grid_linestyle='dashdot')
5108: (4)               for axis in ax.xaxis, ax.yaxis:
```

```
5109: (8)                   assert axis.majorTicks[0].gridline.get_color() == 'b'
5110: (8)                   assert axis.majorTicks[0].gridline.get_linewidth() == 5
5111: (8)                   assert axis.majorTicks[0].gridline.get_alpha() == 0.5
5112: (8)                   assert axis.majorTicks[0].gridline.get_linestyle() == '-.'
5113: (0)           def test_axes_tick_params_ylabelside():
5114: (4)               ax = plt.subplot()
5115: (4)               ax.tick_params(labelleft=False, labelright=True,
5116: (19)                              which='major')
5117: (4)               ax.tick_params(labelleft=False, labelright=True,
5118: (19)                              which='minor')
5119: (4)               assert ax.yaxis.majorTicks[0].label1.get_visible() is False
5120: (4)               assert ax.yaxis.majorTicks[0].label2.get_visible() is True
5121: (4)               assert ax.yaxis.minorTicks[0].label1.get_visible() is False
5122: (4)               assert ax.yaxis.minorTicks[0].label2.get_visible() is True
5123: (0)           def test_axes_tick_params_xlabelside():
5124: (4)               ax = plt.subplot()
5125: (4)               ax.tick_params(labeltop=True, labelbottom=False,
5126: (19)                              which='major')
5127: (4)               ax.tick_params(labeltop=True, labelbottom=False,
5128: (19)                              which='minor')
5129: (4)               assert ax.xaxis.majorTicks[0].label1.get_visible() is False
5130: (4)               assert ax.xaxis.majorTicks[0].label2.get_visible() is True
5131: (4)               assert ax.xaxis.minorTicks[0].label1.get_visible() is False
5132: (4)               assert ax.xaxis.minorTicks[0].label2.get_visible() is True
5133: (0)           def test_none_kwargs():
5134: (4)               ax = plt.figure().subplots()
5135: (4)               ln, = ax.plot(range(32), linestyle=None)
5136: (4)               assert ln.get_linestyle() == '-'
5137: (0)           def test_bar_uint8():
5138: (4)               xs = [0, 1, 2, 3]
5139: (4)               b = plt.bar(np.array(xs, dtype=np.uint8), [2, 3, 4, 5], align="edge")
5140: (4)               for (patch, x) in zip(b.patches, xs):
5141: (8)                   assert patch.xy[0] == x
5142: (0)           @image_comparison(['date_timezone_x.png'], tol=1.0)
5143: (0)           def test_date_timezone_x():
5144: (4)               time_index = [datetime.datetime(2016, 2, 22, hour=x,
5145: (36)
tzinfo=dateutil.tz.gettz('Canada/Eastern'))
5146: (18)                              for x in range(3)]
5147: (4)               plt.figure(figsize=(20, 12))
5148: (4)               plt.subplot(2, 1, 1)
5149: (4)               with pytest.warns(mpl.MatplotlibDeprecationWarning):
5150: (8)                   plt.plot_date(time_index, [3] * 3, tz='Canada/Eastern')
5151: (4)               plt.subplot(2, 1, 2)
5152: (4)               with pytest.warns(mpl.MatplotlibDeprecationWarning):
5153: (8)                   plt.plot_date(time_index, [3] * 3, tz='UTC')
5154: (0)           @image_comparison(['date_timezone_y.png'])
5155: (0)           def test_date_timezone_y():
5156: (4)               time_index = [datetime.datetime(2016, 2, 22, hour=x,
5157: (36)
tzinfo=dateutil.tz.gettz('Canada/Eastern'))
5158: (18)                              for x in range(3)]
5159: (4)               plt.figure(figsize=(20, 12))
5160: (4)               plt.subplot(2, 1, 1)
5161: (4)               with pytest.warns(mpl.MatplotlibDeprecationWarning):
5162: (8)                   plt.plot_date([3] * 3, time_index, tz='Canada/Eastern', xdate=False,
ydate=True)
5163: (4)               plt.subplot(2, 1, 2)
5164: (4)               with pytest.warns(mpl.MatplotlibDeprecationWarning):
5165: (8)                   plt.plot_date([3] * 3, time_index, tz='UTC', xdate=False, ydate=True)
5166: (0)           @image_comparison(['date_timezone_x_and_y.png'], tol=1.0)
5167: (0)           def test_date_timezone_x_and_y():
5168: (4)               UTC = datetime.timezone.utc
5169: (4)               time_index = [datetime.datetime(2016, 2, 22, hour=x, tzinfo=UTC)
5170: (18)                              for x in range(3)]
5171: (4)               plt.figure(figsize=(20, 12))
5172: (4)               plt.subplot(2, 1, 1)
5173: (4)               with pytest.warns(mpl.MatplotlibDeprecationWarning):
5174: (8)                   plt.plot_date(time_index, time_index, tz='UTC', ydate=True)
```

```
(4)             plt.subplot(2, 1, 2)
(4)             with pytest.warns(mpl.MatplotlibDeprecationWarning):
(8)                 plt.plot_date(time_index, time_index, tz='US/Eastern', ydate=True)
(0)         @image_comparison(['axisbelow.png'], remove_text=True)
(0)         def test_axisbelow():
(4)             axs = plt.figure().subplots(ncols=3, sharex=True, sharey=True)
(4)             settings = (False, 'line', True)
(4)             for ax, setting in zip(axs, settings):
(8)                 ax.plot((0, 10), (0, 10), lw=10, color='m')
(8)                 circ = mpatches.Circle((3, 3), color='r')
(8)                 ax.add_patch(circ)
(8)                 ax.grid(color='c', linestyle='-', linewidth=3)
(8)                 ax.tick_params(top=False, bottom=False,
(23)                               left=False, right=False)
(8)                 ax.spines[:].set_visible(False)
(8)                 ax.set_axisbelow(setting)
(8)                 assert ax.get_axisbelow() == setting
(0)         def test_titletwiny():
(4)             plt.style.use('mpl20')
(4)             fig, ax = plt.subplots(dpi=72)
(4)             ax2 = ax.twiny()
(4)             xlabel2 = ax2.set_xlabel('Xlabel2')
(4)             title = ax.set_title('Title')
(4)             fig.canvas.draw()
(4)             renderer = fig.canvas.get_renderer()
(4)             bbox_y0_title = title.get_window_extent(renderer).y0  # bottom of title
(4)             bbox_y1_xlabel2 = xlabel2.get_window_extent(renderer).y1  # top of xlabel2
(4)             y_diff = bbox_y0_title - bbox_y1_xlabel2
(4)             assert np.isclose(y_diff, 3)
(0)         def test_titlesetpos():
(4)             fig, ax = plt.subplots()
(4)             fig.subplots_adjust(top=0.8)
(4)             ax2 = ax.twiny()
(4)             ax.set_xlabel('Xlabel')
(4)             ax2.set_xlabel('Xlabel2')
(4)             ax.set_title('Title')
(4)             pos = (0.5, 1.11)
(4)             ax.title.set_position(pos)
(4)             renderer = fig.canvas.get_renderer()
(4)             ax._update_title_position(renderer)
(4)             assert ax.title.get_position() == pos
(0)         def test_title_xticks_top():
(4)             mpl.rcParams['axes.titley'] = None
(4)             fig, ax = plt.subplots()
(4)             ax.xaxis.set_ticks_position('top')
(4)             ax.set_title('xlabel top')
(4)             fig.canvas.draw()
(4)             assert ax.title.get_position()[1] > 1.04
(0)         def test_title_xticks_top_both():
(4)             mpl.rcParams['axes.titley'] = None
(4)             fig, ax = plt.subplots()
(4)             ax.tick_params(axis="x",
(19)                           bottom=True, top=True, labelbottom=True, labeltop=True)
(4)             ax.set_title('xlabel top')
(4)             fig.canvas.draw()
(4)             assert ax.title.get_position()[1] > 1.04
(0)         @pytest.mark.parametrize(
(4)             'left, center', [
(8)                 ('left', ''),
(8)                 ('', 'center'),
(8)                 ('left', 'center')
(4)             ], ids=[
(8)                 'left title moved',
(8)                 'center title kept',
(8)                 'both titles aligned'
(4)             ]
(0)         )
(0)         def test_title_above_offset(left, center):
(4)             mpl.rcParams['axes.titley'] = None
```

```
5244: (4)                fig, ax = plt.subplots()
5245: (4)                ax.set_ylim(1e11)
5246: (4)                ax.set_title(left, loc='left')
5247: (4)                ax.set_title(center)
5248: (4)                fig.draw_without_rendering()
5249: (4)                if left and not center:
5250: (8)                    assert ax._left_title.get_position()[1] > 1.0
5251: (4)                elif not left and center:
5252: (8)                    assert ax.title.get_position()[1] == 1.0
5253: (4)                else:
5254: (8)                    yleft = ax._left_title.get_position()[1]
5255: (8)                    ycenter = ax.title.get_position()[1]
5256: (8)                    assert yleft > 1.0
5257: (8)                    assert ycenter == yleft
5258: (0)            def test_title_no_move_off_page():
5259: (4)                mpl.rcParams['axes.titley'] = None
5260: (4)                fig = plt.figure()
5261: (4)                ax = fig.add_axes([0.1, -0.5, 0.8, 0.2])
5262: (4)                ax.tick_params(axis="x",
5263: (19)                               bottom=True, top=True, labelbottom=True, labeltop=True)
5264: (4)                tt = ax.set_title('Boo')
5265: (4)                fig.canvas.draw()
5266: (4)                assert tt.get_position()[1] == 1.0
5267: (0)            def test_offset_label_color():
5268: (4)                fig, ax = plt.subplots()
5269: (4)                ax.plot([1.01e9, 1.02e9, 1.03e9])
5270: (4)                ax.yaxis.set_tick_params(labelcolor='red')
5271: (4)                assert ax.yaxis.get_offset_text().get_color() == 'red'
5272: (0)            def test_offset_text_visible():
5273: (4)                fig, ax = plt.subplots()
5274: (4)                ax.plot([1.01e9, 1.02e9, 1.03e9])
5275: (4)                ax.yaxis.set_tick_params(label1On=False, label2On=True)
5276: (4)                assert ax.yaxis.get_offset_text().get_visible()
5277: (4)                ax.yaxis.set_tick_params(label2On=False)
5278: (4)                assert not ax.yaxis.get_offset_text().get_visible()
5279: (0)            def test_large_offset():
5280: (4)                fig, ax = plt.subplots()
5281: (4)                ax.plot((1 + np.array([0, 1.e-12])) * 1.e27)
5282: (4)                fig.canvas.draw()
5283: (0)            def test_barb_units():
5284: (4)                fig, ax = plt.subplots()
5285: (4)                dates = [datetime.datetime(2017, 7, 15, 18, i) for i in range(0, 60, 10)]
5286: (4)                y = np.linspace(0, 5, len(dates))
5287: (4)                u = v = np.linspace(0, 50, len(dates))
5288: (4)                ax.barbs(dates, y, u, v)
5289: (0)            def test_quiver_units():
5290: (4)                fig, ax = plt.subplots()
5291: (4)                dates = [datetime.datetime(2017, 7, 15, 18, i) for i in range(0, 60, 10)]
5292: (4)                y = np.linspace(0, 5, len(dates))
5293: (4)                u = v = np.linspace(0, 50, len(dates))
5294: (4)                ax.quiver(dates, y, u, v)
5295: (0)            def test_bar_color_cycle():
5296: (4)                to_rgb = mcolors.to_rgb
5297: (4)                fig, ax = plt.subplots()
5298: (4)                for j in range(5):
5299: (8)                    ln, = ax.plot(range(3))
5300: (8)                    brs = ax.bar(range(3), range(3))
5301: (8)                    for br in brs:
5302: (12)                       assert to_rgb(ln.get_color()) == to_rgb(br.get_facecolor())
5303: (0)            def test_tick_param_label_rotation():
5304: (4)                fix, (ax, ax2) = plt.subplots(1, 2)
5305: (4)                ax.plot([0, 1], [0, 1])
5306: (4)                ax2.plot([0, 1], [0, 1])
5307: (4)                ax.xaxis.set_tick_params(which='both', rotation=75)
5308: (4)                ax.yaxis.set_tick_params(which='both', rotation=90)
5309: (4)                for text in ax.get_xticklabels(which='both'):
5310: (8)                    assert text.get_rotation() == 75
5311: (4)                for text in ax.get_yticklabels(which='both'):
5312: (8)                    assert text.get_rotation() == 90
```

```
(4)             ax2.tick_params(axis='x', labelrotation=53)
(4)             ax2.tick_params(axis='y', rotation=35)
(4)             for text in ax2.get_xticklabels(which='major'):
(8)                 assert text.get_rotation() == 53
(4)             for text in ax2.get_yticklabels(which='major'):
(8)                 assert text.get_rotation() == 35
(0)         @mpl.style.context('default')
(0)         def test_fillbetween_cycle():
(4)             fig, ax = plt.subplots()
(4)             for j in range(3):
(8)                 cc = ax.fill_between(range(3), range(3))
(8)                 target = mcolors.to_rgba(f'C{j}')
(8)                 assert tuple(cc.get_facecolors().squeeze()) == tuple(target)
(4)             for j in range(3, 6):
(8)                 cc = ax.fill_betweenx(range(3), range(3))
(8)                 target = mcolors.to_rgba(f'C{j}')
(8)                 assert tuple(cc.get_facecolors().squeeze()) == tuple(target)
(4)             target = mcolors.to_rgba('k')
(4)             for al in ['facecolor', 'facecolors', 'color']:
(8)                 cc = ax.fill_between(range(3), range(3), **{al: 'k'})
(8)                 assert tuple(cc.get_facecolors().squeeze()) == tuple(target)
(4)             edge_target = mcolors.to_rgba('k')
(4)             for j, el in enumerate(['edgecolor', 'edgecolors'], start=6):
(8)                 cc = ax.fill_between(range(3), range(3), **{el: 'k'})
(8)                 face_target = mcolors.to_rgba(f'C{j}')
(8)                 assert tuple(cc.get_facecolors().squeeze()) == tuple(face_target)
(8)                 assert tuple(cc.get_edgecolors().squeeze()) == tuple(edge_target)
(0)         def test_log_margins():
(4)             plt.rcParams['axes.autolimit_mode'] = 'data'
(4)             fig, ax = plt.subplots()
(4)             margin = 0.05
(4)             ax.set_xmargin(margin)
(4)             ax.semilogx([10, 100], [10, 100])
(4)             xlim0, xlim1 = ax.get_xlim()
(4)             transform = ax.xaxis.get_transform()
(4)             xlim0t, xlim1t = transform.transform([xlim0, xlim1])
(4)             x0t, x1t = transform.transform([10, 100])
(4)             delta = (x1t - x0t) * margin
(4)             assert_allclose([xlim0t + delta, xlim1t - delta], [x0t, x1t])
(0)         def test_color_length_mismatch():
(4)             N = 5
(4)             x, y = np.arange(N), np.arange(N)
(4)             colors = np.arange(N+1)
(4)             fig, ax = plt.subplots()
(4)             with pytest.raises(ValueError):
(8)                 ax.scatter(x, y, c=colors)
(4)             with pytest.warns(match="argument looks like a single numeric RGB"):
(8)                 ax.scatter(x, y, c=(0.5, 0.5, 0.5))
(4)             ax.scatter(x, y, c=[(0.5, 0.5, 0.5)] * N)
(0)         def test_eventplot_legend():
(4)             plt.eventplot([1.0], label='Label')
(4)             plt.legend()
(0)         @pytest.mark.parametrize('err, args, kwargs, match', (
(8)                 (ValueError, [[1]], {'lineoffsets': []}, 'lineoffsets cannot be
empty'),
(8)                 (ValueError, [[1]], {'linelengths': []}, 'linelengths cannot be
empty'),
(8)                 (ValueError, [[1]], {'linewidths': []}, 'linewidths cannot be empty'),
(8)                 (ValueError, [[1]], {'linestyles': []}, 'linestyles cannot be empty'),
(8)                 (ValueError, [[1]], {'alpha': []}, 'alpha cannot be empty'),
(8)                 (ValueError, [1], {}, 'positions must be one-dimensional'),
(8)                 (ValueError, [[1]], {'lineoffsets': [1, 2]},
(9)                  'lineoffsets and positions are unequal sized sequences'),
(8)                 (ValueError, [[1]], {'linelengths': [1, 2]},
(9)                  'linelengths and positions are unequal sized sequences'),
(8)                 (ValueError, [[1]], {'linewidths': [1, 2]},
(9)                  'linewidths and positions are unequal sized sequences'),
(8)                 (ValueError, [[1]], {'linestyles': [1, 2]},
(9)                  'linestyles and positions are unequal sized sequences'),
```

```
(8)                 (ValueError, [[1]], {'alpha': [1, 2]},
(9)                  'alpha and positions are unequal sized sequences'),
(8)                 (ValueError, [[1]], {'colors': [1, 2]},
(9)                  'colors and positions are unequal sized sequences'),
(0)         ))
(0)         def test_eventplot_errors(err, args, kwargs, match):
(4)             with pytest.raises(err, match=match):
(8)                 plt.eventplot(*args, **kwargs)
(0)         def test_bar_broadcast_args():
(4)             fig, ax = plt.subplots()
(4)             ax.bar(range(4), 1)
(4)             ax.barh(0, 1, left=range(4), height=1)
(4)             rect1, rect2 = ax.bar([0, 1], [0, 1], edgecolor=(.1, .2, .3, .4))
(4)             assert rect1.get_edgecolor() == rect2.get_edgecolor() == (.1, .2, .3, .4)
(0)         def test_invalid_axis_limits():
(4)             plt.plot([0, 1], [0, 1])
(4)             with pytest.raises(ValueError):
(8)                 plt.xlim(np.nan)
(4)             with pytest.raises(ValueError):
(8)                 plt.xlim(np.inf)
(4)             with pytest.raises(ValueError):
(8)                 plt.ylim(np.nan)
(4)             with pytest.raises(ValueError):
(8)                 plt.ylim(np.inf)
(0)         @pytest.mark.parametrize('xscale', ['symlog', 'log'])
(0)         @pytest.mark.parametrize('yscale', ['symlog', 'log'])
(0)         def test_minorticks_on(xscale, yscale):
(4)             ax = plt.subplot()
(4)             ax.plot([1, 2, 3, 4])
(4)             ax.set_xscale(xscale)
(4)             ax.set_yscale(yscale)
(4)             ax.minorticks_on()
(0)         def test_twinx_knows_limits():
(4)             fig, ax = plt.subplots()
(4)             ax.axvspan(1, 2)
(4)             xtwin = ax.twinx()
(4)             xtwin.plot([0, 0.5], [1, 2])
(4)             fig2, ax2 = plt.subplots()
(4)             ax2.axvspan(1, 2)
(4)             ax2.plot([0, 0.5], [1, 2])
(4)             assert_array_equal(xtwin.viewLim.intervalx, ax2.viewLim.intervalx)
(0)         def test_zero_linewidth():
(4)             plt.plot([0, 1], [0, 1], ls='--', lw=0)
(0)         def test_empty_errorbar_legend():
(4)             fig, ax = plt.subplots()
(4)             ax.errorbar([], [], xerr=[], label='empty y')
(4)             ax.errorbar([], [], yerr=[], label='empty x')
(4)             ax.legend()
(0)         @check_figures_equal(extensions=["png"])
(0)         def test_plot_decimal(fig_test, fig_ref):
(4)             x0 = np.arange(-10, 10, 0.3)
(4)             y0 = [5.2 * x ** 3 - 2.1 * x ** 2 + 7.34 * x + 4.5 for x in x0]
(4)             x = [Decimal(i) for i in x0]
(4)             y = [Decimal(i) for i in y0]
(4)             fig_test.subplots().plot(x, y)
(4)             fig_ref.subplots().plot(x0, y0)
(0)         @check_figures_equal(extensions=["png"])
(0)         def test_markerfacecolor_none_alpha(fig_test, fig_ref):
(4)             fig_test.subplots().plot(0, "o", mfc="none", alpha=.5)
(4)             fig_ref.subplots().plot(0, "o", mfc="w", alpha=.5)
(0)         def test_tick_padding_tightbbox():
(4)             """Test that tick padding gets turned off if axis is off"""
(4)             plt.rcParams["xtick.direction"] = "out"
(4)             plt.rcParams["ytick.direction"] = "out"
(4)             fig, ax = plt.subplots()
(4)             bb = ax.get_tightbbox(fig.canvas.get_renderer())
(4)             ax.axis('off')
(4)             bb2 = ax.get_tightbbox(fig.canvas.get_renderer())
(4)             assert bb.x0 < bb2.x0
```

```
(4)             assert bb.y0 < bb2.y0
(0)         def test_inset():
(4)             """
(4)             Ensure that inset_ax argument is indeed optional
(4)             """
(4)             dx, dy = 0.05, 0.05
(4)             y, x = np.mgrid[slice(1, 5 + dy, dy),
(20)                            slice(1, 5 + dx, dx)]
(4)             z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
(4)             fig, ax = plt.subplots()
(4)             ax.pcolormesh(x, y, z[:-1, :-1])
(4)             ax.set_aspect(1.)
(4)             ax.apply_aspect()
(4)             xlim = [1.5, 2.15]
(4)             ylim = [2, 2.5]
(4)             rect = [xlim[0], ylim[0], xlim[1] - xlim[0], ylim[1] - ylim[0]]
(4)             rec, connectors = ax.indicate_inset(bounds=rect)
(4)             assert connectors is None
(4)             fig.canvas.draw()
(4)             xx = np.array([[1.5, 2.],
(19)                           [2.15, 2.5]])
(4)             assert np.all(rec.get_bbox().get_points() == xx)
(0)         def test_zoom_inset():
(4)             dx, dy = 0.05, 0.05
(4)             y, x = np.mgrid[slice(1, 5 + dy, dy),
(20)                            slice(1, 5 + dx, dx)]
(4)             z = np.sin(x)**10 + np.cos(10 + y*x) * np.cos(x)
(4)             fig, ax = plt.subplots()
(4)             ax.pcolormesh(x, y, z[:-1, :-1])
(4)             ax.set_aspect(1.)
(4)             ax.apply_aspect()
(4)             axin1 = ax.inset_axes([0.7, 0.7, 0.35, 0.35])
(4)             axin1.pcolormesh(x, y, z[:-1, :-1])
(4)             axin1.set_xlim([1.5, 2.15])
(4)             axin1.set_ylim([2, 2.5])
(4)             axin1.set_aspect(ax.get_aspect())
(4)             rec, connectors = ax.indicate_inset_zoom(axin1)
(4)             assert len(connectors) == 4
(4)             fig.canvas.draw()
(4)             xx = np.array([[1.5,  2.],
(19)                           [2.15, 2.5]])
(4)             assert np.all(rec.get_bbox().get_points() == xx)
(4)             xx = np.array([[0.6325, 0.692308],
(19)                           [0.8425, 0.907692]])
(4)             np.testing.assert_allclose(
(8)                 axin1.get_position().get_points(), xx, rtol=1e-4)
(0)         @image_comparison(['inset_polar.png'], remove_text=True, style='mpl20')
(0)         def test_inset_polar():
(4)             _, ax = plt.subplots()
(4)             axins = ax.inset_axes([0.5, 0.1, 0.45, 0.45], polar=True)
(4)             assert isinstance(axins, PolarAxes)
(4)             r = np.arange(0, 2, 0.01)
(4)             theta = 2 * np.pi * r
(4)             ax.plot(theta, r)
(4)             axins.plot(theta, r)
(0)         def test_inset_projection():
(4)             _, ax = plt.subplots()
(4)             axins = ax.inset_axes([0.2, 0.2, 0.3, 0.3], projection="hammer")
(4)             assert isinstance(axins, HammerAxes)
(0)         def test_inset_subclass():
(4)             _, ax = plt.subplots()
(4)             axins = ax.inset_axes([0.2, 0.2, 0.3, 0.3], axes_class=AA.Axes)
(4)             assert isinstance(axins, AA.Axes)
(0)         @pytest.mark.parametrize('x_inverted', [False, True])
(0)         @pytest.mark.parametrize('y_inverted', [False, True])
(0)         def test_indicate_inset_inverted(x_inverted, y_inverted):
(4)             """
(4)             Test that the inset lines are correctly located with inverted data axes.
(4)             """
```

```
5518: (4)                fig, (ax1, ax2) = plt.subplots(1, 2)
5519: (4)                x = np.arange(10)
5520: (4)                ax1.plot(x, x, 'o')
5521: (4)                if x_inverted:
5522: (8)                    ax1.invert_xaxis()
5523: (4)                if y_inverted:
5524: (8)                    ax1.invert_yaxis()
5525: (4)                rect, bounds = ax1.indicate_inset([2, 2, 5, 4], ax2)
5526: (4)                lower_left, upper_left, lower_right, upper_right = bounds
5527: (4)                sign_x = -1 if x_inverted else 1
5528: (4)                sign_y = -1 if y_inverted else 1
5529: (4)                assert sign_x * (lower_right.xy2[0] - lower_left.xy2[0]) > 0
5530: (4)                assert sign_x * (upper_right.xy2[0] - upper_left.xy2[0]) > 0
5531: (4)                assert sign_y * (upper_left.xy2[1] - lower_left.xy2[1]) > 0
5532: (4)                assert sign_y * (upper_right.xy2[1] - lower_right.xy2[1]) > 0
5533: (0)            def test_set_position():
5534: (4)                fig, ax = plt.subplots()
5535: (4)                ax.set_aspect(3.)
5536: (4)                ax.set_position([0.1, 0.1, 0.4, 0.4], which='both')
5537: (4)                assert np.allclose(ax.get_position().width, 0.1)
5538: (4)                ax.set_aspect(2.)
5539: (4)                ax.set_position([0.1, 0.1, 0.4, 0.4], which='original')
5540: (4)                assert np.allclose(ax.get_position().width, 0.15)
5541: (4)                ax.set_aspect(3.)
5542: (4)                ax.set_position([0.1, 0.1, 0.4, 0.4], which='active')
5543: (4)                assert np.allclose(ax.get_position().width, 0.1)
5544: (0)            def test_spines_properbbox_after_zoom():
5545: (4)                fig, ax = plt.subplots()
5546: (4)                bb = ax.spines.bottom.get_window_extent(fig.canvas.get_renderer())
5547: (4)                ax._set_view_from_bbox((320, 320, 500, 500), 'in',
5548: (27)                                       None, False, False)
5549: (4)                bb2 = ax.spines.bottom.get_window_extent(fig.canvas.get_renderer())
5550: (4)                np.testing.assert_allclose(bb.get_points(), bb2.get_points(), rtol=1e-6)
5551: (0)            def test_limits_after_scroll_zoom():
5552: (4)                fig, ax = plt.subplots()
5553: (4)                xlim = (-0.5, 0.5)
5554: (4)                ylim = (-1, 2)
5555: (4)                ax.set_xlim(xlim)
5556: (4)                ax.set_ylim(ymin=ylim[0], ymax=ylim[1])
5557: (4)                ax._set_view_from_bbox((200, 200, 1.))
5558: (4)                np.testing.assert_allclose(xlim, ax.get_xlim(), atol=1e-16)
5559: (4)                np.testing.assert_allclose(ylim, ax.get_ylim(), atol=1e-16)
5560: (4)                ax._set_view_from_bbox((200, 200, 2.))
5561: (4)                new_xlim = (-0.3790322580645161, 0.12096774193548387)
5562: (4)                new_ylim = (-0.40625, 1.09375)
5563: (4)                res_xlim = ax.get_xlim()
5564: (4)                res_ylim = ax.get_ylim()
5565: (4)                np.testing.assert_allclose(res_xlim[1] - res_xlim[0], 0.5)
5566: (4)                np.testing.assert_allclose(res_ylim[1] - res_ylim[0], 1.5)
5567: (4)                np.testing.assert_allclose(new_xlim, res_xlim, atol=1e-16)
5568: (4)                np.testing.assert_allclose(new_ylim, res_ylim)
5569: (4)                ax._set_view_from_bbox((200, 200, 0.5))
5570: (4)                res_xlim = ax.get_xlim()
5571: (4)                res_ylim = ax.get_ylim()
5572: (4)                np.testing.assert_allclose(res_xlim[1] - res_xlim[0], 1)
5573: (4)                np.testing.assert_allclose(res_ylim[1] - res_ylim[0], 3)
5574: (4)                np.testing.assert_allclose(xlim, res_xlim, atol=1e-16)
5575: (4)                np.testing.assert_allclose(ylim, res_ylim, atol=1e-16)
5576: (0)            def test_gettightbbox_ignore_nan():
5577: (4)                fig, ax = plt.subplots()
5578: (4)                remove_ticks_and_titles(fig)
5579: (4)                ax.text(np.nan, 1, 'Boo')
5580: (4)                renderer = fig.canvas.get_renderer()
5581: (4)                np.testing.assert_allclose(ax.get_tightbbox(renderer).width, 496)
5582: (0)            def test_scatter_series_non_zero_index(pd):
5583: (4)                ids = range(10, 18)
5584: (4)                x = pd.Series(np.random.uniform(size=8), index=ids)
5585: (4)                y = pd.Series(np.random.uniform(size=8), index=ids)
5586: (4)                c = pd.Series([1, 1, 1, 1, 1, 0, 0, 0], index=ids)
```

```
(4)             plt.scatter(x, y, c)
(0)         def test_scatter_empty_data():
(4)             plt.scatter([], [])
(4)             plt.scatter([], [], s=[], c=[])
(0)         @image_comparison(['annotate_across_transforms.png'], style='mpl20',
remove_text=True,
(18)                            tol=0.025 if platform.machine() == 'arm64' else 0)
(0)         def test_annotate_across_transforms():
(4)             x = np.linspace(0, 10, 200)
(4)             y = np.exp(-x) * np.sin(x)
(4)             fig, ax = plt.subplots(figsize=(3.39, 3))
(4)             ax.plot(x, y)
(4)             axins = ax.inset_axes([0.4, 0.5, 0.3, 0.3])
(4)             axins.set_aspect(0.2)
(4)             axins.xaxis.set_visible(False)
(4)             axins.yaxis.set_visible(False)
(4)             ax.annotate("", xy=(x[150], y[150]), xycoords=ax.transData,
(16)                        xytext=(1, 0), textcoords=axins.transAxes,
(16)                        arrowprops=dict(arrowstyle="->"))
(0)         class _Translation(mtransforms.Transform):
(4)             input_dims = 1
(4)             output_dims = 1
(4)             def __init__(self, dx):
(8)                 self.dx = dx
(4)             def transform(self, values):
(8)                 return values + self.dx
(4)             def inverted(self):
(8)                 return _Translation(-self.dx)
(0)         @image_comparison(['secondary_xy.png'], style='mpl20',
(18)                            tol=0.027 if platform.machine() == 'arm64' else 0)
(0)         def test_secondary_xy():
(4)             fig, axs = plt.subplots(1, 2, figsize=(10, 5), constrained_layout=True)
(4)             def invert(x):
(8)                 with np.errstate(divide='ignore'):
(12)                    return 1 / x
(4)             for nn, ax in enumerate(axs):
(8)                 ax.plot(np.arange(2, 11), np.arange(2, 11))
(8)                 if nn == 0:
(12)                    secax = ax.secondary_xaxis
(8)                 else:
(12)                    secax = ax.secondary_yaxis
(8)                 secax(0.2, functions=(invert, invert))
(8)                 secax(0.4, functions=(lambda x: 2 * x, lambda x: x / 2))
(8)                 secax(0.6, functions=(lambda x: x**2, lambda x: x**(1/2)))
(8)                 secax(0.8)
(8)                 secax("top" if nn == 0 else "right", functions=_Translation(2))
(8)                 secax(6.25, transform=ax.transData)
(0)         def test_secondary_fail():
(4)             fig, ax = plt.subplots()
(4)             ax.plot(np.arange(2, 11), np.arange(2, 11))
(4)             with pytest.raises(ValueError):
(8)                 ax.secondary_xaxis(0.2, functions=(lambda x: 1 / x))
(4)             with pytest.raises(ValueError):
(8)                 ax.secondary_xaxis('right')
(4)             with pytest.raises(ValueError):
(8)                 ax.secondary_yaxis('bottom')
(4)             with pytest.raises(TypeError):
(8)                 ax.secondary_xaxis(0.2, transform='error')
(0)         def test_secondary_resize():
(4)             fig, ax = plt.subplots(figsize=(10, 5))
(4)             ax.plot(np.arange(2, 11), np.arange(2, 11))
(4)             def invert(x):
(8)                 with np.errstate(divide='ignore'):
(12)                    return 1 / x
(4)             ax.secondary_xaxis('top', functions=(invert, invert))
(4)             fig.canvas.draw()
(4)             fig.set_size_inches((7, 4))
(4)             assert_allclose(ax.get_position().extents, [0.125, 0.1, 0.9, 0.9])
(0)         def test_secondary_minorloc():
```

```
5655: (4)               fig, ax = plt.subplots(figsize=(10, 5))
5656: (4)               ax.plot(np.arange(2, 11), np.arange(2, 11))
5657: (4)               def invert(x):
5658: (8)                   with np.errstate(divide='ignore'):
5659: (12)                      return 1 / x
5660: (4)               secax = ax.secondary_xaxis('top', functions=(invert, invert))
5661: (4)               assert isinstance(secax._axis.get_minor_locator(),
5662: (22)                                mticker.NullLocator)
5663: (4)               secax.minorticks_on()
5664: (4)               assert isinstance(secax._axis.get_minor_locator(),
5665: (22)                                mticker.AutoMinorLocator)
5666: (4)               ax.set_xscale('log')
5667: (4)               plt.draw()
5668: (4)               assert isinstance(secax._axis.get_minor_locator(),
5669: (22)                                mticker.LogLocator)
5670: (4)               ax.set_xscale('linear')
5671: (4)               plt.draw()
5672: (4)               assert isinstance(secax._axis.get_minor_locator(),
5673: (22)                                mticker.NullLocator)
5674: (0)           def test_secondary_formatter():
5675: (4)               fig, ax = plt.subplots()
5676: (4)               ax.set_xscale("log")
5677: (4)               secax = ax.secondary_xaxis("top")
5678: (4)               secax.xaxis.set_major_formatter(mticker.ScalarFormatter())
5679: (4)               fig.canvas.draw()
5680: (4)               assert isinstance(
5681: (8)                   secax.xaxis.get_major_formatter(), mticker.ScalarFormatter)
5682: (0)           def test_secondary_repr():
5683: (4)               fig, ax = plt.subplots()
5684: (4)               secax = ax.secondary_xaxis("top")
5685: (4)               assert repr(secax) == '<SecondaryAxis: >'
5686: (0)           @image_comparison(['axis_options.png'], remove_text=True, style='mpl20')
5687: (0)           def test_axis_options():
5688: (4)               fig, axes = plt.subplots(2, 3)
5689: (4)               for i, option in enumerate(('scaled', 'tight', 'image')):
5690: (8)                   axes[0, i].plot((1, 2), (1, 3.2))
5691: (8)                   axes[0, i].axis(option)
5692: (8)                   axes[0, i].add_artist(mpatches.Circle((1.5, 1.5), radius=0.5,
5693: (46)                                                        facecolor='none',
edgecolor='k'))
5694: (8)                   axes[1, i].plot((1, 2.25), (1, 1.75))
5695: (8)                   axes[1, i].axis(option)
5696: (8)                   axes[1, i].add_artist(mpatches.Circle((1.5, 1.25), radius=0.25,
5697: (46)                                                        facecolor='none',
edgecolor='k'))
5698: (0)           def color_boxes(fig, ax):
5699: (4)               """
5700: (4)               Helper for the tests below that test the extents of various Axes elements
5701: (4)               """
5702: (4)               fig.canvas.draw()
5703: (4)               renderer = fig.canvas.get_renderer()
5704: (4)               bbaxis = []
5705: (4)               for nn, axx in enumerate([ax.xaxis, ax.yaxis]):
5706: (8)                   bb = axx.get_tightbbox(renderer)
5707: (8)                   if bb:
5708: (12)                      axisr = mpatches.Rectangle(
5709: (16)                          (bb.x0, bb.y0), width=bb.width, height=bb.height,
5710: (16)                          linewidth=0.7, edgecolor='y', facecolor="none",
transform=None,
5711: (16)                          zorder=3)
5712: (12)                      fig.add_artist(axisr)
5713: (8)                   bbaxis += [bb]
5714: (4)               bbspines = []
5715: (4)               for nn, a in enumerate(['bottom', 'top', 'left', 'right']):
5716: (8)                   bb = ax.spines[a].get_window_extent(renderer)
5717: (8)                   spiner = mpatches.Rectangle(
5718: (12)                      (bb.x0, bb.y0), width=bb.width, height=bb.height,
5719: (12)                      linewidth=0.7, edgecolor="green", facecolor="none",
transform=None,
```

```
(12)                        zorder=3)
(8)                     fig.add_artist(spiner)
(8)                     bbspines += [bb]
(4)                 bb = ax.get_window_extent()
(4)                 rect2 = mpatches.Rectangle(
(8)                     (bb.x0, bb.y0), width=bb.width, height=bb.height,
(8)                     linewidth=1.5, edgecolor="magenta", facecolor="none", transform=None,
(8)                     zorder=2)
(4)                 fig.add_artist(rect2)
(4)                 bbax = bb
(4)                 bb2 = ax.get_tightbbox(renderer)
(4)                 rect2 = mpatches.Rectangle(
(8)                     (bb2.x0, bb2.y0), width=bb2.width, height=bb2.height,
(8)                     linewidth=3, edgecolor="red", facecolor="none", transform=None,
(8)                     zorder=1)
(4)                 fig.add_artist(rect2)
(4)                 bbtb = bb2
(4)                 return bbaxis, bbspines, bbax, bbtb
(0)             def test_normal_axes():
(4)                 with rc_context({'_internal.classic_mode': False}):
(8)                     fig, ax = plt.subplots(dpi=200, figsize=(6, 6))
(8)                     fig.canvas.draw()
(8)                     plt.close(fig)
(8)                     bbaxis, bbspines, bbax, bbtb = color_boxes(fig, ax)
(4)                 target = [
(8)                     [123.375, 75.88888888888886, 983.25, 33.0],
(8)                     [85.51388888888889, 99.99999999999997, 53.375, 993.0]
(4)                 ]
(4)                 for nn, b in enumerate(bbaxis):
(8)                     targetbb = mtransforms.Bbox.from_bounds(*target[nn])
(8)                     assert_array_almost_equal(b.bounds, targetbb.bounds, decimal=2)
(4)                 target = [
(8)                     [150.0, 119.999, 930.0, 11.111],
(8)                     [150.0, 1080.0, 930.0, 0.0],
(8)                     [150.0, 119.9999, 11.111, 960.0],
(8)                     [1068.8888, 119.9999, 11.111, 960.0]
(4)                 ]
(4)                 for nn, b in enumerate(bbspines):
(8)                     targetbb = mtransforms.Bbox.from_bounds(*target[nn])
(8)                     assert_array_almost_equal(b.bounds, targetbb.bounds, decimal=2)
(4)                 target = [150.0, 119.99999999999997, 930.0, 960.0]
(4)                 targetbb = mtransforms.Bbox.from_bounds(*target)
(4)                 assert_array_almost_equal(bbax.bounds, targetbb.bounds, decimal=2)
(4)                 target = [85.5138, 75.88888, 1021.11, 1017.11]
(4)                 targetbb = mtransforms.Bbox.from_bounds(*target)
(4)                 assert_array_almost_equal(bbtb.bounds, targetbb.bounds, decimal=2)
(4)                 axbb = ax.get_position().transformed(fig.transFigure).bounds
(4)                 assert_array_almost_equal(axbb, ax.get_window_extent().bounds, decimal=2)
(0)             def test_nodecorator():
(4)                 with rc_context({'_internal.classic_mode': False}):
(8)                     fig, ax = plt.subplots(dpi=200, figsize=(6, 6))
(8)                     fig.canvas.draw()
(8)                     ax.set(xticklabels=[], yticklabels=[])
(8)                     bbaxis, bbspines, bbax, bbtb = color_boxes(fig, ax)
(4)                 for nn, b in enumerate(bbaxis):
(8)                     assert b is None
(4)                 target = [
(8)                     [150.0, 119.999, 930.0, 11.111],
(8)                     [150.0, 1080.0, 930.0, 0.0],
(8)                     [150.0, 119.9999, 11.111, 960.0],
(8)                     [1068.8888, 119.9999, 11.111, 960.0]
(4)                 ]
(4)                 for nn, b in enumerate(bbspines):
(8)                     targetbb = mtransforms.Bbox.from_bounds(*target[nn])
(8)                     assert_allclose(b.bounds, targetbb.bounds, atol=1e-2)
(4)                 target = [150.0, 119.99999999999997, 930.0, 960.0]
(4)                 targetbb = mtransforms.Bbox.from_bounds(*target)
(4)                 assert_allclose(bbax.bounds, targetbb.bounds, atol=1e-2)
(4)                 target = [150., 120., 930., 960.]
```

```
(4)             targetbb = mtransforms.Bbox.from_bounds(*target)
(4)             assert_allclose(bbtb.bounds, targetbb.bounds, atol=1e-2)
(0)         def test_displaced_spine():
(4)             with rc_context({'_internal.classic_mode': False}):
(8)                 fig, ax = plt.subplots(dpi=200, figsize=(6, 6))
(8)                 ax.set(xticklabels=[], yticklabels=[])
(8)                 ax.spines.bottom.set_position(('axes', -0.1))
(8)                 fig.canvas.draw()
(8)                 bbaxis, bbspines, bbax, bbtb = color_boxes(fig, ax)
(4)             targets = [
(8)                 [150., 24., 930., 11.111111],
(8)                 [150.0, 1080.0, 930.0, 0.0],
(8)                 [150.0, 119.9999, 11.111, 960.0],
(8)                 [1068.8888, 119.9999, 11.111, 960.0]
(4)             ]
(4)             for target, bbspine in zip(targets, bbspines):
(8)                 targetbb = mtransforms.Bbox.from_bounds(*target)
(8)                 assert_allclose(bbspine.bounds, targetbb.bounds, atol=1e-2)
(4)             target = [150.0, 119.99999999999997, 930.0, 960.0]
(4)             targetbb = mtransforms.Bbox.from_bounds(*target)
(4)             assert_allclose(bbax.bounds, targetbb.bounds, atol=1e-2)
(4)             target = [150., 24., 930., 1056.]
(4)             targetbb = mtransforms.Bbox.from_bounds(*target)
(4)             assert_allclose(bbtb.bounds, targetbb.bounds, atol=1e-2)
(0)         def test_tickdirs():
(4)             """
(4)             Switch the tickdirs and make sure the bboxes switch with them
(4)             """
(4)             targets = [[[150.0, 120.0, 930.0, 11.1111],
(16)                        [150.0, 120.0, 11.111, 960.0]],
(15)                       [[150.0, 108.8889, 930.0, 11.111111111111114],
(16)                        [138.889, 120, 11.111, 960.0]],
(15)                       [[150.0, 114.44444444444441, 930.0, 11.111111111111114],
(16)                        [144.44444444444446, 119.999, 11.111, 960.0]]]
(4)             for dnum, dirs in enumerate(['in', 'out', 'inout']):
(8)                 with rc_context({'_internal.classic_mode': False}):
(12)                    fig, ax = plt.subplots(dpi=200, figsize=(6, 6))
(12)                    ax.tick_params(direction=dirs)
(12)                    fig.canvas.draw()
(12)                    bbaxis, bbspines, bbax, bbtb = color_boxes(fig, ax)
(12)                    for nn, num in enumerate([0, 2]):
(16)                        targetbb = mtransforms.Bbox.from_bounds(*targets[dnum][nn])
(16)                        assert_allclose(
(20)                            bbspines[num].bounds, targetbb.bounds, atol=1e-2)
(0)         def test_minor_accountedfor():
(4)             with rc_context({'_internal.classic_mode': False}):
(8)                 fig, ax = plt.subplots(dpi=200, figsize=(6, 6))
(8)                 fig.canvas.draw()
(8)                 ax.tick_params(which='both', direction='out')
(8)                 bbaxis, bbspines, bbax, bbtb = color_boxes(fig, ax)
(8)                 bbaxis, bbspines, bbax, bbtb = color_boxes(fig, ax)
(8)                 targets = [[150.0, 108.88888888888886, 930.0, 11.111111111111114],
(19)                           [138.8889, 119.9999, 11.1111, 960.0]]
(8)                 for n in range(2):
(12)                    targetbb = mtransforms.Bbox.from_bounds(*targets[n])
(12)                    assert_allclose(
(16)                        bbspines[n * 2].bounds, targetbb.bounds, atol=1e-2)
(8)                 fig, ax = plt.subplots(dpi=200, figsize=(6, 6))
(8)                 fig.canvas.draw()
(8)                 ax.tick_params(which='both', direction='out')
(8)                 ax.minorticks_on()
(8)                 ax.tick_params(axis='both', which='minor', length=30)
(8)                 fig.canvas.draw()
(8)                 bbaxis, bbspines, bbax, bbtb = color_boxes(fig, ax)
(8)                 targets = [[150.0, 36.66666666666663, 930.0, 83.33333333333334],
(19)                           [66.6667, 120.0, 83.3333, 960.0]]
(8)                 for n in range(2):
(12)                    targetbb = mtransforms.Bbox.from_bounds(*targets[n])
(12)                    assert_allclose(
```

```
(16)                            bbspines[n * 2].bounds, targetbb.bounds, atol=1e-2)
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_axis_bool_arguments(fig_test, fig_ref):
(4)                 fig_test.add_subplot(211).axis(False)
(4)                 fig_ref.add_subplot(211).axis("off")
(4)                 ax = fig_test.add_subplot(212)
(4)                 ax.axis(False)
(4)                 ax.axis(True)
(4)                 fig_ref.add_subplot(212).axis("on")
(0)             def test_axis_extent_arg():
(4)                 fig, ax = plt.subplots()
(4)                 xmin = 5
(4)                 xmax = 10
(4)                 ymin = 15
(4)                 ymax = 20
(4)                 extent = ax.axis([xmin, xmax, ymin, ymax])
(4)                 assert tuple(extent) == (xmin, xmax, ymin, ymax)
(4)                 assert (xmin, xmax) == ax.get_xlim()
(4)                 assert (ymin, ymax) == ax.get_ylim()
(0)             def test_axis_extent_arg2():
(4)                 fig, ax = plt.subplots()
(4)                 xmin = 5
(4)                 xmax = 10
(4)                 ymin = 15
(4)                 ymax = 20
(4)                 extent = ax.axis(xmin=xmin, xmax=xmax, ymin=ymin, ymax=ymax)
(4)                 assert tuple(extent) == (xmin, xmax, ymin, ymax)
(4)                 assert (xmin, xmax) == ax.get_xlim()
(4)                 assert (ymin, ymax) == ax.get_ylim()
(0)             def test_hist_auto_bins():
(4)                 _, bins, _ = plt.hist([[1, 2, 3], [3, 4, 5, 6]], bins='auto')
(4)                 assert bins[0] <= 1
(4)                 assert bins[-1] >= 6
(0)             def test_hist_nan_data():
(4)                 fig, (ax1, ax2) = plt.subplots(2)
(4)                 data = [1, 2, 3]
(4)                 nan_data = data + [np.nan]
(4)                 bins, edges, _ = ax1.hist(data)
(4)                 with np.errstate(invalid='ignore'):
(8)                     nanbins, nanedges, _ = ax2.hist(nan_data)
(4)                 np.testing.assert_allclose(bins, nanbins)
(4)                 np.testing.assert_allclose(edges, nanedges)
(0)             def test_hist_range_and_density():
(4)                 _, bins, _ = plt.hist(np.random.rand(10), "auto",
(26)                                      range=(0, 1), density=True)
(4)                 assert bins[0] == 0
(4)                 assert bins[-1] == 1
(0)             def test_bar_errbar_zorder():
(4)                 fig, ax = plt.subplots()
(4)                 x = [1, 2, 3]
(4)                 barcont = ax.bar(x=x, height=x, yerr=x, capsize=5, zorder=3)
(4)                 data_line, caplines, barlinecols = barcont.errorbar.lines
(4)                 for bar in barcont.patches:
(8)                     for capline in caplines:
(12)                        assert capline.zorder > bar.zorder
(8)                     for barlinecol in barlinecols:
(12)                        assert barlinecol.zorder > bar.zorder
(0)             def test_set_ticks_inverted():
(4)                 fig, ax = plt.subplots()
(4)                 ax.invert_xaxis()
(4)                 ax.set_xticks([.3, .7])
(4)                 assert ax.get_xlim() == (1, 0)
(4)                 ax.set_xticks([-1])
(4)                 assert ax.get_xlim() == (1, -1)
(0)             def test_aspect_nonlinear_adjustable_box():
(4)                 fig = plt.figure(figsize=(10, 10))  # Square.
(4)                 ax = fig.add_subplot()
(4)                 ax.plot([.4, .6], [.4, .6])  # Set minpos to keep logit happy.
(4)                 ax.set(xscale="log", xlim=(1, 10),
```

```
(11)                    yscale="logit", ylim=(1/11, 1/1001),
(11)                    aspect=1, adjustable="box")
(4)             ax.margins(0)
(4)             pos = fig.transFigure.transform_bbox(ax.get_position())
(4)             assert pos.height / pos.width == pytest.approx(2)
(0)         def test_aspect_nonlinear_adjustable_datalim():
(4)             fig = plt.figure(figsize=(10, 10))  # Square.
(4)             ax = fig.add_axes([.1, .1, .8, .8])  # Square.
(4)             ax.plot([.4, .6], [.4, .6])  # Set minpos to keep logit happy.
(4)             ax.set(xscale="log", xlim=(1, 100),
(11)                    yscale="logit", ylim=(1 / 101, 1 / 11),
(11)                    aspect=1, adjustable="datalim")
(4)             ax.margins(0)
(4)             ax.apply_aspect()
(4)             assert ax.get_xlim() == pytest.approx([1*10**(1/2), 100/10**(1/2)])
(4)             assert ax.get_ylim() == (1 / 101, 1 / 11)
(0)         def test_box_aspect():
(4)             fig1, ax1 = plt.subplots()
(4)             axtwin = ax1.twinx()
(4)             axtwin.plot([12, 344])
(4)             ax1.set_box_aspect(1)
(4)             assert ax1.get_box_aspect() == 1.0
(4)             fig2, ax2 = plt.subplots()
(4)             ax2.margins(0)
(4)             ax2.plot([0, 2], [6, 8])
(4)             ax2.set_aspect("equal", adjustable="box")
(4)             fig1.canvas.draw()
(4)             fig2.canvas.draw()
(4)             bb1 = ax1.get_position()
(4)             bbt = axtwin.get_position()
(4)             bb2 = ax2.get_position()
(4)             assert_array_equal(bb1.extents, bb2.extents)
(4)             assert_array_equal(bbt.extents, bb2.extents)
(0)         def test_box_aspect_custom_position():
(4)             fig1, ax1 = plt.subplots()
(4)             ax1.set_position([0.1, 0.1, 0.9, 0.2])
(4)             fig1.canvas.draw()
(4)             ax1.set_box_aspect(1.)
(4)             fig2, ax2 = plt.subplots()
(4)             ax2.set_box_aspect(1.)
(4)             fig2.canvas.draw()
(4)             ax2.set_position([0.1, 0.1, 0.9, 0.2])
(4)             fig1.canvas.draw()
(4)             fig2.canvas.draw()
(4)             bb1 = ax1.get_position()
(4)             bb2 = ax2.get_position()
(4)             assert_array_equal(bb1.extents, bb2.extents)
(0)         def test_bbox_aspect_axes_init():
(4)             fig, axs = plt.subplots(2, 3, subplot_kw=dict(box_aspect=1),
(28)                                    constrained_layout=True)
(4)             fig.canvas.draw()
(4)             renderer = fig.canvas.get_renderer()
(4)             sizes = []
(4)             for ax in axs.flat:
(8)                 bb = ax.get_window_extent(renderer)
(8)                 sizes.extend([bb.width, bb.height])
(4)             assert_allclose(sizes, sizes[0])
(0)         def test_set_aspect_negative():
(4)             fig, ax = plt.subplots()
(4)             with pytest.raises(ValueError, match="must be finite and positive"):
(8)                 ax.set_aspect(-1)
(4)             with pytest.raises(ValueError, match="must be finite and positive"):
(8)                 ax.set_aspect(0)
(4)             with pytest.raises(ValueError, match="must be finite and positive"):
(8)                 ax.set_aspect(np.inf)
(4)             with pytest.raises(ValueError, match="must be finite and positive"):
(8)                 ax.set_aspect(-np.inf)
(0)         def test_redraw_in_frame():
(4)             fig, ax = plt.subplots(1, 1)
```

```
(4)             ax.plot([1, 2, 3])
(4)             fig.canvas.draw()
(4)             ax.redraw_in_frame()
(0)         def test_invisible_axes_events():
(4)             fig, ax = plt.subplots()
(4)             assert fig.canvas.inaxes((200, 200)) is not None
(4)             ax.set_visible(False)
(4)             assert fig.canvas.inaxes((200, 200)) is None
(0)         def test_xtickcolor_is_not_markercolor():
(4)             plt.rcParams['lines.markeredgecolor'] = 'white'
(4)             ax = plt.axes()
(4)             ticks = ax.xaxis.get_major_ticks()
(4)             for tick in ticks:
(8)                 assert tick.tick1line.get_markeredgecolor() != 'white'
(0)         def test_ytickcolor_is_not_markercolor():
(4)             plt.rcParams['lines.markeredgecolor'] = 'white'
(4)             ax = plt.axes()
(4)             ticks = ax.yaxis.get_major_ticks()
(4)             for tick in ticks:
(8)                 assert tick.tick1line.get_markeredgecolor() != 'white'
(0)         @pytest.mark.parametrize('axis', ('x', 'y'))
(0)         @pytest.mark.parametrize('auto', (True, False, None))
(0)         def test_unautoscale(axis, auto):
(4)             fig, ax = plt.subplots()
(4)             x = np.arange(100)
(4)             y = np.linspace(-.1, .1, 100)
(4)             ax.scatter(y, x)
(4)             get_autoscale_on = getattr(ax, f'get_autoscale{axis}_on')
(4)             set_lim = getattr(ax, f'set_{axis}lim')
(4)             get_lim = getattr(ax, f'get_{axis}lim')
(4)             post_auto = get_autoscale_on() if auto is None else auto
(4)             set_lim((-0.5, 0.5), auto=auto)
(4)             assert post_auto == get_autoscale_on()
(4)             fig.canvas.draw()
(4)             assert_array_equal(get_lim(), (-0.5, 0.5))
(0)         @check_figures_equal(extensions=["png"])
(0)         def test_polar_interpolation_steps_variable_r(fig_test, fig_ref):
(4)             l, = fig_test.add_subplot(projection="polar").plot([0, np.pi/2], [1, 2])
(4)             l.get_path()._interpolation_steps = 100
(4)             fig_ref.add_subplot(projection="polar").plot(
(8)                 np.linspace(0, np.pi/2, 101), np.linspace(1, 2, 101))
(0)         @mpl.style.context('default')
(0)         def test_autoscale_tiny_sticky():
(4)             fig, ax = plt.subplots()
(4)             ax.bar(0, 1e-9)
(4)             fig.canvas.draw()
(4)             assert ax.get_ylim() == (0, 1.05e-9)
(0)         def test_xtickcolor_is_not_xticklabelcolor():
(4)             plt.rcParams['xtick.color'] = 'yellow'
(4)             plt.rcParams['xtick.labelcolor'] = 'blue'
(4)             ax = plt.axes()
(4)             ticks = ax.xaxis.get_major_ticks()
(4)             for tick in ticks:
(8)                 assert tick.tick1line.get_color() == 'yellow'
(8)                 assert tick.label1.get_color() == 'blue'
(0)         def test_ytickcolor_is_not_yticklabelcolor():
(4)             plt.rcParams['ytick.color'] = 'yellow'
(4)             plt.rcParams['ytick.labelcolor'] = 'blue'
(4)             ax = plt.axes()
(4)             ticks = ax.yaxis.get_major_ticks()
(4)             for tick in ticks:
(8)                 assert tick.tick1line.get_color() == 'yellow'
(8)                 assert tick.label1.get_color() == 'blue'
(0)         def test_xaxis_offsetText_color():
(4)             plt.rcParams['xtick.labelcolor'] = 'blue'
(4)             ax = plt.axes()
(4)             assert ax.xaxis.offsetText.get_color() == 'blue'
(4)             plt.rcParams['xtick.color'] = 'yellow'
(4)             plt.rcParams['xtick.labelcolor'] = 'inherit'
```

```
6065: (4)               ax = plt.axes()
6066: (4)               assert ax.xaxis.offsetText.get_color() == 'yellow'
6067: (0)           def test_yaxis_offsetText_color():
6068: (4)               plt.rcParams['ytick.labelcolor'] = 'green'
6069: (4)               ax = plt.axes()
6070: (4)               assert ax.yaxis.offsetText.get_color() == 'green'
6071: (4)               plt.rcParams['ytick.color'] = 'red'
6072: (4)               plt.rcParams['ytick.labelcolor'] = 'inherit'
6073: (4)               ax = plt.axes()
6074: (4)               assert ax.yaxis.offsetText.get_color() == 'red'
6075: (0)           @pytest.mark.parametrize('size', [size for size in mfont_manager.font_scalings
6076: (34)                                            if size is not None] + [8, 10, 12])
6077: (0)           @mpl.style.context('default')
6078: (0)           def test_relative_ticklabel_sizes(size):
6079: (4)               mpl.rcParams['xtick.labelsize'] = size
6080: (4)               mpl.rcParams['ytick.labelsize'] = size
6081: (4)               fig, ax = plt.subplots()
6082: (4)               fig.canvas.draw()
6083: (4)               for name, axis in zip(['x', 'y'], [ax.xaxis, ax.yaxis]):
6084: (8)                   for tick in axis.get_major_ticks():
6085: (12)                      assert tick.label1.get_size() == axis._get_tick_label_size(name)
6086: (0)           def test_multiplot_autoscale():
6087: (4)               fig = plt.figure()
6088: (4)               ax1, ax2 = fig.subplots(2, 1, sharex='all')
6089: (4)               ax1.plot([18000, 18250, 18500, 18750], [2, 3, 2, 3])
6090: (4)               ax2.axhspan(-5, 5)
6091: (4)               xlim = ax1.get_xlim()
6092: (4)               assert np.allclose(xlim, [18000, 18800])
6093: (0)           def test_sharing_does_not_link_positions():
6094: (4)               fig = plt.figure()
6095: (4)               ax0 = fig.add_subplot(221)
6096: (4)               ax1 = fig.add_axes([.6, .6, .3, .3], sharex=ax0)
6097: (4)               init_pos = ax1.get_position()
6098: (4)               fig.subplots_adjust(left=0)
6099: (4)               assert (ax1.get_position().get_points() == init_pos.get_points()).all()
6100: (0)           @check_figures_equal(extensions=["pdf"])
6101: (0)           def test_2dcolor_plot(fig_test, fig_ref):
6102: (4)               color = np.array([0.1, 0.2, 0.3])
6103: (4)               axs = fig_test.subplots(5)
6104: (4)               axs[0].plot([1, 2], [1, 2], c=color.reshape(-1))
6105: (4)               with pytest.warns(match="argument looks like a single numeric RGB"):
6106: (8)                   axs[1].scatter([1, 2], [1, 2], c=color.reshape(-1))
6107: (4)               axs[2].step([1, 2], [1, 2], c=color.reshape(-1))
6108: (4)               axs[3].hist(np.arange(10), color=color.reshape(-1))
6109: (4)               axs[4].bar(np.arange(10), np.arange(10), color=color.reshape(-1))
6110: (4)               axs = fig_ref.subplots(5)
6111: (4)               axs[0].plot([1, 2], [1, 2], c=color.reshape((1, -1)))
6112: (4)               axs[1].scatter([1, 2], [1, 2], c=color.reshape((1, -1)))
6113: (4)               axs[2].step([1, 2], [1, 2], c=color.reshape((1, -1)))
6114: (4)               axs[3].hist(np.arange(10), color=color.reshape((1, -1)))
6115: (4)               axs[4].bar(np.arange(10), np.arange(10), color=color.reshape((1, -1)))
6116: (0)           @check_figures_equal(extensions=['png'])
6117: (0)           def test_shared_axes_clear(fig_test, fig_ref):
6118: (4)               x = np.arange(0.0, 2*np.pi, 0.01)
6119: (4)               y = np.sin(x)
6120: (4)               axs = fig_ref.subplots(2, 2, sharex=True, sharey=True)
6121: (4)               for ax in axs.flat:
6122: (8)                   ax.plot(x, y)
6123: (4)               axs = fig_test.subplots(2, 2, sharex=True, sharey=True)
6124: (4)               for ax in axs.flat:
6125: (8)                   ax.clear()
6126: (8)                   ax.plot(x, y)
6127: (0)           def test_shared_axes_retick():
6128: (4)               fig, axs = plt.subplots(2, 2, sharex='all', sharey='all')
6129: (4)               for ax in axs.flat:
6130: (8)                   ax.plot([0, 2], 'o-')
6131: (4)               axs[0, 0].set_xticks([-0.5, 0, 1, 1.5])  # should affect all axes xlims
6132: (4)               for ax in axs.flat:
6133: (8)                   assert ax.get_xlim() == axs[0, 0].get_xlim()
```

```
(4)             axs[0, 0].set_yticks([-0.5, 0, 2, 2.5])  # should affect all axes ylims
(4)             for ax in axs.flat:
(8)                 assert ax.get_ylim() == axs[0, 0].get_ylim()
(0)         @pytest.mark.parametrize('ha', ['left', 'center', 'right'])
(0)         def test_ylabel_ha_with_position(ha):
(4)             fig = Figure()
(4)             ax = fig.subplots()
(4)             ax.set_ylabel("test", y=1, ha=ha)
(4)             ax.yaxis.set_label_position("right")
(4)             assert ax.yaxis.get_label().get_ha() == ha
(0)         def test_bar_label_location_vertical():
(4)             ax = plt.gca()
(4)             xs, heights = [1, 2], [3, -4]
(4)             rects = ax.bar(xs, heights)
(4)             labels = ax.bar_label(rects)
(4)             assert labels[0].xy == (xs[0], heights[0])
(4)             assert labels[0].get_horizontalalignment() == 'center'
(4)             assert labels[0].get_verticalalignment() == 'bottom'
(4)             assert labels[1].xy == (xs[1], heights[1])
(4)             assert labels[1].get_horizontalalignment() == 'center'
(4)             assert labels[1].get_verticalalignment() == 'top'
(0)         def test_bar_label_location_vertical_yinverted():
(4)             ax = plt.gca()
(4)             ax.invert_yaxis()
(4)             xs, heights = [1, 2], [3, -4]
(4)             rects = ax.bar(xs, heights)
(4)             labels = ax.bar_label(rects)
(4)             assert labels[0].xy == (xs[0], heights[0])
(4)             assert labels[0].get_horizontalalignment() == 'center'
(4)             assert labels[0].get_verticalalignment() == 'top'
(4)             assert labels[1].xy == (xs[1], heights[1])
(4)             assert labels[1].get_horizontalalignment() == 'center'
(4)             assert labels[1].get_verticalalignment() == 'bottom'
(0)         def test_bar_label_location_horizontal():
(4)             ax = plt.gca()
(4)             ys, widths = [1, 2], [3, -4]
(4)             rects = ax.barh(ys, widths)
(4)             labels = ax.bar_label(rects)
(4)             assert labels[0].xy == (widths[0], ys[0])
(4)             assert labels[0].get_horizontalalignment() == 'left'
(4)             assert labels[0].get_verticalalignment() == 'center'
(4)             assert labels[1].xy == (widths[1], ys[1])
(4)             assert labels[1].get_horizontalalignment() == 'right'
(4)             assert labels[1].get_verticalalignment() == 'center'
(0)         def test_bar_label_location_horizontal_yinverted():
(4)             ax = plt.gca()
(4)             ax.invert_yaxis()
(4)             ys, widths = [1, 2], [3, -4]
(4)             rects = ax.barh(ys, widths)
(4)             labels = ax.bar_label(rects)
(4)             assert labels[0].xy == (widths[0], ys[0])
(4)             assert labels[0].get_horizontalalignment() == 'left'
(4)             assert labels[0].get_verticalalignment() == 'center'
(4)             assert labels[1].xy == (widths[1], ys[1])
(4)             assert labels[1].get_horizontalalignment() == 'right'
(4)             assert labels[1].get_verticalalignment() == 'center'
(0)         def test_bar_label_location_horizontal_xinverted():
(4)             ax = plt.gca()
(4)             ax.invert_xaxis()
(4)             ys, widths = [1, 2], [3, -4]
(4)             rects = ax.barh(ys, widths)
(4)             labels = ax.bar_label(rects)
(4)             assert labels[0].xy == (widths[0], ys[0])
(4)             assert labels[0].get_horizontalalignment() == 'right'
(4)             assert labels[0].get_verticalalignment() == 'center'
(4)             assert labels[1].xy == (widths[1], ys[1])
(4)             assert labels[1].get_horizontalalignment() == 'left'
(4)             assert labels[1].get_verticalalignment() == 'center'
(0)         def test_bar_label_location_horizontal_xyinverted():
```

```
(4)             ax = plt.gca()
(4)             ax.invert_xaxis()
(4)             ax.invert_yaxis()
(4)             ys, widths = [1, 2], [3, -4]
(4)             rects = ax.barh(ys, widths)
(4)             labels = ax.bar_label(rects)
(4)             assert labels[0].xy == (widths[0], ys[0])
(4)             assert labels[0].get_horizontalalignment() == 'right'
(4)             assert labels[0].get_verticalalignment() == 'center'
(4)             assert labels[1].xy == (widths[1], ys[1])
(4)             assert labels[1].get_horizontalalignment() == 'left'
(4)             assert labels[1].get_verticalalignment() == 'center'
(0)         def test_bar_label_location_center():
(4)             ax = plt.gca()
(4)             ys, widths = [1, 2], [3, -4]
(4)             rects = ax.barh(ys, widths)
(4)             labels = ax.bar_label(rects, label_type='center')
(4)             assert labels[0].xy == (0.5, 0.5)
(4)             assert labels[0].get_horizontalalignment() == 'center'
(4)             assert labels[0].get_verticalalignment() == 'center'
(4)             assert labels[1].xy == (0.5, 0.5)
(4)             assert labels[1].get_horizontalalignment() == 'center'
(4)             assert labels[1].get_verticalalignment() == 'center'
(0)         @image_comparison(['test_centered_bar_label_nonlinear.svg'])
(0)         def test_centered_bar_label_nonlinear():
(4)             _, ax = plt.subplots()
(4)             bar_container = ax.barh(['c', 'b', 'a'], [1_000, 5_000, 7_000])
(4)             ax.set_xscale('log')
(4)             ax.set_xlim(1, None)
(4)             ax.bar_label(bar_container, label_type='center')
(4)             ax.set_axis_off()
(0)         def test_centered_bar_label_label_beyond_limits():
(4)             fig, ax = plt.subplots()
(4)             last = 0
(4)             for label, value in zip(['a', 'b', 'c'], [10, 20, 50]):
(8)                 bar_container = ax.barh('col', value, label=label, left=last)
(8)                 ax.bar_label(bar_container, label_type='center')
(8)                 last += value
(4)             ax.set_xlim(None, 20)
(4)             fig.draw_without_rendering()
(0)         def test_bar_label_location_errorbars():
(4)             ax = plt.gca()
(4)             xs, heights = [1, 2], [3, -4]
(4)             rects = ax.bar(xs, heights, yerr=1)
(4)             labels = ax.bar_label(rects)
(4)             assert labels[0].xy == (xs[0], heights[0] + 1)
(4)             assert labels[0].get_horizontalalignment() == 'center'
(4)             assert labels[0].get_verticalalignment() == 'bottom'
(4)             assert labels[1].xy == (xs[1], heights[1] - 1)
(4)             assert labels[1].get_horizontalalignment() == 'center'
(4)             assert labels[1].get_verticalalignment() == 'top'
(0)         @pytest.mark.parametrize('fmt', [
(4)             '%.2f', '{:.2f}', '{:.2f}'.format
(0)         ])
(0)         def test_bar_label_fmt(fmt):
(4)             ax = plt.gca()
(4)             rects = ax.bar([1, 2], [3, -4])
(4)             labels = ax.bar_label(rects, fmt=fmt)
(4)             assert labels[0].get_text() == '3.00'
(4)             assert labels[1].get_text() == '-4.00'
(0)         def test_bar_label_fmt_error():
(4)             ax = plt.gca()
(4)             rects = ax.bar([1, 2], [3, -4])
(4)             with pytest.raises(TypeError, match='str or callable'):
(8)                 _ = ax.bar_label(rects, fmt=10)
(0)         def test_bar_label_labels():
(4)             ax = plt.gca()
(4)             rects = ax.bar([1, 2], [3, -4])
(4)             labels = ax.bar_label(rects, labels=['A', 'B'])
```

```
(4)             assert labels[0].get_text() == 'A'
(4)             assert labels[1].get_text() == 'B'
(0)         def test_bar_label_nan_ydata():
(4)             ax = plt.gca()
(4)             bars = ax.bar([2, 3], [np.nan, 1])
(4)             labels = ax.bar_label(bars)
(4)             assert [l.get_text() for l in labels] == ['', '1']
(4)             assert labels[0].xy == (2, 0)
(4)             assert labels[0].get_verticalalignment() == 'bottom'
(0)         def test_bar_label_nan_ydata_inverted():
(4)             ax = plt.gca()
(4)             ax.yaxis_inverted()
(4)             bars = ax.bar([2, 3], [np.nan, 1])
(4)             labels = ax.bar_label(bars)
(4)             assert [l.get_text() for l in labels] == ['', '1']
(4)             assert labels[0].xy == (2, 0)
(4)             assert labels[0].get_verticalalignment() == 'bottom'
(0)         def test_nan_barlabels():
(4)             fig, ax = plt.subplots()
(4)             bars = ax.bar([1, 2, 3], [np.nan, 1, 2], yerr=[0.2, 0.4, 0.6])
(4)             labels = ax.bar_label(bars)
(4)             assert [l.get_text() for l in labels] == ['', '1', '2']
(4)             assert np.allclose(ax.get_ylim(), (0.0, 3.0))
(4)             fig, ax = plt.subplots()
(4)             bars = ax.bar([1, 2, 3], [0, 1, 2], yerr=[0.2, np.nan, 0.6])
(4)             labels = ax.bar_label(bars)
(4)             assert [l.get_text() for l in labels] == ['0', '1', '2']
(4)             assert np.allclose(ax.get_ylim(), (-0.5, 3.0))
(4)             fig, ax = plt.subplots()
(4)             bars = ax.bar([1, 2, 3], [np.nan, 1, 2], yerr=[np.nan, np.nan, 0.6])
(4)             labels = ax.bar_label(bars)
(4)             assert [l.get_text() for l in labels] == ['', '1', '2']
(4)             assert np.allclose(ax.get_ylim(), (0.0, 3.0))
(0)         def test_patch_bounds():  # PR 19078
(4)             fig, ax = plt.subplots()
(4)             ax.add_patch(mpatches.Wedge((0, -1), 1.05, 60, 120, width=0.1))
(4)             bot = 1.9*np.sin(15*np.pi/180)**2
(4)             np.testing.assert_array_almost_equal_nulp(
(8)                 np.array((-0.525, -(bot+0.05), 1.05, bot+0.1)), ax.dataLim.bounds, 16)
(0)         @mpl.style.context('default')
(0)         def test_warn_ignored_scatter_kwargs():
(4)             with pytest.warns(UserWarning,
(22)                              match=r"You passed a edgecolor/edgecolors"):
(8)                 plt.scatter([0], [0], marker="+", s=500, facecolor="r", edgecolor="b")
(0)         def test_artist_sublists():
(4)             fig, ax = plt.subplots()
(4)             lines = [ax.plot(np.arange(i, i + 5))[0] for i in range(6)]
(4)             col = ax.scatter(np.arange(5), np.arange(5))
(4)             im = ax.imshow(np.zeros((5, 5)))
(4)             patch = ax.add_patch(mpatches.Rectangle((0, 0), 5, 5))
(4)             text = ax.text(0, 0, 'foo')
(4)             assert list(ax.collections) == [col]
(4)             assert list(ax.images) == [im]
(4)             assert list(ax.lines) == lines
(4)             assert list(ax.patches) == [patch]
(4)             assert not ax.tables
(4)             assert list(ax.texts) == [text]
(4)             assert ax.lines[0] is lines[0]
(4)             assert ax.lines[-1] is lines[-1]
(4)             with pytest.raises(IndexError, match='out of range'):
(8)                 ax.lines[len(lines) + 1]
(4)             assert ax.lines + [1, 2, 3] == [*lines, 1, 2, 3]
(4)             assert [1, 2, 3] + ax.lines == [1, 2, 3, *lines]
(4)             assert ax.lines + (1, 2, 3) == (*lines, 1, 2, 3)
(4)             assert (1, 2, 3) + ax.lines == (1, 2, 3, *lines)
(4)             col.remove()
(4)             assert not ax.collections
(4)             im.remove()
(4)             assert not ax.images
```

```
6341: (4)                patch.remove()
6342: (4)                assert not ax.patches
6343: (4)                assert not ax.tables
6344: (4)                text.remove()
6345: (4)                assert not ax.texts
6346: (4)                for ln in ax.lines:
6347: (8)                    ln.remove()
6348: (4)                assert len(ax.lines) == 0
6349: (0)            def test_empty_line_plots():
6350: (4)                x = np.ones(10)
6351: (4)                y = np.ones((10, 0))
6352: (4)                _, ax = plt.subplots()
6353: (4)                line = ax.plot(x, y)
6354: (4)                assert len(line) == 0
6355: (4)                _, ax = plt.subplots()
6356: (4)                line = ax.plot([], [])
6357: (4)                assert len(line) == 1
6358: (0)            @pytest.mark.parametrize('fmt, match', (
6359: (4)                ("f", r"'f' is not a valid format string \(unrecognized character 'f'\)"),
6360: (4)                ("o+", r"'o\+' is not a valid format string \(two marker symbols\)"),
6361: (4)                (":-", r":-' is not a valid format string \(two linestyle symbols\)"),
6362: (4)                ("rk", r"'rk' is not a valid format string \(two color symbols\)"),
6363: (4)                (":o-r", r":o-r' is not a valid format string \(two linestyle
symbols\)"),
6364: (4)                ("C", r"'C' is not a valid format string \('C' must be followed by a
number\)"),
6365: (4)                (".C", r".C' is not a valid format string \('C' must be followed by a
number\)"),
6366: (0)            ))
6367: (0)            @pytest.mark.parametrize("data", [None, {"string": range(3)}])
6368: (0)            def test_plot_format_errors(fmt, match, data):
6369: (4)                fig, ax = plt.subplots()
6370: (4)                if data is not None:
6371: (8)                    match = match.replace("not", "neither a data key nor")
6372: (4)                with pytest.raises(ValueError, match=r"\A" + match + r"\Z"):
6373: (8)                    ax.plot("string", fmt, data=data)
6374: (0)            def test_plot_format():
6375: (4)                fig, ax = plt.subplots()
6376: (4)                line = ax.plot([1, 2, 3], '1.0')
6377: (4)                assert line[0].get_color() == (1.0, 1.0, 1.0, 1.0)
6378: (4)                assert line[0].get_marker() == 'None'
6379: (4)                fig, ax = plt.subplots()
6380: (4)                line = ax.plot([1, 2, 3], '1')
6381: (4)                assert line[0].get_marker() == '1'
6382: (4)                fig, ax = plt.subplots()
6383: (4)                line = ax.plot([1, 2], [1, 2], '1.0', "1")
6384: (4)                fig.canvas.draw()
6385: (4)                assert line[0].get_color() == (1.0, 1.0, 1.0, 1.0)
6386: (4)                assert ax.get_yticklabels()[0].get_text() == '1'
6387: (4)                fig, ax = plt.subplots()
6388: (4)                line = ax.plot([1, 2], [1, 2], '1', "1.0")
6389: (4)                fig.canvas.draw()
6390: (4)                assert line[0].get_marker() == '1'
6391: (4)                assert ax.get_yticklabels()[0].get_text() == '1.0'
6392: (4)                fig, ax = plt.subplots()
6393: (4)                line = ax.plot([1, 2, 3], 'k3')
6394: (4)                assert line[0].get_marker() == '3'
6395: (4)                assert line[0].get_color() == 'k'
6396: (4)                fig, ax = plt.subplots()
6397: (4)                line = ax.plot([1, 2, 3], '.C12:')
6398: (4)                assert line[0].get_marker() == '.'
6399: (4)                assert line[0].get_color() == mcolors.to_rgba('C12')
6400: (4)                assert line[0].get_linestyle() == ':'
6401: (0)            def test_automatic_legend():
6402: (4)                fig, ax = plt.subplots()
6403: (4)                ax.plot("a", "b", data={"d": 2})
6404: (4)                leg = ax.legend()
6405: (4)                fig.canvas.draw()
6406: (4)                assert leg.get_texts()[0].get_text() == 'a'
```

```
6407: (4)               assert ax.get_yticklabels()[0].get_text() == 'a'
6408: (4)               fig, ax = plt.subplots()
6409: (4)               ax.plot("a", "b", "c", data={"d": 2})
6410: (4)               leg = ax.legend()
6411: (4)               fig.canvas.draw()
6412: (4)               assert leg.get_texts()[0].get_text() == 'b'
6413: (4)               assert ax.get_xticklabels()[0].get_text() == 'a'
6414: (4)               assert ax.get_yticklabels()[0].get_text() == 'b'
6415: (0)           def test_plot_errors():
6416: (4)               with pytest.raises(TypeError, match=r"plot\(\) got an unexpected
keyword"):
6417: (8)                   plt.plot([1, 2, 3], x=1)
6418: (4)               with pytest.raises(ValueError, match=r"plot\(\) with multiple groups"):
6419: (8)                   plt.plot([1, 2, 3], [1, 2, 3], [2, 3, 4], [2, 3, 4], label=['1', '2'])
6420: (4)               with pytest.raises(ValueError, match="x and y must have same first"):
6421: (8)                   plt.plot([1, 2, 3], [1])
6422: (4)               with pytest.raises(ValueError, match="x and y can be no greater than"):
6423: (8)                   plt.plot(np.ones((2, 2, 2)))
6424: (4)               with pytest.raises(ValueError, match="Using arbitrary long args with"):
6425: (8)                   plt.plot("a", "b", "c", "d", data={"a": 2})
6426: (0)           def test_clim():
6427: (4)               ax = plt.figure().add_subplot()
6428: (4)               for plot_method in [
6429: (12)                      partial(ax.scatter, range(3), range(3), c=range(3)),
6430: (12)                      partial(ax.imshow, [[0, 1], [2, 3]]),
6431: (12)                      partial(ax.pcolor,  [[0, 1], [2, 3]]),
6432: (12)                      partial(ax.pcolormesh, [[0, 1], [2, 3]]),
6433: (12)                      partial(ax.pcolorfast, [[0, 1], [2, 3]]),
6434: (4)               ]:
6435: (8)                   clim = (7, 8)
6436: (8)                   norm = plot_method(clim=clim).norm
6437: (8)                   assert (norm.vmin, norm.vmax) == clim
6438: (0)           def test_bezier_autoscale():
6439: (4)               verts = [[-1, 0],
6440: (13)                        [0, -1],
6441: (13)                        [1, 0],
6442: (13)                        [1, 0]]
6443: (4)               codes = [mpath.Path.MOVETO,
6444: (13)                       mpath.Path.CURVE3,
6445: (13)                       mpath.Path.CURVE3,
6446: (13)                       mpath.Path.CLOSEPOLY]
6447: (4)               p = mpath.Path(verts, codes)
6448: (4)               fig, ax = plt.subplots()
6449: (4)               ax.add_patch(mpatches.PathPatch(p))
6450: (4)               ax.autoscale()
6451: (4)               assert ax.get_ylim()[0] == -0.5
6452: (0)           def test_small_autoscale():
6453: (4)               verts = np.array([
6454: (8)                   [-5.45, 0.00], [-5.45, 0.00], [-5.29, 0.00], [-5.29, 0.00],
6455: (8)                   [-5.13, 0.00], [-5.13, 0.00], [-4.97, 0.00], [-4.97, 0.00],
6456: (8)                   [-4.81, 0.00], [-4.81, 0.00], [-4.65, 0.00], [-4.65, 0.00],
6457: (8)                   [-4.49, 0.00], [-4.49, 0.00], [-4.33, 0.00], [-4.33, 0.00],
6458: (8)                   [-4.17, 0.00], [-4.17, 0.00], [-4.01, 0.00], [-4.01, 0.00],
6459: (8)                   [-3.85, 0.00], [-3.85, 0.00], [-3.69, 0.00], [-3.69, 0.00],
6460: (8)                   [-3.53, 0.00], [-3.53, 0.00], [-3.37, 0.00], [-3.37, 0.00],
6461: (8)                   [-3.21, 0.00], [-3.21, 0.01], [-3.05, 0.01], [-3.05, 0.01],
6462: (8)                   [-2.89, 0.01], [-2.89, 0.01], [-2.73, 0.01], [-2.73, 0.02],
6463: (8)                   [-2.57, 0.02], [-2.57, 0.04], [-2.41, 0.04], [-2.41, 0.04],
6464: (8)                   [-2.25, 0.04], [-2.25, 0.06], [-2.09, 0.06], [-2.09, 0.08],
6465: (8)                   [-1.93, 0.08], [-1.93, 0.10], [-1.77, 0.10], [-1.77, 0.12],
6466: (8)                   [-1.61, 0.12], [-1.61, 0.14], [-1.45, 0.14], [-1.45, 0.17],
6467: (8)                   [-1.30, 0.17], [-1.30, 0.19], [-1.14, 0.19], [-1.14, 0.22],
6468: (8)                   [-0.98, 0.22], [-0.98, 0.25], [-0.82, 0.25], [-0.82, 0.27],
6469: (8)                   [-0.66, 0.27], [-0.66, 0.29], [-0.50, 0.29], [-0.50, 0.30],
6470: (8)                   [-0.34, 0.30], [-0.34, 0.32], [-0.18, 0.32], [-0.18, 0.33],
6471: (8)                   [-0.02, 0.33], [-0.02, 0.32], [0.13, 0.32], [0.13, 0.33], [0.29,
0.33],
6472: (8)                   [0.29, 0.31], [0.45, 0.31], [0.45, 0.30], [0.61, 0.30], [0.61, 0.28],
6473: (8)                   [0.77, 0.28], [0.77, 0.25], [0.93, 0.25], [0.93, 0.22], [1.09, 0.22],
```

```
6474: (8)                   [1.09, 0.19], [1.25, 0.19], [1.25, 0.17], [1.41, 0.17], [1.41, 0.15],
6475: (8)                   [1.57, 0.15], [1.57, 0.12], [1.73, 0.12], [1.73, 0.10], [1.89, 0.10],
6476: (8)                   [1.89, 0.08], [2.05, 0.08], [2.05, 0.07], [2.21, 0.07], [2.21, 0.05],
6477: (8)                   [2.37, 0.05], [2.37, 0.04], [2.53, 0.04], [2.53, 0.02], [2.69, 0.02],
6478: (8)                   [2.69, 0.02], [2.85, 0.02], [2.85, 0.01], [3.01, 0.01], [3.01, 0.01],
6479: (8)                   [3.17, 0.01], [3.17, 0.00], [3.33, 0.00], [3.33, 0.00], [3.49, 0.00],
6480: (8)                   [3.49, 0.00], [3.65, 0.00], [3.65, 0.00], [3.81, 0.00], [3.81, 0.00],
6481: (8)                   [3.97, 0.00], [3.97, 0.00], [4.13, 0.00], [4.13, 0.00], [4.29, 0.00],
6482: (8)                   [4.29, 0.00], [4.45, 0.00], [4.45, 0.00], [4.61, 0.00], [4.61, 0.00],
6483: (8)                   [4.77, 0.00], [4.77, 0.00], [4.93, 0.00], [4.93, 0.00],
6484: (4)               ])
6485: (4)               minx = np.min(verts[:, 0])
6486: (4)               miny = np.min(verts[:, 1])
6487: (4)               maxx = np.max(verts[:, 0])
6488: (4)               maxy = np.max(verts[:, 1])
6489: (4)               p = mpath.Path(verts)
6490: (4)               fig, ax = plt.subplots()
6491: (4)               ax.add_patch(mpatches.PathPatch(p))
6492: (4)               ax.autoscale()
6493: (4)               assert ax.get_xlim()[0] <= minx
6494: (4)               assert ax.get_xlim()[1] >= maxx
6495: (4)               assert ax.get_ylim()[0] <= miny
6496: (4)               assert ax.get_ylim()[1] >= maxy
6497: (0)           def test_get_xticklabel():
6498: (4)               fig, ax = plt.subplots()
6499: (4)               ax.plot(np.arange(10))
6500: (4)               for ind in range(10):
6501: (8)                   assert ax.get_xticklabels()[ind].get_text() == f'{ind}'
6502: (8)                   assert ax.get_yticklabels()[ind].get_text() == f'{ind}'
6503: (0)           def test_bar_leading_nan():
6504: (4)               barx = np.arange(3, dtype=float)
6505: (4)               barheights = np.array([0.5, 1.5, 2.0])
6506: (4)               barstarts = np.array([0.77]*3)
6507: (4)               barx[0] = np.nan
6508: (4)               fig, ax = plt.subplots()
6509: (4)               bars = ax.bar(barx, barheights, bottom=barstarts)
6510: (4)               hbars = ax.barh(barx, barheights, left=barstarts)
6511: (4)               for bar_set in (bars, hbars):
6512: (8)                   nanful, *rest = bar_set
6513: (8)                   assert (~np.isfinite(nanful.xy)).any()
6514: (8)                   assert np.isfinite(nanful.get_width())
6515: (8)                   for b in rest:
6516: (12)                      assert np.isfinite(b.xy).all()
6517: (12)                      assert np.isfinite(b.get_width())
6518: (0)           @check_figures_equal(extensions=["png"])
6519: (0)           def test_bar_all_nan(fig_test, fig_ref):
6520: (4)               mpl.style.use("mpl20")
6521: (4)               ax_test = fig_test.subplots()
6522: (4)               ax_ref = fig_ref.subplots()
6523: (4)               ax_test.bar([np.nan], [np.nan])
6524: (4)               ax_test.bar([1], [1])
6525: (4)               ax_ref.bar([1], [1]).remove()
6526: (4)               ax_ref.bar([1], [1])
6527: (0)           @image_comparison(["extent_units.png"], style="mpl20")
6528: (0)           def test_extent_units():
6529: (4)               _, axs = plt.subplots(2, 2)
6530: (4)               date_first = np.datetime64('2020-01-01', 'D')
6531: (4)               date_last = np.datetime64('2020-01-11', 'D')
6532: (4)               arr = [[i+j for i in range(10)] for j in range(10)]
6533: (4)               axs[0, 0].set_title('Date extents on y axis')
6534: (4)               im = axs[0, 0].imshow(arr, origin='lower',
6535: (26)                                    extent=[1, 11, date_first, date_last],
6536: (26)                                    cmap=mpl.colormaps["plasma"])
6537: (4)               axs[0, 1].set_title('Date extents on x axis (Day of Jan 2020)')
6538: (4)               im = axs[0, 1].imshow(arr, origin='lower',
6539: (26)                                    extent=[date_first, date_last, 1, 11],
6540: (26)                                    cmap=mpl.colormaps["plasma"])
6541: (4)               axs[0, 1].xaxis.set_major_formatter(mdates.DateFormatter('%d'))
6542: (4)               im = axs[1, 0].imshow(arr, origin='lower',
```

```
(26)                                    extent=[date_first, date_last,
(34)                                            date_first, date_last],
(26)                                    cmap=mpl.colormaps["plasma"])
(4)                 axs[1, 0].xaxis.set_major_formatter(mdates.DateFormatter('%d'))
(4)                 axs[1, 0].set(xlabel='Day of Jan 2020')
(4)                 im = axs[1, 1].imshow(arr, origin='lower',
(26)                                    cmap=mpl.colormaps["plasma"])
(4)                 im.set_extent([date_last, date_first, date_last, date_first])
(4)                 axs[1, 1].xaxis.set_major_formatter(mdates.DateFormatter('%d'))
(4)                 axs[1, 1].set(xlabel='Day of Jan 2020')
(4)                 with pytest.raises(TypeError, match=r"set_extent\(\) got an unexpected"):
(8)                     im.set_extent([2, 12, date_first, date_last], clip=False)
(0)             def test_cla_clears_children_axes_and_fig():
(4)                 fig, ax = plt.subplots()
(4)                 lines = ax.plot([], [], [], [])
(4)                 img = ax.imshow([[1]])
(4)                 for art in lines + [img]:
(8)                     assert art.axes is ax
(8)                     assert art.figure is fig
(4)                 ax.clear()
(4)                 for art in lines + [img]:
(8)                     assert art.axes is None
(8)                     assert art.figure is None
(0)             def test_child_axes_removal():
(4)                 fig, ax = plt.subplots()
(4)                 marginal = ax.inset_axes([1, 0, .1, 1], sharey=ax)
(4)                 marginal_twin = marginal.twinx()
(4)                 marginal.remove()
(4)                 ax.set(xlim=(-1, 1), ylim=(10, 20))
(0)             def test_scatter_color_repr_error():
(4)                 def get_next_color():
(8)                     return 'blue'  # pragma: no cover
(4)                 msg = (
(12)                        r"'c' argument must be a color, a sequence of colors"
(12)                        r", or a sequence of numbers, not 'red\\n'"
(8)                     )
(4)                 with pytest.raises(ValueError, match=msg):
(8)                     c = 'red\n'
(8)                     mpl.axes.Axes._parse_scatter_color_args(
(12)                        c, None, kwargs={}, xsize=2, get_next_color_func=get_next_color)
(0)             def test_zorder_and_explicit_rasterization():
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_rasterization_zorder(5)
(4)                 ln, = ax.plot(range(5), rasterized=True, zorder=1)
(4)                 with io.BytesIO() as b:
(8)                     fig.savefig(b, format='pdf')
(0)             @image_comparison(["preset_clip_paths.png"], remove_text=True, style="mpl20",
(18)                            tol=0.027 if platform.machine() == "arm64" else 0)
(0)             def test_preset_clip_paths():
(4)                 fig, ax = plt.subplots()
(4)                 poly = mpl.patches.Polygon(
(8)                     [[1, 0], [0, 1], [-1, 0], [0, -1]], facecolor="#ddffdd",
(8)                     edgecolor="#00ff00", linewidth=2, alpha=0.5)
(4)                 ax.add_patch(poly)
(4)                 line = mpl.lines.Line2D((-1, 1), (0.5, 0.5), clip_on=True, clip_path=poly)
(4)                 line.set_path_effects([patheffects.withTickedStroke()])
(4)                 ax.add_artist(line)
(4)                 line = mpl.lines.Line2D((-1, 1), (-0.5, -0.5), color='r', clip_on=True,
(28)                                        clip_path=poly)
(4)                 ax.add_artist(line)
(4)                 poly2 = mpl.patches.Polygon(
(8)                     [[-1, 1], [0, 1], [0, -0.25]], facecolor="#beefc0", alpha=0.3,
(8)                     edgecolor="#faded0", linewidth=2, clip_on=True, clip_path=poly)
(4)                 ax.add_artist(poly2)
(4)                 ax.annotate('Annotation', (-0.75, -0.75), xytext=(0.1, 0.75),
(16)                            arrowprops={'color': 'k'}, clip_on=True, clip_path=poly)
(4)                 poly3 = mpl.patches.Polygon(
(8)                     [[0, 0], [0, 0.5], [0.5, 0.5], [0.5, 0]], facecolor="g",
edgecolor="y",
```

```
6611: (8)                   linewidth=2, alpha=0.3, clip_on=True, clip_path=poly)
6612: (4)               fig.add_artist(poly3, clip=True)
6613: (4)               ax.set_xlim(-1, 1)
6614: (4)               ax.set_ylim(-1, 1)
6615: (0)           @mpl.style.context('default')
6616: (0)           def test_rc_axes_label_formatting():
6617: (4)               mpl.rcParams['axes.labelcolor'] = 'red'
6618: (4)               mpl.rcParams['axes.labelsize'] = 20
6619: (4)               mpl.rcParams['axes.labelweight'] = 'bold'
6620: (4)               ax = plt.axes()
6621: (4)               assert ax.xaxis.label.get_color() == 'red'
6622: (4)               assert ax.xaxis.label.get_fontsize() == 20
6623: (4)               assert ax.xaxis.label.get_fontweight() == 'bold'
6624: (0)           @check_figures_equal(extensions=["png"])
6625: (0)           def test_ecdf(fig_test, fig_ref):
6626: (4)               data = np.array([0, -np.inf, -np.inf, np.inf, 1, 1, 2])
6627: (4)               weights = range(len(data))
6628: (4)               axs_test = fig_test.subplots(1, 2)
6629: (4)               for ax, orientation in zip(axs_test, ["vertical", "horizontal"]):
6630: (8)                   l0 = ax.ecdf(data, orientation=orientation)
6631: (8)                   l1 = ax.ecdf("d", "w", data={"d": np.ma.array(data), "w": weights},
6632: (21)                               orientation=orientation,
6633: (21)                               complementary=True, compress=True, ls=":")
6634: (8)                   assert len(l0.get_xdata()) == (~np.isnan(data)).sum() + 1
6635: (8)                   assert len(l1.get_xdata()) == len({*data[~np.isnan(data)]}) + 1
6636: (4)               axs_ref = fig_ref.subplots(1, 2)
6637: (4)               axs_ref[0].plot([-np.inf, -np.inf, -np.inf, 0, 1, 1, 2, np.inf],
6638: (20)                              np.arange(8) / 7, ds="steps-post")
6639: (4)               axs_ref[0].plot([-np.inf, 0, 1, 2, np.inf, np.inf],
6640: (20)                              np.array([21, 20, 18, 14, 3, 0]) / 21,
6641: (20)                              ds="steps-pre", ls=":")
6642: (4)               axs_ref[1].plot(np.arange(8) / 7,
6643: (20)                              [-np.inf, -np.inf, -np.inf, 0, 1, 1, 2, np.inf],
6644: (20)                              ds="steps-pre")
6645: (4)               axs_ref[1].plot(np.array([21, 20, 18, 14, 3, 0]) / 21,
6646: (20)                              [-np.inf, 0, 1, 2, np.inf, np.inf],
6647: (20)                              ds="steps-post", ls=":")
6648: (0)           def test_ecdf_invalid():
6649: (4)               with pytest.raises(ValueError):
6650: (8)                   plt.ecdf([1, np.nan])
6651: (4)               with pytest.raises(ValueError):
6652: (8)                   plt.ecdf(np.ma.array([1, 2], mask=[True, False]))
6653: (0)           def test_fill_between_axes_limits():
6654: (4)               fig, ax = plt.subplots()
6655: (4)               x = np.arange(0, 4 * np.pi, 0.01)
6656: (4)               y = 0.1*np.sin(x)
6657: (4)               threshold = 0.075
6658: (4)               ax.plot(x, y, color='black')
6659: (4)               original_lims = (ax.get_xlim(), ax.get_ylim())
6660: (4)               ax.axhline(threshold, color='green', lw=2, alpha=0.7)
6661: (4)               ax.fill_between(x, 0, 1, where=y > threshold,
6662: (20)                              color='green', alpha=0.5,
transform=ax.get_xaxis_transform())
6663: (4)               assert (ax.get_xlim(), ax.get_ylim()) == original_lims
6664: (0)           def test_tick_param_labelfont():
6665: (4)               fig, ax = plt.subplots()
6666: (4)               ax.plot([1, 2, 3, 4], [1, 2, 3, 4])
6667: (4)               ax.set_xlabel('X label in Impact font', fontname='Impact')
6668: (4)               ax.set_ylabel('Y label in xkcd script', fontname='xkcd script')
6669: (4)               ax.tick_params(color='r', labelfontfamily='monospace')
6670: (4)               plt.title('Title in sans-serif')
6671: (4)               for text in ax.get_xticklabels():
6672: (8)                   assert text.get_fontfamily()[0] == 'monospace'
6673: (0)           def test_set_secondary_axis_color():
6674: (4)               fig, ax = plt.subplots()
6675: (4)               sax = ax.secondary_xaxis("top", color="red")
6676: (4)               assert mcolors.same_color(sax.spines["bottom"].get_edgecolor(), "red")
6677: (4)               assert mcolors.same_color(sax.spines["top"].get_edgecolor(), "red")
6678: (4)               assert mcolors.same_color(sax.xaxis.get_tick_params()["color"], "red")
```

```
(4)             assert mcolors.same_color(sax.xaxis.get_tick_params()["labelcolor"],
"red")
(4)             assert mcolors.same_color(sax.xaxis.label.get_color(), "red")
(0)         def test_xylim_changed_shared():
(4)             fig, axs = plt.subplots(2, sharex=True, sharey=True)
(4)             events = []
(4)             axs[1].callbacks.connect("xlim_changed", events.append)
(4)             axs[1].callbacks.connect("ylim_changed", events.append)
(4)             axs[0].set(xlim=[1, 3], ylim=[2, 4])
(4)             assert events == [axs[1], axs[1]]
(0)         @image_comparison(["axhvlinespan_interpolation.png"], style="default")
(0)         def test_axhvlinespan_interpolation():
(4)             ax = plt.figure().add_subplot(projection="polar")
(4)             ax.set_axis_off()
(4)             ax.axvline(.1, c="C0")
(4)             ax.axvspan(.2, .3, fc="C1")
(4)             ax.axvspan(.4, .5, .1, .2, fc="C2")
(4)             ax.axhline(1, c="C0", alpha=.5)
(4)             ax.axhspan(.8, .9, fc="C1", alpha=.5)
(4)             ax.axhspan(.6, .7, .8, .9, fc="C2", alpha=.5)
(0)         @check_figures_equal(extensions=["png"])
(0)         @pytest.mark.parametrize("which", ("x", "y"))
(0)         def test_axes_clear_behavior(fig_ref, fig_test, which):
(4)             """Test that the given tick params are not reset by ax.clear()."""
(4)             ax_test = fig_test.subplots()
(4)             ax_ref = fig_ref.subplots()
(4)             target = {
(8)                 "direction": "in",
(8)                 "length": 10,
(8)                 "width": 10,
(8)                 "color": "xkcd:wine red",
(8)                 "pad": 0,
(8)                 "labelfontfamily": "serif",
(8)                 "zorder": 7,
(8)                 "labelrotation": 45,
(8)                 "labelcolor": "xkcd:shocking pink",
(8)                 "grid_color": "xkcd:fluorescent green",
(8)                 "grid_alpha": 0.5,
(8)                 "grid_linewidth": 3,
(8)                 "grid_linestyle": ":",
(8)                 "bottom": False,
(8)                 "top": True,
(8)                 "left": False,
(8)                 "right": True,
(8)                 "labelbottom": True,
(8)                 "labeltop": True,
(8)                 "labelleft": True,
(8)                 "labelright": True,
(4)             }
(4)             ax_ref.tick_params(axis=which, **target)
(4)             ax_test.tick_params(axis=which, **target)
(4)             ax_test.clear()
(4)             ax_ref.grid(True)
(4)             ax_test.grid(True)
(0)         def test_boxplot_tick_labels():
(4)             np.random.seed(19680801)
(4)             data = np.random.random((10, 3))
(4)             fig, axs = plt.subplots(nrows=1, ncols=2, sharey=True)
(4)             with pytest.warns(mpl.MatplotlibDeprecationWarning,
(22)                              match='has been renamed \'tick_labels\''):
(8)                 axs[0].boxplot(data, labels=['A', 'B', 'C'])
(4)             assert [l.get_text() for l in axs[0].get_xticklabels()] == ['A', 'B', 'C']
(4)             axs[1].boxplot(data, tick_labels=['A', 'B', 'C'])
(4)             assert [l.get_text() for l in axs[1].get_xticklabels()] == ['A', 'B', 'C']
(0)         @needs_usetex
(0)         @check_figures_equal()
(0)         def test_latex_pie_percent(fig_test, fig_ref):
(4)             data = [20, 10, 70]
(4)             ax = fig_test.subplots()
```

```
6747: (4)                ax.pie(data, autopct="%1.0f%%", textprops={'usetex': True})
6748: (4)                ax1 = fig_ref.subplots()
6749: (4)                ax1.pie(data, autopct=r"%1.0f\%%", textprops={'usetex': True})

----------------------------------------

File 151 - test_axis.py:

1: (0)              import numpy as np
2: (0)              import matplotlib.pyplot as plt
3: (0)              from matplotlib.axis import XTick
4: (0)              def test_tick_labelcolor_array():
5: (4)                  ax = plt.axes()
6: (4)                  XTick(ax, 0, labelcolor=np.array([1, 0, 0, 1]))

----------------------------------------

File 152 - test_backend_bases.py:

1: (0)              import importlib
2: (0)              from matplotlib import path, transforms
3: (0)              from matplotlib.backend_bases import (
4: (4)                  FigureCanvasBase, KeyEvent, LocationEvent, MouseButton, MouseEvent,
5: (4)                  NavigationToolbar2, RendererBase)
6: (0)              from matplotlib.backend_tools import RubberbandBase
7: (0)              from matplotlib.figure import Figure
8: (0)              from matplotlib.testing._markers import needs_pgf_xelatex
9: (0)              import matplotlib.pyplot as plt
10: (0)             import numpy as np
11: (0)             import pytest
12: (0)             _EXPECTED_WARNING_TOOLMANAGER = (
13: (4)                 r"Treat the new Tool classes introduced in "
14: (4)                 r"v[0-9]*.[0-9]* as experimental for now; "
15: (4)                 "the API and rcParam may change in future versions.")
16: (0)             def test_uses_per_path():
17: (4)                 id = transforms.Affine2D()
18: (4)                 paths = [path.Path.unit_regular_polygon(i) for i in range(3, 7)]
19: (4)                 tforms_matrices = [id.rotate(i).get_matrix().copy() for i in range(1, 5)]
20: (4)                 offsets = np.arange(20).reshape((10, 2))
21: (4)                 facecolors = ['red', 'green']
22: (4)                 edgecolors = ['red', 'green']
23: (4)                 def check(master_transform, paths, all_transforms,
24: (14)                          offsets, facecolors, edgecolors):
25: (8)                     rb = RendererBase()
26: (8)                     raw_paths = list(rb._iter_collection_raw_paths(
27: (12)                        master_transform, paths, all_transforms))
28: (8)                     gc = rb.new_gc()
29: (8)                     ids = [path_id for xo, yo, path_id, gc0, rgbFace in
30: (15)                           rb._iter_collection(
31: (19)                               gc, range(len(raw_paths)), offsets,
32: (19)                               transforms.AffineDeltaTransform(master_transform),
33: (19)                               facecolors, edgecolors, [], [], [False],
34: (19)                               [], 'screen')]
35: (8)                     uses = rb._iter_collection_uses_per_path(
36: (12)                        paths, all_transforms, offsets, facecolors, edgecolors)
37: (8)                     if raw_paths:
38: (12)                        seen = np.bincount(ids, minlength=len(raw_paths))
39: (12)                        assert set(seen).issubset([uses - 1, uses])
40: (4)                 check(id, paths, tforms_matrices, offsets, facecolors, edgecolors)
41: (4)                 check(id, paths[0:1], tforms_matrices, offsets, facecolors, edgecolors)
42: (4)                 check(id, [], tforms_matrices, offsets, facecolors, edgecolors)
43: (4)                 check(id, paths, tforms_matrices[0:1], offsets, facecolors, edgecolors)
44: (4)                 check(id, paths, [], offsets, facecolors, edgecolors)
45: (4)                 for n in range(0, offsets.shape[0]):
46: (8)                     check(id, paths, tforms_matrices, offsets[0:n, :],
47: (14)                          facecolors, edgecolors)
48: (4)                 check(id, paths, tforms_matrices, offsets, [], edgecolors)
49: (4)                 check(id, paths, tforms_matrices, offsets, facecolors, [])
50: (4)                 check(id, paths, tforms_matrices, offsets, [], [])
```

```
 51: (4)                 check(id, paths, tforms_matrices, offsets, facecolors[0:1], edgecolors)
 52: (0)             def test_canvas_ctor():
 53: (4)                 assert isinstance(FigureCanvasBase().figure, Figure)
 54: (0)             def test_get_default_filename():
 55: (4)                 assert plt.figure().canvas.get_default_filename() == 'image.png'
 56: (0)             def test_canvas_change():
 57: (4)                 fig = plt.figure()
 58: (4)                 canvas = FigureCanvasBase(fig)
 59: (4)                 plt.close(fig)
 60: (4)                 assert not plt.fignum_exists(fig.number)
 61: (0)             @pytest.mark.backend('pdf')
 62: (0)             def test_non_gui_warning(monkeypatch):
 63: (4)                 plt.subplots()
 64: (4)                 monkeypatch.setenv("DISPLAY", ":999")
 65: (4)                 with pytest.warns(UserWarning) as rec:
 66: (8)                     plt.show()
 67: (8)                     assert len(rec) == 1
 68: (8)                     assert ('FigureCanvasPdf is non-interactive, and thus cannot be shown'
 69: (16)                            in str(rec[0].message))
 70: (4)                 with pytest.warns(UserWarning) as rec:
 71: (8)                     plt.gcf().show()
 72: (8)                     assert len(rec) == 1
 73: (8)                     assert ('FigureCanvasPdf is non-interactive, and thus cannot be shown'
 74: (16)                            in str(rec[0].message))
 75: (0)             def test_grab_clear():
 76: (4)                 fig, ax = plt.subplots()
 77: (4)                 fig.canvas.grab_mouse(ax)
 78: (4)                 assert fig.canvas.mouse_grabber == ax
 79: (4)                 fig.clear()
 80: (4)                 assert fig.canvas.mouse_grabber is None
 81: (0)             @pytest.mark.parametrize(
 82: (4)                 "x, y", [(42, 24), (None, 42), (None, None), (200, 100.01), (205.75,
 2.0)])
 83: (0)             def test_location_event_position(x, y):
 84: (4)                 fig, ax = plt.subplots()
 85: (4)                 canvas = FigureCanvasBase(fig)
 86: (4)                 event = LocationEvent("test_event", canvas, x, y)
 87: (4)                 if x is None:
 88: (8)                     assert event.x is None
 89: (4)                 else:
 90: (8)                     assert event.x == int(x)
 91: (8)                     assert isinstance(event.x, int)
 92: (4)                 if y is None:
 93: (8)                     assert event.y is None
 94: (4)                 else:
 95: (8)                     assert event.y == int(y)
 96: (8)                     assert isinstance(event.y, int)
 97: (4)                 if x is not None and y is not None:
 98: (8)                     assert (ax.format_coord(x, y)
 99: (16)                            == f"(x, y) = ({ax.format_xdata(x)}, {ax.format_ydata(y)})")
 100: (8)                    ax.fmt_xdata = ax.fmt_ydata = lambda x: "foo"
 101: (8)                    assert ax.format_coord(x, y) == "(x, y) = (foo, foo)"
 102: (0)            def test_location_event_position_twin():
 103: (4)                fig, ax = plt.subplots()
 104: (4)                ax.set(xlim=(0, 10), ylim=(0, 20))
 105: (4)                assert ax.format_coord(5., 5.) == "(x, y) = (5.00, 5.00)"
 106: (4)                ax.twinx().set(ylim=(0, 40))
 107: (4)                assert ax.format_coord(5., 5.) == "(x, y) = (5.00, 5.00) | (5.00, 10.0)"
 108: (4)                ax.twiny().set(xlim=(0, 5))
 109: (4)                assert (ax.format_coord(5., 5.)
 110: (12)                       == "(x, y) = (5.00, 5.00) | (5.00, 10.0) | (2.50, 5.00)")
 111: (0)            def test_pick():
 112: (4)                fig = plt.figure()
 113: (4)                fig.text(.5, .5, "hello", ha="center", va="center", picker=True)
 114: (4)                fig.canvas.draw()
 115: (4)                picks = []
 116: (4)                def handle_pick(event):
 117: (8)                    assert event.mouseevent.key == "a"
 118: (8)                    picks.append(event)
```

```
119: (4)                fig.canvas.mpl_connect("pick_event", handle_pick)
120: (4)                KeyEvent("key_press_event", fig.canvas, "a")._process()
121: (4)                MouseEvent("button_press_event", fig.canvas,
122: (15)                           *fig.transFigure.transform((.5, .5)),
123: (15)                           MouseButton.LEFT)._process()
124: (4)                KeyEvent("key_release_event", fig.canvas, "a")._process()
125: (4)                assert len(picks) == 1
126: (0)            def test_interactive_zoom():
127: (4)                fig, ax = plt.subplots()
128: (4)                ax.set(xscale="logit")
129: (4)                assert ax.get_navigate_mode() is None
130: (4)                tb = NavigationToolbar2(fig.canvas)
131: (4)                tb.zoom()
132: (4)                assert ax.get_navigate_mode() == 'ZOOM'
133: (4)                xlim0 = ax.get_xlim()
134: (4)                ylim0 = ax.get_ylim()
135: (4)                d0 = (1e-6, 0.1)
136: (4)                d1 = (1-1e-5, 0.8)
137: (4)                s0 = ax.transData.transform(d0).astype(int)
138: (4)                s1 = ax.transData.transform(d1).astype(int)
139: (4)                start_event = MouseEvent(
140: (8)                    "button_press_event", fig.canvas, *s0, MouseButton.LEFT)
141: (4)                fig.canvas.callbacks.process(start_event.name, start_event)
142: (4)                stop_event = MouseEvent(
143: (8)                    "button_release_event", fig.canvas, *s1, MouseButton.LEFT)
144: (4)                fig.canvas.callbacks.process(stop_event.name, stop_event)
145: (4)                assert ax.get_xlim() == (start_event.xdata, stop_event.xdata)
146: (4)                assert ax.get_ylim() == (start_event.ydata, stop_event.ydata)
147: (4)                start_event = MouseEvent(
148: (8)                    "button_press_event", fig.canvas, *s1, MouseButton.RIGHT)
149: (4)                fig.canvas.callbacks.process(start_event.name, start_event)
150: (4)                stop_event = MouseEvent(
151: (8)                    "button_release_event", fig.canvas, *s0, MouseButton.RIGHT)
152: (4)                fig.canvas.callbacks.process(stop_event.name, stop_event)
153: (4)                assert ax.get_xlim() == pytest.approx(xlim0, rel=0, abs=1e-10)
154: (4)                assert ax.get_ylim() == pytest.approx(ylim0, rel=0, abs=1e-10)
155: (4)                tb.zoom()
156: (4)                assert ax.get_navigate_mode() is None
157: (4)                assert not ax.get_autoscalex_on() and not ax.get_autoscaley_on()
158: (0)            def test_widgetlock_zoompan():
159: (4)                fig, ax = plt.subplots()
160: (4)                ax.plot([0, 1], [0, 1])
161: (4)                fig.canvas.widgetlock(ax)
162: (4)                tb = NavigationToolbar2(fig.canvas)
163: (4)                tb.zoom()
164: (4)                assert ax.get_navigate_mode() is None
165: (4)                tb.pan()
166: (4)                assert ax.get_navigate_mode() is None
167: (0)            @pytest.mark.parametrize("plot_func", ["imshow", "contourf"])
168: (0)            @pytest.mark.parametrize("orientation", ["vertical", "horizontal"])
169: (0)            @pytest.mark.parametrize("tool,button,expected",
170: (25)                                   [("zoom", MouseButton.LEFT, (4, 6)),  # zoom in
171: (26)                                    ("zoom", MouseButton.RIGHT, (-20, 30)),  # zoom out
172: (26)                                    ("pan", MouseButton.LEFT, (-2, 8)),
173: (26)                                    ("pan", MouseButton.RIGHT, (1.47, 7.78))])  # zoom
174: (0)            def test_interactive_colorbar(plot_func, orientation, tool, button, expected):
175: (4)                fig, ax = plt.subplots()
176: (4)                data = np.arange(12).reshape((4, 3))
177: (4)                vmin0, vmax0 = 0, 10
178: (4)                coll = getattr(ax, plot_func)(data, vmin=vmin0, vmax=vmax0)
179: (4)                cb = fig.colorbar(coll, ax=ax, orientation=orientation)
180: (4)                if plot_func == "contourf":
181: (8)                    assert not cb.ax.get_navigate()
182: (8)                    return
183: (4)                assert cb.ax.get_navigate()
184: (4)                vmin, vmax = 4, 6
185: (4)                d0 = (vmin, 0.5)
186: (4)                d1 = (vmax, 0.5)
187: (4)                if orientation == "vertical":
```

```
188: (8)                    d0 = d0[::-1]
189: (8)                    d1 = d1[::-1]
190: (4)                s0 = cb.ax.transData.transform(d0).astype(int)
191: (4)                s1 = cb.ax.transData.transform(d1).astype(int)
192: (4)                start_event = MouseEvent(
193: (8)                    "button_press_event", fig.canvas, *s0, button)
194: (4)                stop_event = MouseEvent(
195: (8)                    "button_release_event", fig.canvas, *s1, button)
196: (4)                tb = NavigationToolbar2(fig.canvas)
197: (4)                if tool == "zoom":
198: (8)                    tb.zoom()
199: (8)                    tb.press_zoom(start_event)
200: (8)                    tb.drag_zoom(stop_event)
201: (8)                    tb.release_zoom(stop_event)
202: (4)                else:
203: (8)                    tb.pan()
204: (8)                    tb.press_pan(start_event)
205: (8)                    tb.drag_pan(stop_event)
206: (8)                    tb.release_pan(stop_event)
207: (4)                assert (cb.vmin, cb.vmax) == pytest.approx(expected, abs=0.15)
208: (0)            def test_toolbar_zoompan():
209: (4)                with pytest.warns(UserWarning, match=_EXPECTED_WARNING_TOOLMANAGER):
210: (8)                    plt.rcParams['toolbar'] = 'toolmanager'
211: (4)                ax = plt.gca()
212: (4)                assert ax.get_navigate_mode() is None
213: (4)                ax.figure.canvas.manager.toolmanager.trigger_tool('zoom')
214: (4)                assert ax.get_navigate_mode() == "ZOOM"
215: (4)                ax.figure.canvas.manager.toolmanager.trigger_tool('pan')
216: (4)                assert ax.get_navigate_mode() == "PAN"
217: (0)            def test_toolbar_home_restores_autoscale():
218: (4)                fig, ax = plt.subplots()
219: (4)                ax.plot(range(11), range(11))
220: (4)                tb = NavigationToolbar2(fig.canvas)
221: (4)                tb.zoom()
222: (4)                KeyEvent("key_press_event", fig.canvas, "k", 100, 100)._process()
223: (4)                KeyEvent("key_press_event", fig.canvas, "l", 100, 100)._process()
224: (4)                assert ax.get_xlim() == ax.get_ylim() == (1, 10)  # Autolimits excluding
0.
225: (4)                KeyEvent("key_press_event", fig.canvas, "k", 100, 100)._process()
226: (4)                KeyEvent("key_press_event", fig.canvas, "l", 100, 100)._process()
227: (4)                assert ax.get_xlim() == ax.get_ylim() == (0, 10)  # Autolimits.
228: (4)                start, stop = ax.transData.transform([(2, 2), (5, 5)])
229: (4)                MouseEvent("button_press_event", fig.canvas, *start,
MouseButton.LEFT)._process()
230: (4)                MouseEvent("button_release_event", fig.canvas, *stop,
MouseButton.LEFT)._process()
231: (4)                KeyEvent("key_press_event", fig.canvas, "h")._process()
232: (4)                assert ax.get_xlim() == ax.get_ylim() == (0, 10)
233: (4)                KeyEvent("key_press_event", fig.canvas, "k", 100, 100)._process()
234: (4)                KeyEvent("key_press_event", fig.canvas, "l", 100, 100)._process()
235: (4)                assert ax.get_xlim() == ax.get_ylim() == (1, 10)  # Autolimits excluding
0.
236: (0)            @pytest.mark.parametrize(
237: (4)                "backend", ['svg', 'ps', 'pdf',
238: (16)                            pytest.param('pgf', marks=needs_pgf_xelatex)]
239: (0)            )
240: (0)            def test_draw(backend):
241: (4)                from matplotlib.figure import Figure
242: (4)                from matplotlib.backends.backend_agg import FigureCanvas
243: (4)                test_backend =
importlib.import_module(f'matplotlib.backends.backend_{backend}')
244: (4)                TestCanvas = test_backend.FigureCanvas
245: (4)                fig_test = Figure(constrained_layout=True)
246: (4)                TestCanvas(fig_test)
247: (4)                axes_test = fig_test.subplots(2, 2)
248: (4)                fig_agg = Figure(constrained_layout=True)
249: (4)                FigureCanvas(fig_agg)
250: (4)                axes_agg = fig_agg.subplots(2, 2)
251: (4)                init_pos = [ax.get_position() for ax in axes_test.ravel()]
```

```
252: (4)                fig_test.canvas.draw()
253: (4)                fig_agg.canvas.draw()
254: (4)                layed_out_pos_test = [ax.get_position() for ax in axes_test.ravel()]
255: (4)                layed_out_pos_agg = [ax.get_position() for ax in axes_agg.ravel()]
256: (4)                for init, placed in zip(init_pos, layed_out_pos_test):
257: (8)                    assert not np.allclose(init, placed, atol=0.005)
258: (4)                for ref, test in zip(layed_out_pos_agg, layed_out_pos_test):
259: (8)                    np.testing.assert_allclose(ref, test, atol=0.005)
260: (0)            @pytest.mark.parametrize(
261: (4)                "key,mouseend,expectedxlim,expectedylim",
262: (4)                [(None, (0.2, 0.2), (3.49, 12.49), (2.7, 11.7)),
263: (5)                 (None, (0.2, 0.5), (3.49, 12.49), (0, 9)),
264: (5)                 (None, (0.5, 0.2), (0, 9), (2.7, 11.7)),
265: (5)                 (None, (0.5, 0.5), (0, 9), (0, 9)),  # No move
266: (5)                 (None, (0.8, 0.25), (-3.47, 5.53), (2.25, 11.25)),
267: (5)                 (None, (0.2, 0.25), (3.49, 12.49), (2.25, 11.25)),
268: (5)                 (None, (0.8, 0.85), (-3.47, 5.53), (-3.14, 5.86)),
269: (5)                 (None, (0.2, 0.85), (3.49, 12.49), (-3.14, 5.86)),
270: (5)                 ("shift", (0.2, 0.4), (3.49, 12.49), (0, 9)),  # snap to x
271: (5)                 ("shift", (0.4, 0.2), (0, 9), (2.7, 11.7)),  # snap to y
272: (5)                 ("shift", (0.2, 0.25), (3.49, 12.49), (3.49, 12.49)),  # snap to diagonal
273: (5)                 ("shift", (0.8, 0.25), (-3.47, 5.53), (3.47, 12.47)),  # snap to diagonal
274: (5)                 ("shift", (0.8, 0.9), (-3.58, 5.41), (-3.58, 5.41)),  # snap to diagonal
275: (5)                 ("shift", (0.2, 0.85), (3.49, 12.49), (-3.49, 5.51)),  # snap to diagonal
276: (5)                 ("x", (0.2, 0.1), (3.49, 12.49), (0, 9)),  # only x
277: (5)                 ("y", (0.1, 0.2), (0, 9), (2.7, 11.7)),  # only y
278: (5)                 ("control", (0.2, 0.2), (3.49, 12.49), (3.49, 12.49)),  # diagonal
279: (5)                 ("control", (0.4, 0.2), (2.72, 11.72), (2.72, 11.72)),  # diagonal
280: (5)                 ])
281: (0)            def test_interactive_pan(key, mouseend, expectedxlim, expectedylim):
282: (4)                fig, ax = plt.subplots()
283: (4)                ax.plot(np.arange(10))
284: (4)                assert ax.get_navigate()
285: (4)                ax.set_aspect('equal')
286: (4)                mousestart = (0.5, 0.5)
287: (4)                sstart = ax.transData.transform(mousestart).astype(int)
288: (4)                send = ax.transData.transform(mouseend).astype(int)
289: (4)                start_event = MouseEvent(
290: (8)                    "button_press_event", fig.canvas, *sstart, button=MouseButton.LEFT,
291: (8)                    key=key)
292: (4)                stop_event = MouseEvent(
293: (8)                    "button_release_event", fig.canvas, *send, button=MouseButton.LEFT,
294: (8)                    key=key)
295: (4)                tb = NavigationToolbar2(fig.canvas)
296: (4)                tb.pan()
297: (4)                tb.press_pan(start_event)
298: (4)                tb.drag_pan(stop_event)
299: (4)                tb.release_pan(stop_event)
300: (4)                assert tuple(ax.get_xlim()) == pytest.approx(expectedxlim, abs=0.02)
301: (4)                assert tuple(ax.get_ylim()) == pytest.approx(expectedylim, abs=0.02)
302: (0)            def test_toolmanager_remove():
303: (4)                with pytest.warns(UserWarning, match=_EXPECTED_WARNING_TOOLMANAGER):
304: (8)                    plt.rcParams['toolbar'] = 'toolmanager'
305: (4)                fig = plt.gcf()
306: (4)                initial_len = len(fig.canvas.manager.toolmanager.tools)
307: (4)                assert 'forward' in fig.canvas.manager.toolmanager.tools
308: (4)                fig.canvas.manager.toolmanager.remove_tool('forward')
309: (4)                assert len(fig.canvas.manager.toolmanager.tools) == initial_len - 1
310: (4)                assert 'forward' not in fig.canvas.manager.toolmanager.tools
311: (0)            def test_toolmanager_get_tool():
312: (4)                with pytest.warns(UserWarning, match=_EXPECTED_WARNING_TOOLMANAGER):
313: (8)                    plt.rcParams['toolbar'] = 'toolmanager'
314: (4)                fig = plt.gcf()
315: (4)                rubberband = fig.canvas.manager.toolmanager.get_tool('rubberband')
316: (4)                assert isinstance(rubberband, RubberbandBase)
317: (4)                assert fig.canvas.manager.toolmanager.get_tool(rubberband) is rubberband
318: (4)                with pytest.warns(UserWarning,
319: (22)                                  match="ToolManager does not control tool 'foo'"):
320: (8)                    assert fig.canvas.manager.toolmanager.get_tool('foo') is None
```

```
321: (4)                assert fig.canvas.manager.toolmanager.get_tool('foo', warn=False) is None
322: (4)                with pytest.warns(UserWarning,
323: (22)                                  match="ToolManager does not control tool 'foo'"):
324: (8)                    assert fig.canvas.manager.toolmanager.trigger_tool('foo') is None
325: (0)            def test_toolmanager_update_keymap():
326: (4)                with pytest.warns(UserWarning, match=_EXPECTED_WARNING_TOOLMANAGER):
327: (8)                    plt.rcParams['toolbar'] = 'toolmanager'
328: (4)                fig = plt.gcf()
329: (4)                assert 'v' in fig.canvas.manager.toolmanager.get_tool_keymap('forward')
330: (4)                with pytest.warns(UserWarning,
331: (22)                                  match="Key c changed from back to forward"):
332: (8)                    fig.canvas.manager.toolmanager.update_keymap('forward', 'c')
333: (4)                assert fig.canvas.manager.toolmanager.get_tool_keymap('forward') == ['c']
334: (4)                with pytest.raises(KeyError, match="'foo' not in Tools"):
335: (8)                    fig.canvas.manager.toolmanager.update_keymap('foo', 'c')
336: (0)            @pytest.mark.parametrize("tool", ["zoom", "pan"])
337: (0)            @pytest.mark.parametrize("button", [MouseButton.LEFT, MouseButton.RIGHT])
338: (0)            @pytest.mark.parametrize("patch_vis", [True, False])
339: (0)            @pytest.mark.parametrize("forward_nav", [True, False, "auto"])
340: (0)            @pytest.mark.parametrize("t_s", ["twin", "share"])
341: (0)            def test_interactive_pan_zoom_events(tool, button, patch_vis, forward_nav,
t_s):
342: (4)                fig, ax_b = plt.subplots()
343: (4)                ax_t = fig.add_subplot(221, zorder=99)
344: (4)                ax_t.set_forward_navigation_events(forward_nav)
345: (4)                ax_t.patch.set_visible(patch_vis)
346: (4)                if t_s == "share":
347: (8)                    ax_t_twin = fig.add_subplot(222)
348: (8)                    ax_t_twin.sharex(ax_t)
349: (8)                    ax_t_twin.sharey(ax_t)
350: (8)                    ax_b_twin = fig.add_subplot(223)
351: (8)                    ax_b_twin.sharex(ax_b)
352: (8)                    ax_b_twin.sharey(ax_b)
353: (4)                elif t_s == "twin":
354: (8)                    ax_t_twin = ax_t.twinx()
355: (8)                    ax_b_twin = ax_b.twinx()
356: (4)                ax_t.set_label("ax_t")
357: (4)                ax_t.patch.set_facecolor((1, 0, 0, 0.5))
358: (4)                ax_t_twin.set_label("ax_t_twin")
359: (4)                ax_t_twin.patch.set_facecolor("r")
360: (4)                ax_b.set_label("ax_b")
361: (4)                ax_b.patch.set_facecolor((0, 0, 1, 0.5))
362: (4)                ax_b_twin.set_label("ax_b_twin")
363: (4)                ax_b_twin.patch.set_facecolor("b")
364: (4)                init_xlim, init_ylim = (0, 10), (0, 10)
365: (4)                for ax in [ax_t, ax_b]:
366: (8)                    ax.set_xlim(*init_xlim)
367: (8)                    ax.set_ylim(*init_ylim)
368: (4)                xstart_t, xstop_t, ystart_t, ystop_t = 1, 2, 1, 2
369: (4)                s0 = ax_t.transData.transform((xstart_t, ystart_t)).astype(int)
370: (4)                s1 = ax_t.transData.transform((xstop_t, ystop_t)).astype(int)
371: (4)                xstart_b, ystart_b = ax_b.transData.inverted().transform(s0)
372: (4)                xstop_b, ystop_b = ax_b.transData.inverted().transform(s1)
373: (4)                start_event = MouseEvent("button_press_event", fig.canvas, *s0, button)
374: (4)                stop_event = MouseEvent("button_release_event", fig.canvas, *s1, button)
375: (4)                tb = NavigationToolbar2(fig.canvas)
376: (4)                if tool == "zoom":
377: (8)                    direction = ("in" if button == 1 else "out")
378: (8)                    xlim_t, ylim_t = ax_t._prepare_view_from_bbox([*s0, *s1], direction)
379: (8)                    if ax_t.get_forward_navigation_events() is True:
380: (12)                       xlim_b, ylim_b = ax_b._prepare_view_from_bbox([*s0, *s1],
direction)
381: (8)                    elif ax_t.get_forward_navigation_events() is False:
382: (12)                       xlim_b = init_xlim
383: (12)                       ylim_b = init_ylim
384: (8)                    else:
385: (12)                       if not ax_t.patch.get_visible():
386: (16)                           xlim_b, ylim_b = ax_b._prepare_view_from_bbox([*s0, *s1],
direction)
```

```
387: (12)                        else:
388: (16)                            xlim_b = init_xlim
389: (16)                            ylim_b = init_ylim
390: (8)                    tb.zoom()
391: (8)                    tb.press_zoom(start_event)
392: (8)                    tb.drag_zoom(stop_event)
393: (8)                    tb.release_zoom(stop_event)
394: (8)                    assert ax_t.get_xlim() == pytest.approx(xlim_t, abs=0.15)
395: (8)                    assert ax_t.get_ylim() == pytest.approx(ylim_t, abs=0.15)
396: (8)                    assert ax_b.get_xlim() == pytest.approx(xlim_b, abs=0.15)
397: (8)                    assert ax_b.get_ylim() == pytest.approx(ylim_b, abs=0.15)
398: (8)                    assert ax_t.get_xlim() == pytest.approx(ax_t_twin.get_xlim(),
abs=0.15)
399: (8)                    assert ax_b.get_xlim() == pytest.approx(ax_b_twin.get_xlim(),
abs=0.15)
400: (4)                else:
401: (8)                    ax_t.start_pan(*s0, button)
402: (8)                    xlim_t, ylim_t = ax_t._get_pan_points(button, None,
*s1).T.astype(float)
403: (8)                    ax_t.end_pan()
404: (8)                    if ax_t.get_forward_navigation_events() is True:
405: (12)                       ax_b.start_pan(*s0, button)
406: (12)                       xlim_b, ylim_b = ax_b._get_pan_points(button, None,
*s1).T.astype(float)
407: (12)                       ax_b.end_pan()
408: (8)                    elif ax_t.get_forward_navigation_events() is False:
409: (12)                       xlim_b = init_xlim
410: (12)                       ylim_b = init_ylim
411: (8)                    else:
412: (12)                       if not ax_t.patch.get_visible():
413: (16)                           ax_b.start_pan(*s0, button)
414: (16)                           xlim_b, ylim_b = ax_b._get_pan_points(button, None,
*s1).T.astype(float)
415: (16)                           ax_b.end_pan()
416: (12)                       else:
417: (16)                           xlim_b = init_xlim
418: (16)                           ylim_b = init_ylim
419: (8)                    tb.pan()
420: (8)                    tb.press_pan(start_event)
421: (8)                    tb.drag_pan(stop_event)
422: (8)                    tb.release_pan(stop_event)
423: (8)                    assert ax_t.get_xlim() == pytest.approx(xlim_t, abs=0.15)
424: (8)                    assert ax_t.get_ylim() == pytest.approx(ylim_t, abs=0.15)
425: (8)                    assert ax_b.get_xlim() == pytest.approx(xlim_b, abs=0.15)
426: (8)                    assert ax_b.get_ylim() == pytest.approx(ylim_b, abs=0.15)
427: (8)                    assert ax_t.get_xlim() == pytest.approx(ax_t_twin.get_xlim(),
abs=0.15)
428: (8)                    assert ax_b.get_xlim() == pytest.approx(ax_b_twin.get_xlim(),
abs=0.15)

----------------------------------------

File 153 - test_backend_cairo.py:

1: (0)              import numpy as np
2: (0)              import pytest
3: (0)              from matplotlib.testing.decorators import check_figures_equal
4: (0)              from matplotlib import (
5: (4)                  collections as mcollections, patches as mpatches, path as mpath)
6: (0)              @pytest.mark.backend('cairo')
7: (0)              @check_figures_equal(extensions=["png"])
8: (0)              def test_patch_alpha_coloring(fig_test, fig_ref):
9: (4)                  """
10: (4)                 Test checks that the patch and collection are rendered with the specified
11: (4)                 alpha values in their facecolor and edgecolor.
12: (4)                 """
13: (4)                 star = mpath.Path.unit_regular_star(6)
14: (4)                 circle = mpath.Path.unit_circle()
15: (4)                 verts = np.concatenate([circle.vertices, star.vertices[::-1]])
```

```
16: (4)                 codes = np.concatenate([circle.codes, star.codes])
17: (4)                 cut_star1 = mpath.Path(verts, codes)
18: (4)                 cut_star2 = mpath.Path(verts + 1, codes)
19: (4)                 ax = fig_ref.subplots()
20: (4)                 ax.set_xlim([-1, 2])
21: (4)                 ax.set_ylim([-1, 2])
22: (4)                 patch = mpatches.PathPatch(cut_star1,
23: (31)                                           linewidth=5, linestyle='dashdot',
24: (31)                                           facecolor=(1, 0, 0, 0.5),
25: (31)                                           edgecolor=(0, 0, 1, 0.75))
26: (4)                 ax.add_patch(patch)
27: (4)                 patch = mpatches.PathPatch(cut_star2,
28: (31)                                           linewidth=5, linestyle='dashdot',
29: (31)                                           facecolor=(1, 0, 0, 0.5),
30: (31)                                           edgecolor=(0, 0, 1, 0.75))
31: (4)                 ax.add_patch(patch)
32: (4)                 ax = fig_test.subplots()
33: (4)                 ax.set_xlim([-1, 2])
34: (4)                 ax.set_ylim([-1, 2])
35: (4)                 col = mcollections.PathCollection([cut_star1, cut_star2],
36: (38)                                                  linewidth=5, linestyles='dashdot',
37: (38)                                                  facecolor=(1, 0, 0, 0.5),
38: (38)                                                  edgecolor=(0, 0, 1, 0.75))
39: (4)                 ax.add_collection(col)

----------------------------------------

File 154 - test_backend_gtk3.py:

1: (0)              from matplotlib import pyplot as plt
2: (0)              import pytest
3: (0)              @pytest.mark.backend("gtk3agg", skip_on_importerror=True)
4: (0)              def test_correct_key():
5: (4)                  pytest.xfail("test_widget_send_event is not triggering key_press_event")
6: (4)                  from gi.repository import Gdk, Gtk  # type: ignore
7: (4)                  fig = plt.figure()
8: (4)                  buf = []
9: (4)                  def send(event):
10: (8)                     for key, mod in [
11: (16)                            (Gdk.KEY_a, Gdk.ModifierType.SHIFT_MASK),
12: (16)                            (Gdk.KEY_a, 0),
13: (16)                            (Gdk.KEY_a, Gdk.ModifierType.CONTROL_MASK),
14: (16)                            (Gdk.KEY_agrave, 0),
15: (16)                            (Gdk.KEY_Control_L, Gdk.ModifierType.MOD1_MASK),
16: (16)                            (Gdk.KEY_Alt_L, Gdk.ModifierType.CONTROL_MASK),
17: (16)                            (Gdk.KEY_agrave,
18: (17)                             Gdk.ModifierType.CONTROL_MASK
19: (17)                             | Gdk.ModifierType.MOD1_MASK
20: (17)                             | Gdk.ModifierType.MOD4_MASK),
21: (16)                            (0xfd16, 0),   # KEY_3270_Play.
22: (16)                            (Gdk.KEY_BackSpace, 0),
23: (16)                            (Gdk.KEY_BackSpace, Gdk.ModifierType.CONTROL_MASK),
24: (8)                     ]:
25: (12)                        Gtk.test_widget_send_key(fig.canvas, key, mod)
26: (4)                 def receive(event):
27: (8)                     buf.append(event.key)
28: (8)                     if buf == [
29: (16)                            "A", "a", "ctrl+a",
30: (16)                            "\N{LATIN SMALL LETTER A WITH GRAVE}",
31: (16)                            "alt+control", "ctrl+alt",
32: (16)                            "ctrl+alt+super+\N{LATIN SMALL LETTER A WITH GRAVE}",
33: (16)                            "backspace", "ctrl+backspace",
34: (8)                     ]:
35: (12)                        plt.close(fig)
36: (4)                 fig.canvas.mpl_connect("draw_event", send)
37: (4)                 fig.canvas.mpl_connect("key_press_event", receive)
38: (4)                 plt.show()

----------------------------------------
```

File 155 - test_backend_macosx.py:

```
1: (0)              import os
2: (0)              import pytest
3: (0)              import matplotlib as mpl
4: (0)              import matplotlib.pyplot as plt
5: (0)              try:
6: (4)                  from matplotlib.backends import _macosx
7: (0)              except ImportError:
8: (4)                  pytest.skip("These are mac only tests", allow_module_level=True)
9: (0)              @pytest.mark.backend('macosx')
10: (0)             def test_cached_renderer():
11: (4)                 fig = plt.figure(1)
12: (4)                 fig.canvas.draw()
13: (4)                 assert fig.canvas.get_renderer()._renderer is not None
14: (4)                 fig = plt.figure(2)
15: (4)                 fig.draw_without_rendering()
16: (4)                 assert fig.canvas.get_renderer()._renderer is not None
17: (0)             @pytest.mark.backend('macosx')
18: (0)             def test_savefig_rcparam(monkeypatch, tmp_path):
19: (4)                 def new_choose_save_file(title, directory, filename):
20: (8)                     assert directory == str(tmp_path)
21: (8)                     os.makedirs(f"{directory}/test")
22: (8)                     return f"{directory}/test/{filename}"
23: (4)                 monkeypatch.setattr(_macosx, "choose_save_file", new_choose_save_file)
24: (4)                 fig = plt.figure()
25: (4)                 with mpl.rc_context({"savefig.directory": tmp_path}):
26: (8)                     fig.canvas.toolbar.save_figure()
27: (8)                     save_file = f"{tmp_path}/test/{fig.canvas.get_default_filename()}"
28: (8)                     assert os.path.exists(save_file)
29: (8)                     assert mpl.rcParams["savefig.directory"] == f"{tmp_path}/test"
30: (0)             @pytest.mark.backend('macosx')
31: (0)             def test_ipython():
32: (4)                 from matplotlib.testing import ipython_in_subprocess
33: (4)                 ipython_in_subprocess("osx", {(8, 24): "macosx", (7, 0): "MacOSX"})
```

File 156 - test_backend_nbagg.py:

```
1: (0)              import os
2: (0)              from pathlib import Path
3: (0)              from tempfile import TemporaryDirectory
4: (0)              import pytest
5: (0)              from matplotlib.testing import subprocess_run_for_testing
6: (0)              nbformat = pytest.importorskip('nbformat')
7: (0)              pytest.importorskip('nbconvert')
8: (0)              pytest.importorskip('ipykernel')
9: (0)              def test_ipynb():
10: (4)                 nb_path = Path(__file__).parent / 'test_nbagg_01.ipynb'
11: (4)                 with TemporaryDirectory() as tmpdir:
12: (8)                     out_path = Path(tmpdir, "out.ipynb")
13: (8)                     subprocess_run_for_testing(
14: (12)                        ["jupyter", "nbconvert", "--to", "notebook",
15: (13)                         "--execute", "--ExecutePreprocessor.timeout=500",
16: (13)                         "--output", str(out_path), str(nb_path)],
17: (12)                        env={**os.environ, "IPYTHONDIR": tmpdir},
18: (12)                        check=True)
19: (8)                     with out_path.open() as out:
20: (12)                        nb = nbformat.read(out, nbformat.current_nbformat)
21: (4)                 errors = [output for cell in nb.cells for output in cell.get("outputs",
[])
22: (14)                          if output.output_type == "error"]
23: (4)                 assert not errors
24: (4)                 import IPython
25: (4)                 if IPython.version_info[:2] >= (8, 24):
26: (8)                     expected_backend = "notebook"
27: (4)                 else:
```

```
28: (8)                     expected_backend = "nbAgg"
29: (4)                 backend_outputs = nb.cells[2]["outputs"]
30: (4)                 assert backend_outputs[0]["data"]["text/plain"] == f"'{expected_backend}'"
```

File 157 - test_backend_pdf.py:

```
1: (0)              import datetime
2: (0)              import decimal
3: (0)              import io
4: (0)              import os
5: (0)              from pathlib import Path
6: (0)              import numpy as np
7: (0)              import pytest
8: (0)              import matplotlib as mpl
9: (0)              from matplotlib import (
10: (4)                 pyplot as plt, rcParams, font_manager as fm
11: (0)             )
12: (0)             from matplotlib.cbook import _get_data_path
13: (0)             from matplotlib.ft2font import FT2Font
14: (0)             from matplotlib.font_manager import findfont, FontProperties
15: (0)             from matplotlib.backends._backend_pdf_ps import get_glyphs_subset
16: (0)             from matplotlib.backends.backend_pdf import PdfPages
17: (0)             from matplotlib.patches import Rectangle
18: (0)             from matplotlib.testing.decorators import check_figures_equal,
image_comparison
19: (0)             from matplotlib.testing._markers import needs_usetex
20: (0)             @image_comparison(['pdf_use14corefonts.pdf'])
21: (0)             def test_use14corefonts():
22: (4)                 rcParams['pdf.use14corefonts'] = True
23: (4)                 rcParams['font.family'] = 'sans-serif'
24: (4)                 rcParams['font.size'] = 8
25: (4)                 rcParams['font.sans-serif'] = ['Helvetica']
26: (4)                 rcParams['pdf.compression'] = 0
27: (4)                 text = '''A three-line text positioned just above a blue line
28: (0)             and containing some French characters and the euro symbol:
29: (0)             "Merci pépé pour les 10 €"'''
30: (4)                 fig, ax = plt.subplots()
31: (4)                 ax.set_title('Test PDF backend with option use14corefonts=True')
32: (4)                 ax.text(0.5, 0.5, text, horizontalalignment='center',
33: (12)                        verticalalignment='bottom',
34: (12)                        fontsize=14)
35: (4)                 ax.axhline(0.5, linewidth=0.5)
36: (0)             @pytest.mark.parametrize('fontname, fontfile', [
37: (4)                 ('DejaVu Sans', 'DejaVuSans.ttf'),
38: (4)                 ('WenQuanYi Zen Hei', 'wqy-zenhei.ttc'),
39: (0)             ])
40: (0)             @pytest.mark.parametrize('fonttype', [3, 42])
41: (0)             def test_embed_fonts(fontname, fontfile, fonttype):
42: (4)                 if Path(findfont(FontProperties(family=[fontname]))).name != fontfile:
43: (8)                     pytest.skip(f'Font {fontname!r} may be missing')
44: (4)                 rcParams['pdf.fonttype'] = fonttype
45: (4)                 fig, ax = plt.subplots()
46: (4)                 ax.plot([1, 2, 3])
47: (4)                 ax.set_title('Axes Title', font=fontname)
48: (4)                 fig.savefig(io.BytesIO(), format='pdf')
49: (0)             def test_multipage_pagecount():
50: (4)                 with PdfPages(io.BytesIO()) as pdf:
51: (8)                     assert pdf.get_pagecount() == 0
52: (8)                     fig, ax = plt.subplots()
53: (8)                     ax.plot([1, 2, 3])
54: (8)                     fig.savefig(pdf, format="pdf")
55: (8)                     assert pdf.get_pagecount() == 1
56: (8)                     pdf.savefig()
57: (8)                     assert pdf.get_pagecount() == 2
58: (0)             def test_multipage_properfinalize():
59: (4)                 pdfio = io.BytesIO()
60: (4)                 with PdfPages(pdfio) as pdf:
```

```
 61: (8)                    for i in range(10):
 62: (12)                       fig, ax = plt.subplots()
 63: (12)                       ax.set_title('This is a long title')
 64: (12)                       fig.savefig(pdf, format="pdf")
 65: (4)                s = pdfio.getvalue()
 66: (4)                assert s.count(b'startxref') == 1
 67: (4)                assert len(s) < 40000
 68: (0)            def test_multipage_keep_empty(tmp_path):
 69: (4)                fn = tmp_path / "a.pdf"
 70: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning), PdfPages(fn) as pdf:
 71: (8)                    pass
 72: (4)                assert fn.exists()
 73: (4)                fn = tmp_path / "b.pdf"
 74: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning), \
 75: (12)                       PdfPages(fn, keep_empty=True) as pdf:
 76: (8)                    pass
 77: (4)                assert fn.exists()
 78: (4)                fn = tmp_path / "c.pdf"
 79: (4)                with PdfPages(fn, keep_empty=False) as pdf:
 80: (8)                    pass
 81: (4)                assert not fn.exists()
 82: (4)                fn = tmp_path / "d.pdf"
 83: (4)                with PdfPages(fn) as pdf:
 84: (8)                    pdf.savefig(plt.figure())
 85: (4)                assert fn.exists()
 86: (4)                fn = tmp_path / "e.pdf"
 87: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning), \
 88: (12)                       PdfPages(fn, keep_empty=True) as pdf:
 89: (8)                    pdf.savefig(plt.figure())
 90: (4)                assert fn.exists()
 91: (4)                fn = tmp_path / "f.pdf"
 92: (4)                with PdfPages(fn, keep_empty=False) as pdf:
 93: (8)                    pdf.savefig(plt.figure())
 94: (4)                assert fn.exists()
 95: (0)            def test_composite_image():
 96: (4)                X, Y = np.meshgrid(np.arange(-5, 5, 1), np.arange(-5, 5, 1))
 97: (4)                Z = np.sin(Y ** 2)
 98: (4)                fig, ax = plt.subplots()
 99: (4)                ax.set_xlim(0, 3)
 100: (4)               ax.imshow(Z, extent=[0, 1, 0, 1])
 101: (4)               ax.imshow(Z[::-1], extent=[2, 3, 0, 1])
 102: (4)               plt.rcParams['image.composite_image'] = True
 103: (4)               with PdfPages(io.BytesIO()) as pdf:
 104: (8)                   fig.savefig(pdf, format="pdf")
 105: (8)                   assert len(pdf._file._images) == 1
 106: (4)               plt.rcParams['image.composite_image'] = False
 107: (4)               with PdfPages(io.BytesIO()) as pdf:
 108: (8)                   fig.savefig(pdf, format="pdf")
 109: (8)                   assert len(pdf._file._images) == 2
 110: (0)           def test_indexed_image():
 111: (4)               pikepdf = pytest.importorskip('pikepdf')
 112: (4)               data = np.zeros((256, 1, 3), dtype=np.uint8)
 113: (4)               data[:, 0, 0] = np.arange(256)  # Maximum unique colours for an indexed
image.
 114: (4)               rcParams['pdf.compression'] = True
 115: (4)               fig = plt.figure()
 116: (4)               fig.figimage(data, resize=True)
 117: (4)               buf = io.BytesIO()
 118: (4)               fig.savefig(buf, format='pdf', dpi='figure')
 119: (4)               with pikepdf.Pdf.open(buf) as pdf:
 120: (8)                   page, = pdf.pages
 121: (8)                   image, = page.images.values()
 122: (8)                   pdf_image = pikepdf.PdfImage(image)
 123: (8)                   assert pdf_image.indexed
 124: (8)                   pil_image = pdf_image.as_pil_image()
 125: (8)                   rgb = np.asarray(pil_image.convert('RGB'))
 126: (4)               np.testing.assert_array_equal(data, rgb)
 127: (0)           def test_savefig_metadata(monkeypatch):
 128: (4)               pikepdf = pytest.importorskip('pikepdf')
```

```
129: (4)                monkeypatch.setenv('SOURCE_DATE_EPOCH', '0')
130: (4)                fig, ax = plt.subplots()
131: (4)                ax.plot(range(5))
132: (4)                md = {
133: (8)                    'Author': 'me',
134: (8)                    'Title': 'Multipage PDF',
135: (8)                    'Subject': 'Test page',
136: (8)                    'Keywords': 'test,pdf,multipage',
137: (8)                    'ModDate': datetime.datetime(
138: (12)                       1968, 8, 1, tzinfo=datetime.timezone(datetime.timedelta(0))),
139: (8)                    'Trapped': 'True'
140: (4)                }
141: (4)                buf = io.BytesIO()
142: (4)                fig.savefig(buf, metadata=md, format='pdf')
143: (4)                with pikepdf.Pdf.open(buf) as pdf:
144: (8)                    info = {k: str(v) for k, v in pdf.docinfo.items()}
145: (4)                assert info == {
146: (8)                    '/Author': 'me',
147: (8)                    '/CreationDate': 'D:19700101000000Z',
148: (8)                    '/Creator': f'Matplotlib v{mpl.__version__}, https://matplotlib.org',
149: (8)                    '/Keywords': 'test,pdf,multipage',
150: (8)                    '/ModDate': 'D:19680801000000Z',
151: (8)                    '/Producer': f'Matplotlib pdf backend v{mpl.__version__}',
152: (8)                    '/Subject': 'Test page',
153: (8)                    '/Title': 'Multipage PDF',
154: (8)                    '/Trapped': '/True',
155: (4)                }
156: (0)            def test_invalid_metadata():
157: (4)                fig, ax = plt.subplots()
158: (4)                with pytest.warns(UserWarning,
159: (22)                                  match="Unknown infodict keyword: 'foobar'."):
160: (8)                    fig.savefig(io.BytesIO(), format='pdf', metadata={'foobar':
'invalid'})
161: (4)                with pytest.warns(UserWarning,
162: (22)                                  match='not an instance of datetime.datetime.'):
163: (8)                    fig.savefig(io.BytesIO(), format='pdf',
164: (20)                               metadata={'ModDate': '1968-08-01'})
165: (4)                with pytest.warns(UserWarning,
166: (22)                                  match='not one of {"True", "False", "Unknown"}'):
167: (8)                    fig.savefig(io.BytesIO(), format='pdf', metadata={'Trapped': 'foo'})
168: (4)                with pytest.warns(UserWarning, match='not an instance of str.'):
169: (8)                    fig.savefig(io.BytesIO(), format='pdf', metadata={'Title': 1234})
170: (0)            def test_multipage_metadata(monkeypatch):
171: (4)                pikepdf = pytest.importorskip('pikepdf')
172: (4)                monkeypatch.setenv('SOURCE_DATE_EPOCH', '0')
173: (4)                fig, ax = plt.subplots()
174: (4)                ax.plot(range(5))
175: (4)                md = {
176: (8)                    'Author': 'me',
177: (8)                    'Title': 'Multipage PDF',
178: (8)                    'Subject': 'Test page',
179: (8)                    'Keywords': 'test,pdf,multipage',
180: (8)                    'ModDate': datetime.datetime(
181: (12)                       1968, 8, 1, tzinfo=datetime.timezone(datetime.timedelta(0))),
182: (8)                    'Trapped': 'True'
183: (4)                }
184: (4)                buf = io.BytesIO()
185: (4)                with PdfPages(buf, metadata=md) as pdf:
186: (8)                    pdf.savefig(fig)
187: (8)                    pdf.savefig(fig)
188: (4)                with pikepdf.Pdf.open(buf) as pdf:
189: (8)                    info = {k: str(v) for k, v in pdf.docinfo.items()}
190: (4)                assert info == {
191: (8)                    '/Author': 'me',
192: (8)                    '/CreationDate': 'D:19700101000000Z',
193: (8)                    '/Creator': f'Matplotlib v{mpl.__version__}, https://matplotlib.org',
194: (8)                    '/Keywords': 'test,pdf,multipage',
195: (8)                    '/ModDate': 'D:19680801000000Z',
196: (8)                    '/Producer': f'Matplotlib pdf backend v{mpl.__version__}',
```

```
(8)                 '/Subject': 'Test page',
(8)                 '/Title': 'Multipage PDF',
(8)                 '/Trapped': '/True',
(4)             }
(0)         def test_text_urls():
(4)             pikepdf = pytest.importorskip('pikepdf')
(4)             test_url = 'https://test_text_urls.matplotlib.org/'
(4)             fig = plt.figure(figsize=(2, 1))
(4)             fig.text(0.1, 0.1, 'test plain 123', url=f'{test_url}plain')
(4)             fig.text(0.1, 0.4, 'test mathtext $123$', url=f'{test_url}mathtext')
(4)             with io.BytesIO() as fd:
(8)                 fig.savefig(fd, format='pdf')
(8)                 with pikepdf.Pdf.open(fd) as pdf:
(12)                    annots = pdf.pages[0].Annots
(12)                    for y, fragment in [('0.1', 'plain'), ('0.4', 'mathtext')]:
(16)                        annot = next(
(20)                            (a for a in annots if a.A.URI == f'{test_url}{fragment}'),
(20)                            None)
(16)                        assert annot is not None
(16)                        assert getattr(annot, 'QuadPoints', None) is None
(16)                        assert annot.Rect[1] == decimal.Decimal(y) * 72
(0)         def test_text_rotated_urls():
(4)             pikepdf = pytest.importorskip('pikepdf')
(4)             test_url = 'https://test_text_urls.matplotlib.org/'
(4)             fig = plt.figure(figsize=(1, 1))
(4)             fig.text(0.1, 0.1, 'N', rotation=45, url=f'{test_url}')
(4)             with io.BytesIO() as fd:
(8)                 fig.savefig(fd, format='pdf')
(8)                 with pikepdf.Pdf.open(fd) as pdf:
(12)                    annots = pdf.pages[0].Annots
(12)                    annot = next(
(16)                        (a for a in annots if a.A.URI == f'{test_url}'),
(16)                        None)
(12)                    assert annot is not None
(12)                    assert getattr(annot, 'QuadPoints', None) is not None
(12)                    assert annot.Rect[0] == \
(15)                       annot.QuadPoints[6] - decimal.Decimal('0.00001')
(0)         @needs_usetex
(0)         def test_text_urls_tex():
(4)             pikepdf = pytest.importorskip('pikepdf')
(4)             test_url = 'https://test_text_urls.matplotlib.org/'
(4)             fig = plt.figure(figsize=(2, 1))
(4)             fig.text(0.1, 0.7, 'test tex $123$', usetex=True, url=f'{test_url}tex')
(4)             with io.BytesIO() as fd:
(8)                 fig.savefig(fd, format='pdf')
(8)                 with pikepdf.Pdf.open(fd) as pdf:
(12)                    annots = pdf.pages[0].Annots
(12)                    annot = next(
(16)                        (a for a in annots if a.A.URI == f'{test_url}tex'),
(16)                        None)
(12)                    assert annot is not None
(12)                    assert annot.Rect[1] == decimal.Decimal('0.7') * 72
(0)         def test_pdfpages_fspath():
(4)             with PdfPages(Path(os.devnull)) as pdf:
(8)                 pdf.savefig(plt.figure())
(0)         @image_comparison(['hatching_legend.pdf'])
(0)         def test_hatching_legend():
(4)             """Test for correct hatching on patches in legend"""
(4)             fig = plt.figure(figsize=(1, 2))
(4)             a = Rectangle([0, 0], 0, 0, facecolor="green", hatch="XXXX")
(4)             b = Rectangle([0, 0], 0, 0, facecolor="blue", hatch="XXXX")
(4)             fig.legend([a, b, a, b], ["", "", "", ""])
(0)         @image_comparison(['grayscale_alpha.pdf'])
(0)         def test_grayscale_alpha():
(4)             """Masking images with NaN did not work for grayscale images"""
(4)             x, y = np.ogrid[-2:2:.1, -2:2:.1]
(4)             dd = np.exp(-(x**2 + y**2))
(4)             dd[dd < .1] = np.nan
(4)             fig, ax = plt.subplots()
```

```
(4)                 ax.imshow(dd, interpolation='none', cmap='gray_r')
(4)                 ax.set_xticks([])
(4)                 ax.set_yticks([])
(0)             @mpl.style.context('default')
(0)             @check_figures_equal(extensions=["pdf", "eps"])
(0)             def test_pdf_eps_savefig_when_color_is_none(fig_test, fig_ref):
(4)                 ax_test = fig_test.add_subplot()
(4)                 ax_test.set_axis_off()
(4)                 ax_test.plot(np.sin(np.linspace(-5, 5, 100)), "v", c="none")
(4)                 ax_ref = fig_ref.add_subplot()
(4)                 ax_ref.set_axis_off()
(0)             @needs_usetex
(0)             def test_failing_latex():
(4)                 """Test failing latex subprocess call"""
(4)                 plt.xlabel("$22_2_2$", usetex=True)  # This fails with "Double subscript"
(4)                 with pytest.raises(RuntimeError):
(8)                     plt.savefig(io.BytesIO(), format="pdf")
(0)             def test_empty_rasterized():
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot([], [], rasterized=True)
(4)                 fig.savefig(io.BytesIO(), format="pdf")
(0)             @image_comparison(['kerning.pdf'])
(0)             def test_kerning():
(4)                 fig = plt.figure()
(4)                 s = "AVAVAVAVAVAVAVAV€AAVV"
(4)                 fig.text(0, .25, s, size=5)
(4)                 fig.text(0, .75, s, size=20)
(0)             def test_glyphs_subset():
(4)                 fpath = str(_get_data_path("fonts/ttf/DejaVuSerif.ttf"))
(4)                 chars = "these should be subsetted! 1234567890"
(4)                 nosubfont = FT2Font(fpath)
(4)                 nosubfont.set_text(chars)
(4)                 subfont = FT2Font(get_glyphs_subset(fpath, chars))
(4)                 subfont.set_text(chars)
(4)                 nosubcmap = nosubfont.get_charmap()
(4)                 subcmap = subfont.get_charmap()
(4)                 assert {*chars} == {chr(key) for key in subcmap}
(4)                 assert len(subcmap) < len(nosubcmap)
(4)                 assert subfont.get_num_glyphs() == nosubfont.get_num_glyphs()
(0)             @image_comparison(["multi_font_type3.pdf"], tol=4.6)
(0)             def test_multi_font_type3():
(4)                 fp = fm.FontProperties(family=["WenQuanYi Zen Hei"])
(4)                 if Path(fm.findfont(fp)).name != "wqy-zenhei.ttc":
(8)                     pytest.skip("Font may be missing")
(4)                 plt.rc('font', family=['DejaVu Sans', 'WenQuanYi Zen Hei'], size=27)
(4)                 plt.rc('pdf', fonttype=3)
(4)                 fig = plt.figure()
(4)                 fig.text(0.15, 0.475, "There are 几个汉字 in between!")
(0)             @image_comparison(["multi_font_type42.pdf"], tol=2.2)
(0)             def test_multi_font_type42():
(4)                 fp = fm.FontProperties(family=["WenQuanYi Zen Hei"])
(4)                 if Path(fm.findfont(fp)).name != "wqy-zenhei.ttc":
(8)                     pytest.skip("Font may be missing")
(4)                 plt.rc('font', family=['DejaVu Sans', 'WenQuanYi Zen Hei'], size=27)
(4)                 plt.rc('pdf', fonttype=42)
(4)                 fig = plt.figure()
(4)                 fig.text(0.15, 0.475, "There are 几个汉字 in between!")
(0)             @pytest.mark.parametrize('family_name, file_name',
(25)                                         [("Noto Sans", "NotoSans-Regular.otf"),
(26)                                          ("FreeMono", "FreeMono.otf")])
(0)             def test_otf_font_smoke(family_name, file_name):
(4)                 fp = fm.FontProperties(family=[family_name])
(4)                 if Path(fm.findfont(fp)).name != file_name:
(8)                     pytest.skip(f"Font {family_name} may be missing")
(4)                 plt.rc('font', family=[family_name], size=27)
(4)                 fig = plt.figure()
(4)                 fig.text(0.15, 0.475, "Привет мир!")
(4)                 fig.savefig(io.BytesIO(), format="pdf")
```

```
----------------------------------------

File 158 - test_backend_ps.py:

1: (0)              from collections import Counter
2: (0)              from pathlib import Path
3: (0)              import io
4: (0)              import re
5: (0)              import tempfile
6: (0)              import numpy as np
7: (0)              import pytest
8: (0)              from matplotlib import cbook, path, patheffects, font_manager as fm
9: (0)              from matplotlib.figure import Figure
10: (0)             from matplotlib.patches import Ellipse
11: (0)             from matplotlib.testing._markers import needs_ghostscript, needs_usetex
12: (0)             from matplotlib.testing.decorators import check_figures_equal,
image_comparison
13: (0)             import matplotlib as mpl
14: (0)             import matplotlib.collections as mcollections
15: (0)             import matplotlib.colors as mcolors
16: (0)             import matplotlib.pyplot as plt
17: (0)             @pytest.mark.flaky(reruns=3)
18: (0)             @pytest.mark.parametrize('papersize', ['letter', 'figure'])
19: (0)             @pytest.mark.parametrize('orientation', ['portrait', 'landscape'])
20: (0)             @pytest.mark.parametrize('format, use_log, rcParams', [
21: (4)                 ('ps', False, {}),
22: (4)                 ('ps', False, {'ps.usedistiller': 'ghostscript'}),
23: (4)                 ('ps', False, {'ps.usedistiller': 'xpdf'}),
24: (4)                 ('ps', False, {'text.usetex': True}),
25: (4)                 ('eps', False, {}),
26: (4)                 ('eps', True, {'ps.useafm': True}),
27: (4)                 ('eps', False, {'text.usetex': True}),
28: (0)             ], ids=[
29: (4)                 'ps',
30: (4)                 'ps with distiller=ghostscript',
31: (4)                 'ps with distiller=xpdf',
32: (4)                 'ps with usetex',
33: (4)                 'eps',
34: (4)                 'eps afm',
35: (4)                 'eps with usetex'
36: (0)             ])
37: (0)             def test_savefig_to_stringio(format, use_log, rcParams, orientation,
papersize):
38: (4)                 mpl.rcParams.update(rcParams)
39: (4)                 if mpl.rcParams["ps.usedistiller"] == "ghostscript":
40: (8)                     try:
41: (12)                        mpl._get_executable_info("gs")
42: (8)                     except mpl.ExecutableNotFoundError as exc:
43: (12)                        pytest.skip(str(exc))
44: (4)                 elif mpl.rcParams["ps.usedistiller"] == "xpdf":
45: (8)                     try:
46: (12)                        mpl._get_executable_info("gs")  # Effectively checks for ps2pdf.
47: (12)                        mpl._get_executable_info("pdftops")
48: (8)                     except mpl.ExecutableNotFoundError as exc:
49: (12)                        pytest.skip(str(exc))
50: (4)                 fig, ax = plt.subplots()
51: (4)                 with io.StringIO() as s_buf, io.BytesIO() as b_buf:
52: (8)                     if use_log:
53: (12)                        ax.set_yscale('log')
54: (8)                     ax.plot([1, 2], [1, 2])
55: (8)                     title = "Déjà vu"
56: (8)                     if not mpl.rcParams["text.usetex"]:
57: (12)                        title += " \N{MINUS SIGN}\N{EURO SIGN}"
58: (8)                     ax.set_title(title)
59: (8)                     allowable_exceptions = []
60: (8)                     if mpl.rcParams["text.usetex"]:
61: (12)                        allowable_exceptions.append(RuntimeError)
62: (8)                     if mpl.rcParams["ps.useafm"]:
63: (12)                        allowable_exceptions.append(mpl.MatplotlibDeprecationWarning)
```

```
 64: (8)                    try:
 65: (12)                       fig.savefig(s_buf, format=format, orientation=orientation,
 66: (24)                                   papertype=papersize)
 67: (12)                       fig.savefig(b_buf, format=format, orientation=orientation,
 68: (24)                                   papertype=papersize)
 69: (8)                    except tuple(allowable_exceptions) as exc:
 70: (12)                       pytest.skip(str(exc))
 71: (8)                    assert not s_buf.closed
 72: (8)                    assert not b_buf.closed
 73: (8)                    s_val = s_buf.getvalue().encode('ascii')
 74: (8)                    b_val = b_buf.getvalue()
 75: (8)                    if format == 'ps':
 76: (12)                       if mpl.rcParams["ps.usedistiller"] == "xpdf":
 77: (16)                           if papersize == 'figure':
 78: (20)                               assert b'letter' not in s_val.lower()
 79: (16)                           else:
 80: (20)                               assert b'letter' in s_val.lower()
 81: (12)                       elif mpl.rcParams["ps.usedistiller"] or
mpl.rcParams["text.usetex"]:
 82: (16)                           width = b'432.0' if orientation == 'landscape' else b'576.0'
 83: (16)                           wanted = (b'-dDEVICEWIDTHPOINTS=' + width if papersize ==
'figure'
 84: (26)                                     else b'-sPAPERSIZE')
 85: (16)                           assert wanted in s_val
 86: (12)                       else:
 87: (16)                           if papersize == 'figure':
 88: (20)                               assert b'%%DocumentPaperSizes' not in s_val
 89: (16)                           else:
 90: (20)                               assert b'%%DocumentPaperSizes' in s_val
 91: (8)                    s_val = re.sub(b"(?<=\n%%CreationDate: ).*", b"", s_val)
 92: (8)                    b_val = re.sub(b"(?<=\n%%CreationDate: ).*", b"", b_val)
 93: (8)                    assert s_val == b_val.replace(b'\r\n', b'\n')
 94: (0)            def test_patheffects():
 95: (4)                mpl.rcParams['path.effects'] = [
 96: (8)                    patheffects.withStroke(linewidth=4, foreground='w')]
 97: (4)                fig, ax = plt.subplots()
 98: (4)                ax.plot([1, 2, 3])
 99: (4)                with io.BytesIO() as ps:
100: (8)                    fig.savefig(ps, format='ps')
101: (0)            @needs_usetex
102: (0)            @needs_ghostscript
103: (0)            def test_tilde_in_tempfilename(tmp_path):
104: (4)                base_tempdir = tmp_path / "short-1"
105: (4)                base_tempdir.mkdir()
106: (4)                with cbook._setattr_cm(tempfile, tempdir=str(base_tempdir)):
107: (8)                    mpl.rcParams['text.usetex'] = True
108: (8)                    plt.plot([1, 2, 3, 4])
109: (8)                    plt.xlabel(r'\textbf{time} (s)')
110: (8)                    plt.savefig(base_tempdir / 'tex_demo.eps', format="ps")
111: (0)            @image_comparison(["empty.eps"])
112: (0)            def test_transparency():
113: (4)                fig, ax = plt.subplots()
114: (4)                ax.set_axis_off()
115: (4)                ax.plot([0, 1], color="r", alpha=0)
116: (4)                ax.text(.5, .5, "foo", color="r", alpha=0)
117: (0)            @needs_usetex
118: (0)            @image_comparison(["empty.eps"])
119: (0)            def test_transparency_tex():
120: (4)                mpl.rcParams['text.usetex'] = True
121: (4)                fig, ax = plt.subplots()
122: (4)                ax.set_axis_off()
123: (4)                ax.plot([0, 1], color="r", alpha=0)
124: (4)                ax.text(.5, .5, "foo", color="r", alpha=0)
125: (0)            def test_bbox():
126: (4)                fig, ax = plt.subplots()
127: (4)                with io.BytesIO() as buf:
128: (8)                    fig.savefig(buf, format='eps')
129: (8)                    buf = buf.getvalue()
130: (4)                bb = re.search(b'^%%BoundingBox: (.+) (.+) (.+) (.+)$', buf, re.MULTILINE)
```

```
(4)                 assert bb
(4)                 hibb = re.search(b'^%%HiResBoundingBox: (.+) (.+) (.+) (.+)$', buf,
(21)                                 re.MULTILINE)
(4)                 assert hibb
(4)                 for i in range(1, 5):
(8)                     assert b'.' not in bb.group(i)
(8)                     assert int(bb.group(i)) == pytest.approx(float(hibb.group(i)), 1)
(0)             @needs_usetex
(0)             def test_failing_latex():
(4)                 """Test failing latex subprocess call"""
(4)                 mpl.rcParams['text.usetex'] = True
(4)                 plt.xlabel("$22_2_2$")
(4)                 with pytest.raises(RuntimeError):
(8)                     plt.savefig(io.BytesIO(), format="ps")
(0)             @needs_usetex
(0)             def test_partial_usetex(caplog):
(4)                 caplog.set_level("WARNING")
(4)                 plt.figtext(.1, .1, "foo", usetex=True)
(4)                 plt.figtext(.2, .2, "bar", usetex=True)
(4)                 plt.savefig(io.BytesIO(), format="ps")
(4)                 record, = caplog.records  # asserts there's a single record.
(4)                 assert "as if usetex=False" in record.getMessage()
(0)             @needs_usetex
(0)             def test_usetex_preamble(caplog):
(4)                 mpl.rcParams.update({
(8)                     "text.usetex": True,
(8)                     "text.latex.preamble": r"\usepackage{color,graphicx,textcomp}",
(4)                 })
(4)                 plt.figtext(.5, .5, "foo")
(4)                 plt.savefig(io.BytesIO(), format="ps")
(0)             @image_comparison(["useafm.eps"])
(0)             def test_useafm():
(4)                 mpl.rcParams["ps.useafm"] = True
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_axis_off()
(4)                 ax.axhline(.5)
(4)                 ax.text(.5, .5, "qk")
(0)             @image_comparison(["type3.eps"])
(0)             def test_type3_font():
(4)                 plt.figtext(.5, .5, "I/J")
(0)             @image_comparison(["coloredhatcheszerolw.eps"])
(0)             def test_colored_hatch_zero_linewidth():
(4)                 ax = plt.gca()
(4)                 ax.add_patch(Ellipse((0, 0), 1, 1, hatch='/', facecolor='none',
(25)                                     edgecolor='r', linewidth=0))
(4)                 ax.add_patch(Ellipse((0.5, 0.5), 0.5, 0.5, hatch='+', facecolor='none',
(25)                                     edgecolor='g', linewidth=0.2))
(4)                 ax.add_patch(Ellipse((1, 1), 0.3, 0.8, hatch='\\', facecolor='none',
(25)                                     edgecolor='b', linewidth=0))
(4)                 ax.set_axis_off()
(0)             @check_figures_equal(extensions=["eps"])
(0)             def test_text_clip(fig_test, fig_ref):
(4)                 ax = fig_test.add_subplot()
(4)                 ax.text(0, 0, "hello", transform=fig_test.transFigure, clip_on=True)
(4)                 fig_ref.add_subplot()
(0)             @needs_ghostscript
(0)             def test_d_glyph(tmp_path):
(4)                 fig = plt.figure()
(4)                 fig.text(.5, .5, "def")
(4)                 out = tmp_path / "test.eps"
(4)                 fig.savefig(out)
(4)                 mpl.testing.compare.convert(out, cache=False)  # Should not raise.
(0)             @image_comparison(["type42_without_prep.eps"], style='mpl20')
(0)             def test_type42_font_without_prep():
(4)                 mpl.rcParams["ps.fonttype"] = 42
(4)                 mpl.rcParams["mathtext.fontset"] = "stix"
(4)                 plt.figtext(0.5, 0.5, "Mass $m$")
(0)             @pytest.mark.parametrize('fonttype', ["3", "42"])
(0)             def test_fonttype(fonttype):
```

```
200: (4)                mpl.rcParams["ps.fonttype"] = fonttype
201: (4)                fig, ax = plt.subplots()
202: (4)                ax.text(0.25, 0.5, "Forty-two is the answer to everything!")
203: (4)                buf = io.BytesIO()
204: (4)                fig.savefig(buf, format="ps")
205: (4)                test = b'/FontType ' + bytes(f"{fonttype}", encoding='utf-8') + b' def'
206: (4)                assert re.search(test, buf.getvalue(), re.MULTILINE)
207: (0)            def test_linedash():
208: (4)                """Test that dashed lines do not break PS output"""
209: (4)                fig, ax = plt.subplots()
210: (4)                ax.plot([0, 1], linestyle="--")
211: (4)                buf = io.BytesIO()
212: (4)                fig.savefig(buf, format="ps")
213: (4)                assert buf.tell() > 0
214: (0)            def test_empty_line():
215: (4)                figure = Figure()
216: (4)                figure.text(0.5, 0.5, "\nfoo\n\n")
217: (4)                buf = io.BytesIO()
218: (4)                figure.savefig(buf, format='eps')
219: (4)                figure.savefig(buf, format='ps')
220: (0)            def test_no_duplicate_definition():
221: (4)                fig = Figure()
222: (4)                axs = fig.subplots(4, 4, subplot_kw=dict(projection="polar"))
223: (4)                for ax in axs.flat:
224: (8)                    ax.set(xticks=[], yticks=[])
225: (8)                    ax.plot([1, 2])
226: (4)                fig.suptitle("hello, world")
227: (4)                buf = io.StringIO()
228: (4)                fig.savefig(buf, format='eps')
229: (4)                buf.seek(0)
230: (4)                wds = [ln.partition(' ')[0] for
231: (11)                      ln in buf.readlines()
232: (11)                      if ln.startswith('/')]
233: (4)                assert max(Counter(wds).values()) == 1
234: (0)            @image_comparison(["multi_font_type3.eps"], tol=0.51)
235: (0)            def test_multi_font_type3():
236: (4)                fp = fm.FontProperties(family=["WenQuanYi Zen Hei"])
237: (4)                if Path(fm.findfont(fp)).name != "wqy-zenhei.ttc":
238: (8)                    pytest.skip("Font may be missing")
239: (4)                plt.rc('font', family=['DejaVu Sans', 'WenQuanYi Zen Hei'], size=27)
240: (4)                plt.rc('ps', fonttype=3)
241: (4)                fig = plt.figure()
242: (4)                fig.text(0.15, 0.475, "There are 几个汉字 in between!")
243: (0)            @image_comparison(["multi_font_type42.eps"], tol=1.6)
244: (0)            def test_multi_font_type42():
245: (4)                fp = fm.FontProperties(family=["WenQuanYi Zen Hei"])
246: (4)                if Path(fm.findfont(fp)).name != "wqy-zenhei.ttc":
247: (8)                    pytest.skip("Font may be missing")
248: (4)                plt.rc('font', family=['DejaVu Sans', 'WenQuanYi Zen Hei'], size=27)
249: (4)                plt.rc('ps', fonttype=42)
250: (4)                fig = plt.figure()
251: (4)                fig.text(0.15, 0.475, "There are 几个汉字 in between!")
252: (0)            @image_comparison(["scatter.eps"])
253: (0)            def test_path_collection():
254: (4)                rng = np.random.default_rng(19680801)
255: (4)                xvals = rng.uniform(0, 1, 10)
256: (4)                yvals = rng.uniform(0, 1, 10)
257: (4)                sizes = rng.uniform(30, 100, 10)
258: (4)                fig, ax = plt.subplots()
259: (4)                ax.scatter(xvals, yvals, sizes, edgecolor=[0.9, 0.2, 0.1], marker='<')
260: (4)                ax.set_axis_off()
261: (4)                paths = [path.Path.unit_regular_polygon(i) for i in range(3, 7)]
262: (4)                offsets = rng.uniform(0, 200, 20).reshape(10, 2)
263: (4)                sizes = [0.02, 0.04]
264: (4)                pc = mcollections.PathCollection(paths, sizes, zorder=-1,
265: (37)                                                facecolors='yellow', offsets=offsets)
266: (4)                ax.add_collection(pc)
267: (4)                ax.set_xlim(0, 1)
268: (0)            @image_comparison(["colorbar_shift.eps"], savefig_kwarg={"bbox_inches":
```

```
"tight"},
269: (18)                               style="mpl20")
270: (0)            def test_colorbar_shift(tmp_path):
271: (4)                cmap = mcolors.ListedColormap(["r", "g", "b"])
272: (4)                norm = mcolors.BoundaryNorm([-1, -0.5, 0.5, 1], cmap.N)
273: (4)                plt.scatter([0, 1], [1, 1], c=[0, 1], cmap=cmap, norm=norm)
274: (4)                plt.colorbar()
275: (0)            def test_auto_papersize_deprecation():
276: (4)                fig = plt.figure()
277: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning):
278: (8)                    fig.savefig(io.BytesIO(), format='eps', papertype='auto')
279: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning):
280: (8)                    mpl.rcParams['ps.papersize'] = 'auto'

----------------------------------------

File 159 - test_backend_pgf.py:

1: (0)              import datetime
2: (0)              from io import BytesIO
3: (0)              import os
4: (0)              import shutil
5: (0)              import numpy as np
6: (0)              from packaging.version import parse as parse_version
7: (0)              import pytest
8: (0)              import matplotlib as mpl
9: (0)              import matplotlib.pyplot as plt
10: (0)             from matplotlib.testing import _has_tex_package, _check_for_pgf
11: (0)             from matplotlib.testing.exceptions import ImageComparisonFailure
12: (0)             from matplotlib.testing.compare import compare_images
13: (0)             from matplotlib.backends.backend_pgf import PdfPages
14: (0)             from matplotlib.testing.decorators import (
15: (4)                 _image_directories, check_figures_equal, image_comparison)
16: (0)             from matplotlib.testing._markers import (
17: (4)                 needs_ghostscript, needs_pgf_lualatex, needs_pgf_pdflatex,
18: (4)                 needs_pgf_xelatex)
19: (0)             baseline_dir, result_dir = _image_directories(lambda: 'dummy func')
20: (0)             def compare_figure(fname, savefig_kwargs={}, tol=0):
21: (4)                 actual = os.path.join(result_dir, fname)
22: (4)                 plt.savefig(actual, **savefig_kwargs)
23: (4)                 expected = os.path.join(result_dir, "expected_%s" % fname)
24: (4)                 shutil.copyfile(os.path.join(baseline_dir, fname), expected)
25: (4)                 err = compare_images(expected, actual, tol=tol)
26: (4)                 if err:
27: (8)                     raise ImageComparisonFailure(err)
28: (0)             @needs_pgf_xelatex
29: (0)             @needs_ghostscript
30: (0)             @pytest.mark.backend('pgf')
31: (0)             def test_tex_special_chars(tmp_path):
32: (4)                 fig = plt.figure()
33: (4)                 fig.text(.5, .5, "%_^ $a_b^c$")
34: (4)                 buf = BytesIO()
35: (4)                 fig.savefig(buf, format="png", backend="pgf")
36: (4)                 buf.seek(0)
37: (4)                 t = plt.imread(buf)
38: (4)                 assert not (t == 1).all()  # The leading "%" didn't eat up everything.
39: (0)             def create_figure():
40: (4)                 plt.figure()
41: (4)                 x = np.linspace(0, 1, 15)
42: (4)                 plt.plot(x, x ** 2, "b-")
43: (4)                 plt.plot(x, 1 - x**2, "g>")
44: (4)                 plt.fill_between([0., .4], [.4, 0.], hatch='//', facecolor="lightgray",
45: (21)                                 edgecolor="red")
46: (4)                 plt.fill([3, 3, .8, .8, 3], [2, -2, -2, 0, 2], "b")
47: (4)                 plt.plot([0.9], [0.5], "ro", markersize=3)
48: (4)                 plt.text(0.9, 0.5, 'unicode (ü, °, \N{Section Sign}) and math ($\\mu_i =
x_i^2$)',
49: (13)                         ha='right', fontsize=20)
50: (4)                 plt.ylabel('sans-serif, blue, $\\frac{\\sqrt{x}}{y^2}$..',
```

```
 51: (15)                            family='sans-serif', color='blue')
 52: (4)                 plt.text(1, 1, 'should be clipped as default clip_box is Axes bbox',
 53: (13)                         fontsize=20, clip_on=True)
 54: (4)                 plt.xlim(0, 1)
 55: (4)                 plt.ylim(0, 1)
 56: (0)             @needs_pgf_xelatex
 57: (0)             @pytest.mark.backend('pgf')
 58: (0)             @image_comparison(['pgf_xelatex.pdf'], style='default')
 59: (0)             def test_xelatex():
 60: (4)                 rc_xelatex = {'font.family': 'serif',
 61: (18)                              'pgf.rcfonts': False}
 62: (4)                 mpl.rcParams.update(rc_xelatex)
 63: (4)                 create_figure()
 64: (0)             try:
 65: (4)                 _old_gs_version = \
 66: (8)                     mpl._get_executable_info('gs').version < parse_version('9.50')
 67: (0)             except mpl.ExecutableNotFoundError:
 68: (4)                 _old_gs_version = True
 69: (0)             @needs_pgf_pdflatex
 70: (0)             @pytest.mark.skipif(not _has_tex_package('type1ec'), reason='needs
type1ec.sty')
 71: (0)             @pytest.mark.skipif(not _has_tex_package('ucs'), reason='needs ucs.sty')
 72: (0)             @pytest.mark.backend('pgf')
 73: (0)             @image_comparison(['pgf_pdflatex.pdf'], style='default',
 74: (18)                             tol=11.71 if _old_gs_version else 0)
 75: (0)             def test_pdflatex():
 76: (4)                 rc_pdflatex = {'font.family': 'serif',
 77: (19)                               'pgf.rcfonts': False,
 78: (19)                               'pgf.texsystem': 'pdflatex',
 79: (19)                               'pgf.preamble': ('\\usepackage[utf8x]{inputenc}'
 80: (36)                                                '\\usepackage[T1]{fontenc}')}
 81: (4)                 mpl.rcParams.update(rc_pdflatex)
 82: (4)                 create_figure()
 83: (0)             @needs_pgf_xelatex
 84: (0)             @needs_pgf_pdflatex
 85: (0)             @mpl.style.context('default')
 86: (0)             @pytest.mark.backend('pgf')
 87: (0)             def test_rcupdate():
 88: (4)                 rc_sets = [{'font.family': 'sans-serif',
 89: (16)                            'font.size': 30,
 90: (16)                            'figure.subplot.left': .2,
 91: (16)                            'lines.markersize': 10,
 92: (16)                            'pgf.rcfonts': False,
 93: (16)                            'pgf.texsystem': 'xelatex'},
 94: (15)                           {'font.family': 'monospace',
 95: (16)                            'font.size': 10,
 96: (16)                            'figure.subplot.left': .1,
 97: (16)                            'lines.markersize': 20,
 98: (16)                            'pgf.rcfonts': False,
 99: (16)                            'pgf.texsystem': 'pdflatex',
100: (16)                            'pgf.preamble': ('\\usepackage[utf8x]{inputenc}'
101: (33)                                             '\\usepackage[T1]{fontenc}'
102: (33)                                             '\\usepackage{sfmath}')}]
103: (4)                 tol = [0, 13.2] if _old_gs_version else [0, 0]
104: (4)                 for i, rc_set in enumerate(rc_sets):
105: (8)                     with mpl.rc_context(rc_set):
106: (12)                        for substring, pkg in [('sfmath', 'sfmath'), ('utf8x', 'ucs')]:
107: (16)                            if (substring in mpl.rcParams['pgf.preamble']
108: (24)                                    and not _has_tex_package(pkg)):
109: (20)                                pytest.skip(f'needs {pkg}.sty')
110: (12)                        create_figure()
111: (12)                        compare_figure(f'pgf_rcupdate{i + 1}.pdf', tol=tol[i])
112: (0)             @needs_pgf_xelatex
113: (0)             @mpl.style.context('default')
114: (0)             @pytest.mark.backend('pgf')
115: (0)             def test_pathclip():
116: (4)                 np.random.seed(19680801)
117: (4)                 mpl.rcParams.update({'font.family': 'serif', 'pgf.rcfonts': False})
118: (4)                 fig, axs = plt.subplots(1, 2)
```

```
119: (4)                axs[0].plot([0., 1e100], [0., 1e100])
120: (4)                axs[0].set_xlim(0, 1)
121: (4)                axs[0].set_ylim(0, 1)
122: (4)                axs[1].scatter([0, 1], [1, 1])
123: (4)                axs[1].hist(np.random.normal(size=1000), bins=20, range=[-10, 10])
124: (4)                axs[1].set_xscale('log')
125: (4)                fig.savefig(BytesIO(), format="pdf")  # No image comparison.
126: (0)            @needs_pgf_xelatex
127: (0)            @pytest.mark.backend('pgf')
128: (0)            @image_comparison(['pgf_mixedmode.pdf'], style='default')
129: (0)            def test_mixedmode():
130: (4)                mpl.rcParams.update({'font.family': 'serif', 'pgf.rcfonts': False})
131: (4)                Y, X = np.ogrid[-1:1:40j, -1:1:40j]
132: (4)                plt.pcolor(X**2 + Y**2).set_rasterized(True)
133: (0)            @needs_pgf_xelatex
134: (0)            @mpl.style.context('default')
135: (0)            @pytest.mark.backend('pgf')
136: (0)            def test_bbox_inches():
137: (4)                mpl.rcParams.update({'font.family': 'serif', 'pgf.rcfonts': False})
138: (4)                fig, (ax1, ax2) = plt.subplots(1, 2)
139: (4)                ax1.plot(range(5))
140: (4)                ax2.plot(range(5))
141: (4)                plt.tight_layout()
142: (4)                bbox = ax1.get_window_extent().transformed(fig.dpi_scale_trans.inverted())
143: (4)                compare_figure('pgf_bbox_inches.pdf', savefig_kwargs={'bbox_inches':
bbox},
144: (19)                               tol=0)
145: (0)            @mpl.style.context('default')
146: (0)            @pytest.mark.backend('pgf')
147: (0)            @pytest.mark.parametrize('system', [
148: (4)                pytest.param('lualatex', marks=[needs_pgf_lualatex]),
149: (4)                pytest.param('pdflatex', marks=[needs_pgf_pdflatex]),
150: (4)                pytest.param('xelatex', marks=[needs_pgf_xelatex]),
151: (0)            ])
152: (0)            def test_pdf_pages(system):
153: (4)                rc_pdflatex = {
154: (8)                    'font.family': 'serif',
155: (8)                    'pgf.rcfonts': False,
156: (8)                    'pgf.texsystem': system,
157: (4)                }
158: (4)                mpl.rcParams.update(rc_pdflatex)
159: (4)                fig1, ax1 = plt.subplots()
160: (4)                ax1.plot(range(5))
161: (4)                fig1.tight_layout()
162: (4)                fig2, ax2 = plt.subplots(figsize=(3, 2))
163: (4)                ax2.plot(range(5))
164: (4)                fig2.tight_layout()
165: (4)                path = os.path.join(result_dir, f'pdfpages_{system}.pdf')
166: (4)                md = {
167: (8)                    'Author': 'me',
168: (8)                    'Title': 'Multipage PDF with pgf',
169: (8)                    'Subject': 'Test page',
170: (8)                    'Keywords': 'test,pdf,multipage',
171: (8)                    'ModDate': datetime.datetime(
172: (12)                       1968, 8, 1, tzinfo=datetime.timezone(datetime.timedelta(0))),
173: (8)                    'Trapped': 'Unknown'
174: (4)                }
175: (4)                with PdfPages(path, metadata=md) as pdf:
176: (8)                    pdf.savefig(fig1)
177: (8)                    pdf.savefig(fig2)
178: (8)                    pdf.savefig(fig1)
179: (8)                    assert pdf.get_pagecount() == 3
180: (0)            @mpl.style.context('default')
181: (0)            @pytest.mark.backend('pgf')
182: (0)            @pytest.mark.parametrize('system', [
183: (4)                pytest.param('lualatex', marks=[needs_pgf_lualatex]),
184: (4)                pytest.param('pdflatex', marks=[needs_pgf_pdflatex]),
185: (4)                pytest.param('xelatex', marks=[needs_pgf_xelatex]),
186: (0)            ])
```

```
(0)             def test_pdf_pages_metadata_check(monkeypatch, system):
(4)                 pikepdf = pytest.importorskip('pikepdf')
(4)                 monkeypatch.setenv('SOURCE_DATE_EPOCH', '0')
(4)                 mpl.rcParams.update({'pgf.texsystem': system})
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(range(5))
(4)                 md = {
(8)                     'Author': 'me',
(8)                     'Title': 'Multipage PDF with pgf',
(8)                     'Subject': 'Test page',
(8)                     'Keywords': 'test,pdf,multipage',
(8)                     'ModDate': datetime.datetime(
(12)                        1968, 8, 1, tzinfo=datetime.timezone(datetime.timedelta(0))),
(8)                     'Trapped': 'True'
(4)                 }
(4)                 path = os.path.join(result_dir, f'pdfpages_meta_check_{system}.pdf')
(4)                 with PdfPages(path, metadata=md) as pdf:
(8)                     pdf.savefig(fig)
(4)                 with pikepdf.Pdf.open(path) as pdf:
(8)                     info = {k: str(v) for k, v in pdf.docinfo.items()}
(4)                 if '/PTEX.FullBanner' in info:
(8)                     del info['/PTEX.FullBanner']
(4)                 if '/PTEX.Fullbanner' in info:
(8)                     del info['/PTEX.Fullbanner']
(4)                 producer = info.pop('/Producer')
(4)                 assert producer == f'Matplotlib pgf backend v{mpl.__version__}' or (
(12)                        system == 'lualatex' and 'LuaTeX' in producer)
(4)                 assert info == {
(8)                     '/Author': 'me',
(8)                     '/CreationDate': 'D:19700101000000Z',
(8)                     '/Creator': f'Matplotlib v{mpl.__version__}, https://matplotlib.org',
(8)                     '/Keywords': 'test,pdf,multipage',
(8)                     '/ModDate': 'D:19680801000000Z',
(8)                     '/Subject': 'Test page',
(8)                     '/Title': 'Multipage PDF with pgf',
(8)                     '/Trapped': '/True',
(4)                 }
(0)             @needs_pgf_xelatex
(0)             def test_multipage_keep_empty(tmp_path):
(4)                 fn = tmp_path / "a.pdf"
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning), PdfPages(fn) as pdf:
(8)                     pass
(4)                 assert fn.exists()
(4)                 fn = tmp_path / "b.pdf"
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning), \
(12)                        PdfPages(fn, keep_empty=True) as pdf:
(8)                     pass
(4)                 assert fn.exists()
(4)                 fn = tmp_path / "c.pdf"
(4)                 with PdfPages(fn, keep_empty=False) as pdf:
(8)                     pass
(4)                 assert not fn.exists()
(4)                 fn = tmp_path / "d.pdf"
(4)                 with PdfPages(fn) as pdf:
(8)                     pdf.savefig(plt.figure())
(4)                 assert fn.exists()
(4)                 fn = tmp_path / "e.pdf"
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning), \
(12)                        PdfPages(fn, keep_empty=True) as pdf:
(8)                     pdf.savefig(plt.figure())
(4)                 assert fn.exists()
(4)                 fn = tmp_path / "f.pdf"
(4)                 with PdfPages(fn, keep_empty=False) as pdf:
(8)                     pdf.savefig(plt.figure())
(4)                 assert fn.exists()
(0)             @needs_pgf_xelatex
(0)             def test_tex_restart_after_error():
(4)                 fig = plt.figure()
(4)                 fig.suptitle(r"\oops")
```

```
256: (4)                with pytest.raises(ValueError):
257: (8)                    fig.savefig(BytesIO(), format="pgf")
258: (4)                fig = plt.figure()  # start from scratch
259: (4)                fig.suptitle(r"this is ok")
260: (4)                fig.savefig(BytesIO(), format="pgf")
261: (0)            @needs_pgf_xelatex
262: (0)            def test_bbox_inches_tight():
263: (4)                fig, ax = plt.subplots()
264: (4)                ax.imshow([[0, 1], [2, 3]])
265: (4)                fig.savefig(BytesIO(), format="pdf", backend="pgf", bbox_inches="tight")
266: (0)            @needs_pgf_xelatex
267: (0)            @needs_ghostscript
268: (0)            def test_png_transparency():  # Actually, also just testing that png works.
269: (4)                buf = BytesIO()
270: (4)                plt.figure().savefig(buf, format="png", backend="pgf", transparent=True)
271: (4)                buf.seek(0)
272: (4)                t = plt.imread(buf)
273: (4)                assert (t[..., 3] == 0).all()  # fully transparent.
274: (0)            @needs_pgf_xelatex
275: (0)            def test_unknown_font(caplog):
276: (4)                with caplog.at_level("WARNING"):
277: (8)                    mpl.rcParams["font.family"] = "this-font-does-not-exist"
278: (8)                    plt.figtext(.5, .5, "hello, world")
279: (8)                    plt.savefig(BytesIO(), format="pgf")
280: (4)                assert "Ignoring unknown font: this-font-does-not-exist" in [
281: (8)                    r.getMessage() for r in caplog.records]
282: (0)            @check_figures_equal(extensions=["pdf"])
283: (0)            @pytest.mark.parametrize("texsystem", ("pdflatex", "xelatex", "lualatex"))
284: (0)            @pytest.mark.backend("pgf")
285: (0)            def test_minus_signs_with_tex(fig_test, fig_ref, texsystem):
286: (4)                if not _check_for_pgf(texsystem):
287: (8)                    pytest.skip(texsystem + ' + pgf is required')
288: (4)                mpl.rcParams["pgf.texsystem"] = texsystem
289: (4)                fig_test.text(.5, .5, "$-1$")
290: (4)                fig_ref.text(.5, .5, "$\N{MINUS SIGN}1$")
291: (0)            @pytest.mark.backend("pgf")
292: (0)            def test_sketch_params():
293: (4)                fig, ax = plt.subplots(figsize=(3, 3))
294: (4)                ax.set_xticks([])
295: (4)                ax.set_yticks([])
296: (4)                ax.set_frame_on(False)
297: (4)                handle, = ax.plot([0, 1])
298: (4)                handle.set_sketch_params(scale=5, length=30, randomness=42)
299: (4)                with BytesIO() as fd:
300: (8)                    fig.savefig(fd, format='pgf')
301: (8)                    buf = fd.getvalue().decode()
302: (4)                baseline = r"""\pgfpathmoveto{\pgfqpoint{0.375000in}{0.300000in}}%
303: (0)            \pgfpathlineto{\pgfqpoint{2.700000in}{2.700000in}}%
304: (0)            \usepgfmodule{decorations}%
305: (0)            \usepgflibrary{decorations.pathmorphing}%
306: (0)            \pgfkeys{/pgf/decoration/.cd, """ \
307: (4)                r"""segment length = 0.150000in, amplitude = 0.100000in}%
308: (0)            \pgfmathsetseed{42}%
309: (0)            \pgfdecoratecurrentpath{random steps}%
310: (0)            \pgfusepath{stroke}%"""
311: (4)                assert baseline in buf
```

File 160 - test_backend_qt.py:

```
1: (0)              import copy
2: (0)              import importlib
3: (0)              import os
4: (0)              import signal
5: (0)              import sys
6: (0)              from datetime import date, datetime
7: (0)              from unittest import mock
8: (0)              import pytest
```

```
 9: (0)              import matplotlib
10: (0)              from matplotlib import pyplot as plt
11: (0)              from matplotlib._pylab_helpers import Gcf
12: (0)              from matplotlib import _c_internal_utils
13: (0)              try:
14: (4)                  from matplotlib.backends.qt_compat import QtGui, QtWidgets  # type: ignore
# noqa
15: (4)                  from matplotlib.backends.qt_editor import _formlayout
16: (0)              except ImportError:
17: (4)                  pytestmark = pytest.mark.skip('No usable Qt bindings')
18: (0)              _test_timeout = 60  # A reasonably safe value for slower architectures.
19: (0)              @pytest.fixture
20: (0)              def qt_core(request):
21: (4)                  from matplotlib.backends.qt_compat import QtCore
22: (4)                  return QtCore
23: (0)              @pytest.mark.backend('QtAgg', skip_on_importerror=True)
24: (0)              def test_fig_close():
25: (4)                  init_figs = copy.copy(Gcf.figs)
26: (4)                  fig = plt.figure()
27: (4)                  fig.canvas.manager.window.close()
28: (4)                  assert init_figs == Gcf.figs
29: (0)              @pytest.mark.parametrize(
30: (4)                  "qt_key, qt_mods, answer",
31: (4)                  [
32: (8)                      ("Key_A", ["ShiftModifier"], "A"),
33: (8)                      ("Key_A", [], "a"),
34: (8)                      ("Key_A", ["ControlModifier"], ("ctrl+a")),
35: (8)                      (
36: (12)                         "Key_Aacute",
37: (12)                         ["ShiftModifier"],
38: (12)                         "\N{LATIN CAPITAL LETTER A WITH ACUTE}",
39: (8)                      ),
40: (8)                      ("Key_Aacute", [], "\N{LATIN SMALL LETTER A WITH ACUTE}"),
41: (8)                      ("Key_Control", ["AltModifier"], ("alt+control")),
42: (8)                      ("Key_Alt", ["ControlModifier"], "ctrl+alt"),
43: (8)                      (
44: (12)                         "Key_Aacute",
45: (12)                         ["ControlModifier", "AltModifier", "MetaModifier"],
46: (12)                         ("ctrl+alt+meta+\N{LATIN SMALL LETTER A WITH ACUTE}"),
47: (8)                      ),
48: (8)                      ("Key_Play", [], None),
49: (8)                      ("Key_Backspace", [], "backspace"),
50: (8)                      (
51: (12)                         "Key_Backspace",
52: (12)                         ["ControlModifier"],
53: (12)                         "ctrl+backspace",
54: (8)                      ),
55: (4)                  ],
56: (4)                  ids=[
57: (8)                      'shift',
58: (8)                      'lower',
59: (8)                      'control',
60: (8)                      'unicode_upper',
61: (8)                      'unicode_lower',
62: (8)                      'alt_control',
63: (8)                      'control_alt',
64: (8)                      'modifier_order',
65: (8)                      'non_unicode_key',
66: (8)                      'backspace',
67: (8)                      'backspace_mod',
68: (4)                  ]
69: (0)              )
70: (0)              @pytest.mark.parametrize('backend', [
71: (4)                  pytest.param(
72: (8)                      'Qt5Agg',
73: (8)                      marks=pytest.mark.backend('Qt5Agg', skip_on_importerror=True)),
74: (4)                  pytest.param(
75: (8)                      'QtAgg',
76: (8)                      marks=pytest.mark.backend('QtAgg', skip_on_importerror=True)),
```

```
(0)             ])
(0)             def test_correct_key(backend, qt_core, qt_key, qt_mods, answer, monkeypatch):
(4)                 """
(4)                 Make a figure.
(4)                 Send a key_press_event event (using non-public, qtX backend specific api).
(4)                 Catch the event.
(4)                 Assert sent and caught keys are the same.
(4)                 """
(4)                 from matplotlib.backends.qt_compat import _to_int, QtCore
(4)                 if sys.platform == "darwin" and answer is not None:
(8)                     answer = answer.replace("ctrl", "cmd")
(8)                     answer = answer.replace("control", "cmd")
(8)                     answer = answer.replace("meta", "ctrl")
(4)                 result = None
(4)                 qt_mod = QtCore.Qt.KeyboardModifier.NoModifier
(4)                 for mod in qt_mods:
(8)                     qt_mod |= getattr(QtCore.Qt.KeyboardModifier, mod)
(4)                 class _Event:
(8)                     def isAutoRepeat(self): return False
(8)                     def key(self): return _to_int(getattr(QtCore.Qt.Key, qt_key))
(4)                 monkeypatch.setattr(QtWidgets.QApplication, "keyboardModifiers",
(24)                                    lambda self: qt_mod)
(4)                 def on_key_press(event):
(8)                     nonlocal result
(8)                     result = event.key
(4)                 qt_canvas = plt.figure().canvas
(4)                 qt_canvas.mpl_connect('key_press_event', on_key_press)
(4)                 qt_canvas.keyPressEvent(_Event())
(4)                 assert result == answer
(0)             @pytest.mark.backend('QtAgg', skip_on_importerror=True)
(0)             def test_device_pixel_ratio_change():
(4)                 """
(4)                 Make sure that if the pixel ratio changes, the figure dpi changes but the
(4)                 widget remains the same logical size.
(4)                 """
(4)                 prop = 'matplotlib.backends.backend_qt.FigureCanvasQT.devicePixelRatioF'
(4)                 with mock.patch(prop) as p:
(8)                     p.return_value = 3
(8)                     fig = plt.figure(figsize=(5, 2), dpi=120)
(8)                     qt_canvas = fig.canvas
(8)                     qt_canvas.show()
(8)                     def set_device_pixel_ratio(ratio):
(12)                        p.return_value = ratio
(12)                        screen.logicalDotsPerInchChanged.emit(96)
(12)                        qt_canvas.draw()
(12)                        qt_canvas.flush_events()
(12)                        assert qt_canvas.device_pixel_ratio == ratio
(8)                     qt_canvas.manager.show()
(8)                     size = qt_canvas.size()
(8)                     screen = qt_canvas.window().windowHandle().screen()
(8)                     set_device_pixel_ratio(3)
(8)                     assert fig.dpi == 360
(8)                     assert qt_canvas.renderer.width == 1800
(8)                     assert qt_canvas.renderer.height == 720
(8)                     assert size.width() == 600
(8)                     assert size.height() == 240
(8)                     assert qt_canvas.get_width_height() == (600, 240)
(8)                     assert (fig.get_size_inches() == (5, 2)).all()
(8)                     set_device_pixel_ratio(2)
(8)                     assert fig.dpi == 240
(8)                     assert qt_canvas.renderer.width == 1200
(8)                     assert qt_canvas.renderer.height == 480
(8)                     assert size.width() == 600
(8)                     assert size.height() == 240
(8)                     assert qt_canvas.get_width_height() == (600, 240)
(8)                     assert (fig.get_size_inches() == (5, 2)).all()
(8)                     set_device_pixel_ratio(1.5)
(8)                     assert fig.dpi == 180
(8)                     assert qt_canvas.renderer.width == 900
```

```
(8)                 assert qt_canvas.renderer.height == 360
(8)                 assert size.width() == 600
(8)                 assert size.height() == 240
(8)                 assert qt_canvas.get_width_height() == (600, 240)
(8)                 assert (fig.get_size_inches() == (5, 2)).all()
(0)         @pytest.mark.backend('QtAgg', skip_on_importerror=True)
(0)         def test_subplottool():
(4)             fig, ax = plt.subplots()
(4)             with mock.patch("matplotlib.backends.qt_compat._exec", lambda obj: None):
(8)                 fig.canvas.manager.toolbar.configure_subplots()
(0)         @pytest.mark.backend('QtAgg', skip_on_importerror=True)
(0)         def test_figureoptions():
(4)             fig, ax = plt.subplots()
(4)             ax.plot([1, 2])
(4)             ax.imshow([[1]])
(4)             ax.scatter(range(3), range(3), c=range(3))
(4)             with mock.patch("matplotlib.backends.qt_compat._exec", lambda obj: None):
(8)                 fig.canvas.manager.toolbar.edit_parameters()
(0)         @pytest.mark.backend('QtAgg', skip_on_importerror=True)
(0)         def test_figureoptions_with_datetime_axes():
(4)             fig, ax = plt.subplots()
(4)             xydata = [
(8)                 datetime(year=2021, month=1, day=1),
(8)                 datetime(year=2021, month=2, day=1)
(4)             ]
(4)             ax.plot(xydata, xydata)
(4)             with mock.patch("matplotlib.backends.qt_compat._exec", lambda obj: None):
(8)                 fig.canvas.manager.toolbar.edit_parameters()
(0)         @pytest.mark.backend('QtAgg', skip_on_importerror=True)
(0)         def test_double_resize():
(4)             fig, ax = plt.subplots()
(4)             fig.canvas.draw()
(4)             window = fig.canvas.manager.window
(4)             w, h = 3, 2
(4)             fig.set_size_inches(w, h)
(4)             assert fig.canvas.width() == w * matplotlib.rcParams['figure.dpi']
(4)             assert fig.canvas.height() == h * matplotlib.rcParams['figure.dpi']
(4)             old_width = window.width()
(4)             old_height = window.height()
(4)             fig.set_size_inches(w, h)
(4)             assert window.width() == old_width
(4)             assert window.height() == old_height
(0)         @pytest.mark.backend('QtAgg', skip_on_importerror=True)
(0)         def test_canvas_reinit():
(4)             from matplotlib.backends.backend_qtagg import FigureCanvasQTAgg
(4)             called = False
(4)             def crashing_callback(fig, stale):
(8)                 nonlocal called
(8)                 fig.canvas.draw_idle()
(8)                 called = True
(4)             fig, ax = plt.subplots()
(4)             fig.stale_callback = crashing_callback
(4)             canvas = FigureCanvasQTAgg(fig)
(4)             fig.stale = True
(4)             assert called
(0)         @pytest.mark.backend('Qt5Agg', skip_on_importerror=True)
(0)         def test_form_widget_get_with_datetime_and_date_fields():
(4)             from matplotlib.backends.backend_qt import _create_qApp
(4)             _create_qApp()
(4)             form = [
(8)                 ("Datetime field", datetime(year=2021, month=3, day=11)),
(8)                 ("Date field", date(year=2021, month=3, day=11))
(4)             ]
(4)             widget = _formlayout.FormWidget(form)
(4)             widget.setup()
(4)             values = widget.get()
(4)             assert values == [
(8)                 datetime(year=2021, month=3, day=11),
(8)                 date(year=2021, month=3, day=11)
```

```
215: (4)                ]
216: (0)            def _get_testable_qt_backends():
217: (4)                envs = []
218: (4)                for deps, env in [
219: (12)                       ([qt_api], {"MPLBACKEND": "qtagg", "QT_API": qt_api})
220: (12)                       for qt_api in ["PyQt6", "PySide6", "PyQt5", "PySide2"]
221: (4)                ]:
222: (8)                    reason = None
223: (8)                    missing = [dep for dep in deps if not importlib.util.find_spec(dep)]
224: (8)                    if (sys.platform == "linux" and
225: (16)                           not _c_internal_utils.display_is_valid()):
226: (12)                       reason = "$DISPLAY and $WAYLAND_DISPLAY are unset"
227: (8)                    elif missing:
228: (12)                       reason = "{} cannot be imported".format(", ".join(missing))
229: (8)                    elif env["MPLBACKEND"] == 'macosx' and os.environ.get('TF_BUILD'):
230: (12)                       reason = "macosx backend fails on Azure"
231: (8)                    marks = []
232: (8)                    if reason:
233: (12)                       marks.append(pytest.mark.skip(
234: (16)                           reason=f"Skipping {env} because {reason}"))
235: (8)                    envs.append(pytest.param(env, marks=marks, id=str(env)))
236: (4)                return envs
237: (0)            @pytest.mark.backend('QtAgg', skip_on_importerror=True)
238: (0)            def test_fig_sigint_override(qt_core):
239: (4)                from matplotlib.backends.backend_qt5 import _BackendQT5
240: (4)                plt.figure()
241: (4)                event_loop_handler = None
242: (4)                def fire_signal_and_quit():
243: (8)                    nonlocal event_loop_handler
244: (8)                    event_loop_handler = signal.getsignal(signal.SIGINT)
245: (8)                    qt_core.QCoreApplication.exit()
246: (4)                qt_core.QTimer.singleShot(0, fire_signal_and_quit)
247: (4)                original_handler = signal.getsignal(signal.SIGINT)
248: (4)                def custom_handler(signum, frame):
249: (8)                    pass
250: (4)                signal.signal(signal.SIGINT, custom_handler)
251: (4)                try:
252: (8)                    matplotlib.backends.backend_qt._BackendQT.mainloop()
253: (8)                    assert event_loop_handler != custom_handler
254: (8)                    assert signal.getsignal(signal.SIGINT) == custom_handler
255: (8)                    for custom_handler in (signal.SIG_DFL, signal.SIG_IGN):
256: (12)                       qt_core.QTimer.singleShot(0, fire_signal_and_quit)
257: (12)                       signal.signal(signal.SIGINT, custom_handler)
258: (12)                       _BackendQT5.mainloop()
259: (12)                       assert event_loop_handler == custom_handler
260: (12)                       assert signal.getsignal(signal.SIGINT) == custom_handler
261: (4)                finally:
262: (8)                    signal.signal(signal.SIGINT, original_handler)
263: (0)            @pytest.mark.backend('QtAgg', skip_on_importerror=True)
264: (0)            def test_ipython():
265: (4)                from matplotlib.testing import ipython_in_subprocess
266: (4)                ipython_in_subprocess("qt", {(8, 24): "qtagg", (8, 15): "QtAgg", (7, 0):
"Qt5Agg"})

----------------------------------------

File 161 - test_backends_interactive.py:

1: (0)              import functools
2: (0)              import importlib
3: (0)              import importlib.util
4: (0)              import inspect
5: (0)              import json
6: (0)              import os
7: (0)              import platform
8: (0)              import signal
9: (0)              import subprocess
10: (0)             import sys
11: (0)             import tempfile
```

```
12: (0)             import time
13: (0)             import urllib.request
14: (0)             from PIL import Image
15: (0)             import pytest
16: (0)             import matplotlib as mpl
17: (0)             from matplotlib import _c_internal_utils
18: (0)             from matplotlib.backend_tools import ToolToggleBase
19: (0)             from matplotlib.testing import subprocess_run_helper as _run_helper,
is_ci_environment
20: (0)             class _WaitForStringPopen(subprocess.Popen):
21: (4)                 """
22: (4)                 A Popen that passes flags that allow triggering KeyboardInterrupt.
23: (4)                 """
24: (4)                 def __init__(self, *args, **kwargs):
25: (8)                     if sys.platform == 'win32':
26: (12)                        kwargs['creationflags'] = subprocess.CREATE_NEW_CONSOLE
27: (8)                     super().__init__(
28: (12)                        *args, **kwargs,
29: (12)                        env={**os.environ, "MPLBACKEND": "Agg", "SOURCE_DATE_EPOCH": "0"},
30: (12)                        stdout=subprocess.PIPE, universal_newlines=True)
31: (4)                 def wait_for(self, terminator):
32: (8)                     """Read until the terminator is reached."""
33: (8)                     buf = ''
34: (8)                     while True:
35: (12)                        c = self.stdout.read(1)
36: (12)                        if not c:
37: (16)                            raise RuntimeError(
38: (20)                                f'Subprocess died before emitting expected
{terminator!r}')
39: (12)                        buf += c
40: (12)                        if buf.endswith(terminator):
41: (16)                            return
42: (0)             @functools.lru_cache
43: (0)             def _get_available_interactive_backends():
44: (4)                 _is_linux_and_display_invalid = (sys.platform == "linux" and
45: (37)                                                 not _c_internal_utils.display_is_valid())
46: (4)                 envs = []
47: (4)                 for deps, env in [
48: (12)                        *[([qt_api],
49: (15)                           {"MPLBACKEND": "qtagg", "QT_API": qt_api})
50: (14)                          for qt_api in ["PyQt6", "PySide6", "PyQt5", "PySide2"]],
51: (12)                        *[([qt_api, "cairocffi"],
52: (15)                           {"MPLBACKEND": "qtcairo", "QT_API": qt_api})
53: (14)                          for qt_api in ["PyQt6", "PySide6", "PyQt5", "PySide2"]],
54: (12)                        *[(["cairo", "gi"], {"MPLBACKEND": f"gtk{version}{renderer}"})
55: (14)                          for version in [3, 4] for renderer in ["agg", "cairo"]],
56: (12)                        (["tkinter"], {"MPLBACKEND": "tkagg"}),
57: (12)                        (["wx"], {"MPLBACKEND": "wx"}),
58: (12)                        (["wx"], {"MPLBACKEND": "wxagg"}),
59: (12)                        (["matplotlib.backends._macosx"], {"MPLBACKEND": "macosx"}),
60: (4)                 ]:
61: (8)                     reason = None
62: (8)                     missing = [dep for dep in deps if not importlib.util.find_spec(dep)]
63: (8)                     if _is_linux_and_display_invalid:
64: (12)                        reason = "$DISPLAY and $WAYLAND_DISPLAY are unset"
65: (8)                     elif missing:
66: (12)                        reason = "{} cannot be imported".format(", ".join(missing))
67: (8)                     elif env["MPLBACKEND"] == 'macosx' and os.environ.get('TF_BUILD'):
68: (12)                        reason = "macosx backend fails on Azure"
69: (8)                     elif env["MPLBACKEND"].startswith('gtk'):
70: (12)                        import gi  # type: ignore
71: (12)                        version = env["MPLBACKEND"][3]
72: (12)                        repo = gi.Repository.get_default()
73: (12)                        if f'{version}.0' not in repo.enumerate_versions('Gtk'):
74: (16)                            reason = "no usable GTK bindings"
75: (8)                     marks = []
76: (8)                     if reason:
77: (12)                        marks.append(pytest.mark.skip(reason=f"Skipping {env} because
{reason}"))
```

```
 78: (8)                     elif env["MPLBACKEND"].startswith('wx') and sys.platform == 'darwin':
 79: (12)                        marks.append(pytest.mark.xfail(reason='github #16849'))
 80: (8)                     elif (env['MPLBACKEND'] == 'tkagg' and
 81: (14)                          ('TF_BUILD' in os.environ or 'GITHUB_ACTION' in os.environ) and
 82: (14)                          sys.platform == 'darwin' and
 83: (14)                          sys.version_info[:2] < (3, 11)
 84: (14)                          ):
 85: (12)                        marks.append(  # https://github.com/actions/setup-
python/issues/649
 86: (16)                            pytest.mark.xfail(reason='Tk version mismatch on Azure macOS
CI'))
 87: (8)                     envs.append(({**env, 'BACKEND_DEPS': ','.join(deps)}, marks))
 88: (4)                 return envs
 89: (0)             def _get_testable_interactive_backends():
 90: (4)                 return [pytest.param({**env}, marks=[*marks],
 91: (25)                                     id='-'.join(f'{k}={v}' for k, v in env.items()))
 92: (12)                        for env, marks in _get_available_interactive_backends()]
 93: (0)             _test_timeout = 120 if is_ci_environment() else 20
 94: (0)             def _test_toolbar_button_la_mode_icon(fig):
 95: (4)                 with tempfile.TemporaryDirectory() as tempdir:
 96: (8)                     img = Image.new("LA", (26, 26))
 97: (8)                     tmp_img_path = os.path.join(tempdir, "test_la_icon.png")
 98: (8)                     img.save(tmp_img_path)
 99: (8)                     class CustomTool(ToolToggleBase):
 100: (12)                       image = tmp_img_path
 101: (12)                       description = ""  # gtk3 backend does not allow None
 102: (8)                    toolmanager = fig.canvas.manager.toolmanager
 103: (8)                    toolbar = fig.canvas.manager.toolbar
 104: (8)                    toolmanager.add_tool("test", CustomTool)
 105: (8)                    toolbar.add_tool("test", "group")
 106: (0)            def _test_interactive_impl():
 107: (4)                import importlib.util
 108: (4)                import io
 109: (4)                import json
 110: (4)                import sys
 111: (4)                import pytest
 112: (4)                import matplotlib as mpl
 113: (4)                from matplotlib import pyplot as plt
 114: (4)                from matplotlib.backend_bases import KeyEvent
 115: (4)                mpl.rcParams.update({
 116: (8)                    "webagg.open_in_browser": False,
 117: (8)                    "webagg.port_retries": 1,
 118: (4)                })
 119: (4)                mpl.rcParams.update(json.loads(sys.argv[1]))
 120: (4)                backend = plt.rcParams["backend"].lower()
 121: (4)                if backend.endswith("agg") and not backend.startswith(("gtk", "web")):
 122: (8)                    plt.figure()
 123: (8)                    if backend != "tkagg":
 124: (12)                       with pytest.raises(ImportError):
 125: (16)                           mpl.use("tkagg", force=True)
 126: (8)                    def check_alt_backend(alt_backend):
 127: (12)                       mpl.use(alt_backend, force=True)
 128: (12)                       fig = plt.figure()
 129: (12)                       assert (type(fig.canvas).__module__ ==
 130: (20)                               f"matplotlib.backends.backend_{alt_backend}")
 131: (12)                       plt.close("all")
 132: (8)                    if importlib.util.find_spec("cairocffi"):
 133: (12)                       check_alt_backend(backend[:-3] + "cairo")
 134: (8)                    check_alt_backend("svg")
 135: (4)                mpl.use(backend, force=True)
 136: (4)                fig, ax = plt.subplots()
 137: (4)                assert type(fig.canvas).__module__ ==
f"matplotlib.backends.backend_{backend}"
 138: (4)                assert fig.canvas.manager.get_window_title() == "Figure 1"
 139: (4)                if mpl.rcParams["toolbar"] == "toolmanager":
 140: (8)                    _test_toolbar_button_la_mode_icon(fig)
 141: (4)                ax.plot([0, 1], [2, 3])
 142: (4)                if fig.canvas.toolbar:  # i.e toolbar2.
 143: (8)                    fig.canvas.toolbar.draw_rubberband(None, 1., 1, 2., 2)
```

```
144: (4)                timer = fig.canvas.new_timer(1.)  # Test that floats are cast to int.
145: (4)                timer.add_callback(KeyEvent("key_press_event", fig.canvas, "q")._process)
146: (4)                fig.canvas.mpl_connect("draw_event", lambda event: timer.start())
147: (4)                fig.canvas.mpl_connect("close_event", print)
148: (4)                result = io.BytesIO()
149: (4)                fig.savefig(result, format='png')
150: (4)                plt.show()
151: (4)                plt.pause(0.5)
152: (4)                result_after = io.BytesIO()
153: (4)                fig.savefig(result_after, format='png')
154: (4)                if not backend.startswith('qt5') and sys.platform == 'darwin':
155: (8)                    assert result.getvalue() == result_after.getvalue()
156: (0)            @pytest.mark.parametrize("env", _get_testable_interactive_backends())
157: (0)            @pytest.mark.parametrize("toolbar", ["toolbar2", "toolmanager"])
158: (0)            @pytest.mark.flaky(reruns=3)
159: (0)            def test_interactive_backend(env, toolbar):
160: (4)                if env["MPLBACKEND"] == "macosx":
161: (8)                    if toolbar == "toolmanager":
162: (12)                       pytest.skip("toolmanager is not implemented for macosx.")
163: (4)                if env["MPLBACKEND"] == "wx":
164: (8)                    pytest.skip("wx backend is deprecated; tests failed on appveyor")
165: (4)                try:
166: (8)                    proc = _run_helper(
167: (12)                       _test_interactive_impl,
168: (12)                       json.dumps({"toolbar": toolbar}),
169: (12)                       timeout=_test_timeout,
170: (12)                       extra_env=env,
171: (8)                    )
172: (4)                except subprocess.CalledProcessError as err:
173: (8)                    pytest.fail(
174: (12)                       "Subprocess failed to test intended behavior\n"
175: (12)                       + str(err.stderr))
176: (4)                assert proc.stdout.count("CloseEvent") == 1
177: (0)            def _test_thread_impl():
178: (4)                from concurrent.futures import ThreadPoolExecutor
179: (4)                import matplotlib as mpl
180: (4)                from matplotlib import pyplot as plt
181: (4)                mpl.rcParams.update({
182: (8)                    "webagg.open_in_browser": False,
183: (8)                    "webagg.port_retries": 1,
184: (4)                })
185: (4)                fig, ax = plt.subplots()
186: (4)                plt.pause(0.5)
187: (4)                future = ThreadPoolExecutor().submit(ax.plot, [1, 3, 6])
188: (4)                future.result()  # Joins the thread; rethrows any exception.
189: (4)                fig.canvas.mpl_connect("close_event", print)
190: (4)                future = ThreadPoolExecutor().submit(fig.canvas.draw)
191: (4)                plt.pause(0.5)  # flush_events fails here on at least Tkagg (bpo-41176)
192: (4)                future.result()  # Joins the thread; rethrows any exception.
193: (4)                plt.close()  # backend is responsible for flushing any events here
194: (4)                if plt.rcParams["backend"].lower().startswith("wx"):
195: (8)                    fig.canvas.flush_events()
196: (0)            _thread_safe_backends = _get_testable_interactive_backends()
197: (0)            for param in _thread_safe_backends:
198: (4)                backend = param.values[0]["MPLBACKEND"]
199: (4)                if "cairo" in backend:
200: (8)                    param.marks.append(
201: (12)                       pytest.mark.xfail(raises=subprocess.CalledProcessError))
202: (4)                elif backend == "wx":
203: (8)                    param.marks.append(
204: (12)                       pytest.mark.xfail(raises=subprocess.CalledProcessError))
205: (4)                elif backend == "macosx":
206: (8)                    from packaging.version import parse
207: (8)                    mac_ver = platform.mac_ver()[0]
208: (8)                    if mac_ver and parse(mac_ver) < parse('10.16'):
209: (12)                       param.marks.append(
210: (16)                           pytest.mark.xfail(raises=subprocess.TimeoutExpired,
211: (34)                                             strict=True))
212: (4)                elif param.values[0].get("QT_API") == "PySide2":
```

```
213: (8)                    param.marks.append(
214: (12)                       pytest.mark.xfail(raises=subprocess.CalledProcessError))
215: (4)                elif backend == "tkagg" and platform.python_implementation() != 'CPython':
216: (8)                    param.marks.append(
217: (12)                       pytest.mark.xfail(
218: (16)                           reason='PyPy does not support Tkinter threading: '
219: (23)                                  'https://foss.heptapod.net/pypy/pypy/-/issues/1929',
220: (16)                           strict=True))
221: (4)                elif (backend == 'tkagg' and
222: (10)                      ('TF_BUILD' in os.environ or 'GITHUB_ACTION' in os.environ) and
223: (10)                      sys.platform == 'darwin' and sys.version_info[:2] < (3, 11)):
224: (8)                    param.marks.append(  # https://github.com/actions/setup-
python/issues/649
225: (12)                       pytest.mark.xfail('Tk version mismatch on Azure macOS CI'))
226: (0)            @pytest.mark.parametrize("env", _thread_safe_backends)
227: (0)            @pytest.mark.flaky(reruns=3)
228: (0)            def test_interactive_thread_safety(env):
229: (4)                proc = _run_helper(_test_thread_impl, timeout=_test_timeout,
extra_env=env)
230: (4)                assert proc.stdout.count("CloseEvent") == 1
231: (0)            def _impl_test_lazy_auto_backend_selection():
232: (4)                import matplotlib
233: (4)                import matplotlib.pyplot as plt
234: (4)                bk = matplotlib.rcParams._get('backend')
235: (4)                assert not isinstance(bk, str)
236: (4)                assert plt._backend_mod is None
237: (4)                plt.plot(5)
238: (4)                assert plt._backend_mod is not None
239: (4)                bk = matplotlib.rcParams._get('backend')
240: (4)                assert isinstance(bk, str)
241: (0)            def test_lazy_auto_backend_selection():
242: (4)                _run_helper(_impl_test_lazy_auto_backend_selection,
243: (16)                           timeout=_test_timeout)
244: (0)            def _implqt5agg():
245: (4)                import matplotlib.backends.backend_qt5agg  # noqa
246: (4)                import sys
247: (4)                assert 'PyQt6' not in sys.modules
248: (4)                assert 'pyside6' not in sys.modules
249: (4)                assert 'PyQt5' in sys.modules or 'pyside2' in sys.modules
250: (0)            def _implcairo():
251: (4)                import matplotlib.backends.backend_qt5cairo  # noqa
252: (4)                import sys
253: (4)                assert 'PyQt6' not in sys.modules
254: (4)                assert 'pyside6' not in sys.modules
255: (4)                assert 'PyQt5' in sys.modules or 'pyside2' in sys.modules
256: (0)            def _implcore():
257: (4)                import matplotlib.backends.backend_qt5  # noqa
258: (4)                import sys
259: (4)                assert 'PyQt6' not in sys.modules
260: (4)                assert 'pyside6' not in sys.modules
261: (4)                assert 'PyQt5' in sys.modules or 'pyside2' in sys.modules
262: (0)            def test_qt5backends_uses_qt5():
263: (4)                qt5_bindings = [
264: (8)                    dep for dep in ['PyQt5', 'pyside2']
265: (8)                    if importlib.util.find_spec(dep) is not None
266: (4)                ]
267: (4)                qt6_bindings = [
268: (8)                    dep for dep in ['PyQt6', 'pyside6']
269: (8)                    if importlib.util.find_spec(dep) is not None
270: (4)                ]
271: (4)                if len(qt5_bindings) == 0 or len(qt6_bindings) == 0:
272: (8)                    pytest.skip('need both QT6 and QT5 bindings')
273: (4)                _run_helper(_implqt5agg, timeout=_test_timeout)
274: (4)                if importlib.util.find_spec('pycairo') is not None:
275: (8)                    _run_helper(_implcairo, timeout=_test_timeout)
276: (4)                _run_helper(_implcore, timeout=_test_timeout)
277: (0)            def _impl_missing():
278: (4)                import sys
279: (4)                sys.modules["PyQt6"] = None
```

```
280: (4)                sys.modules["PyQt5"] = None
281: (4)                sys.modules["PySide2"] = None
282: (4)                sys.modules["PySide6"] = None
283: (4)                import matplotlib.pyplot as plt
284: (4)                with pytest.raises(ImportError, match="Failed to import any of the
following Qt"):
285: (8)                    plt.switch_backend("qtagg")
286: (4)                with pytest.raises(ImportError, match="^(?:(?!(PySide6|PyQt6)).)*$"):
287: (8)                    plt.switch_backend("qt5agg")
288: (0)            def test_qt_missing():
289: (4)                _run_helper(_impl_missing, timeout=_test_timeout)
290: (0)            def _impl_test_cross_Qt_imports():
291: (4)                import importlib
292: (4)                import sys
293: (4)                import warnings
294: (4)                _, host_binding, mpl_binding = sys.argv
295: (4)                importlib.import_module(f'{mpl_binding}.QtCore')
296: (4)                mpl_binding_qwidgets = importlib.import_module(f'{mpl_binding}.QtWidgets')
297: (4)                import matplotlib.backends.backend_qt
298: (4)                host_qwidgets = importlib.import_module(f'{host_binding}.QtWidgets')
299: (4)                host_app = host_qwidgets.QApplication(["mpl testing"])
300: (4)                warnings.filterwarnings("error", message=r".*Mixing Qt major.*",
301: (28)                                        category=UserWarning)
302: (4)                matplotlib.backends.backend_qt._create_qApp()
303: (0)            def qt5_and_qt6_pairs():
304: (4)                qt5_bindings = [
305: (8)                    dep for dep in ['PyQt5', 'PySide2']
306: (8)                    if importlib.util.find_spec(dep) is not None
307: (4)                ]
308: (4)                qt6_bindings = [
309: (8)                    dep for dep in ['PyQt6', 'PySide6']
310: (8)                    if importlib.util.find_spec(dep) is not None
311: (4)                ]
312: (4)                if len(qt5_bindings) == 0 or len(qt6_bindings) == 0:
313: (8)                    yield pytest.param(None, None,
314: (27)                                       marks=[pytest.mark.skip('need both QT6 and QT5
bindings')])
315: (8)                    return
316: (4)                for qt5 in qt5_bindings:
317: (8)                    for qt6 in qt6_bindings:
318: (12)                       for pair in ([qt5, qt6], [qt6, qt5]):
319: (16)                           yield pair
320: (0)            @pytest.mark.parametrize('host, mpl', [*qt5_and_qt6_pairs()])
321: (0)            def test_cross_Qt_imports(host, mpl):
322: (4)                try:
323: (8)                    proc = _run_helper(_impl_test_cross_Qt_imports, host, mpl,
324: (27)                                       timeout=_test_timeout)
325: (4)                except subprocess.CalledProcessError as ex:
326: (8)                    stderr = ex.stderr
327: (4)                else:
328: (8)                    stderr = proc.stderr
329: (4)                assert "Mixing Qt major versions may not work as expected." in stderr
330: (0)            @pytest.mark.skipif('TF_BUILD' in os.environ,
331: (20)                                reason="this test fails an azure for unknown reasons")
332: (0)            @pytest.mark.skipif(sys.platform == "win32", reason="Cannot send SIGINT on
Windows.")
333: (0)            def test_webagg():
334: (4)                pytest.importorskip("tornado")
335: (4)                proc = subprocess.Popen(
336: (8)                    [sys.executable, "-c",
337: (9)                     inspect.getsource(_test_interactive_impl)
338: (9)                     + "\n_test_interactive_impl()", "{}"],
339: (8)                    env={**os.environ, "MPLBACKEND": "webagg", "SOURCE_DATE_EPOCH": "0"})
340: (4)                url = f'http://{mpl.rcParams["webagg.address"]}:
{mpl.rcParams["webagg.port"]}'
341: (4)                timeout = time.perf_counter() + _test_timeout
342: (4)                try:
343: (8)                    while True:
344: (12)                       try:
```

```
(16)                            retcode = proc.poll()
(16)                            assert retcode is None
(16)                            conn = urllib.request.urlopen(url)
(16)                            break
(12)                        except urllib.error.URLError:
(16)                            if time.perf_counter() > timeout:
(20)                                pytest.fail("Failed to connect to the webagg server.")
(16)                            else:
(20)                                continue
(8)                     conn.close()
(8)                     proc.send_signal(signal.SIGINT)
(8)                     assert proc.wait(timeout=_test_timeout) == 0
(4)                 finally:
(8)                     if proc.poll() is None:
(12)                        proc.kill()
(0)             def _lazy_headless():
(4)                 import os
(4)                 import sys
(4)                 backend, deps = sys.argv[1:]
(4)                 deps = deps.split(',')
(4)                 os.environ.pop('DISPLAY', None)
(4)                 os.environ.pop('WAYLAND_DISPLAY', None)
(4)                 for dep in deps:
(8)                     assert dep not in sys.modules
(4)                 import matplotlib.pyplot as plt
(4)                 assert plt.get_backend() == 'agg'
(4)                 for dep in deps:
(8)                     assert dep not in sys.modules
(4)                 for dep in deps:
(8)                     importlib.import_module(dep)
(8)                     assert dep in sys.modules
(4)                 try:
(8)                     plt.switch_backend(backend)
(4)                 except ImportError:
(8)                     pass
(4)                 else:
(8)                     sys.exit(1)
(0)             @pytest.mark.skipif(sys.platform != "linux", reason="this a linux-only test")
(0)             @pytest.mark.parametrize("env", _get_testable_interactive_backends())
(0)             def test_lazy_linux_headless(env):
(4)                 proc = _run_helper(
(8)                     _lazy_headless,
(8)                     env.pop('MPLBACKEND'), env.pop("BACKEND_DEPS"),
(8)                     timeout=_test_timeout,
(8)                     extra_env={**env, 'DISPLAY': '', 'WAYLAND_DISPLAY': ''}
(4)                 )
(0)             def _test_number_of_draws_script():
(4)                 import matplotlib.pyplot as plt
(4)                 fig, ax = plt.subplots()
(4)                 ln, = ax.plot([0, 1], [1, 2], animated=True)
(4)                 plt.show(block=False)
(4)                 plt.pause(0.3)
(4)                 fig.canvas.mpl_connect('draw_event', print)
(4)                 bg = fig.canvas.copy_from_bbox(fig.bbox)
(4)                 ax.draw_artist(ln)
(4)                 fig.canvas.blit(fig.bbox)
(4)                 for j in range(10):
(8)                     fig.canvas.restore_region(bg)
(8)                     ln, = ax.plot([0, 1], [1, 2])
(8)                     ax.draw_artist(ln)
(8)                     fig.canvas.blit(fig.bbox)
(8)                     fig.canvas.flush_events()
(4)                 plt.pause(0.1)
(0)             _blit_backends = _get_testable_interactive_backends()
(0)             for param in _blit_backends:
(4)                 backend = param.values[0]["MPLBACKEND"]
(4)                 if backend == "gtk3cairo":
(8)                     param.marks.append(
(12)                        pytest.mark.skip("gtk3cairo does not support blitting"))
```

```
414: (4)                elif backend == "gtk4cairo":
415: (8)                    param.marks.append(
416: (12)                       pytest.mark.skip("gtk4cairo does not support blitting"))
417: (4)                elif backend == "wx":
418: (8)                    param.marks.append(
419: (12)                       pytest.mark.skip("wx does not support blitting"))
420: (4)                elif (backend == 'tkagg' and
421: (10)                     ('TF_BUILD' in os.environ or 'GITHUB_ACTION' in os.environ) and
422: (10)                     sys.platform == 'darwin' and
423: (10)                     sys.version_info[:2] < (3, 11)
424: (10)                     ):
425: (8)                    param.marks.append(  # https://github.com/actions/setup-
python/issues/649
426: (12)                       pytest.mark.xfail('Tk version mismatch on Azure macOS CI')
427: (8)                    )
428: (0)            @pytest.mark.parametrize("env", _blit_backends)
429: (0)            @pytest.mark.flaky(reruns=4)
430: (0)            def test_blitting_events(env):
431: (4)                proc = _run_helper(
432: (8)                    _test_number_of_draws_script, timeout=_test_timeout, extra_env=env)
433: (4)                ndraws = proc.stdout.count("DrawEvent")
434: (4)                assert 0 < ndraws < 5
435: (0)            def _impl_test_interactive_timers():
436: (4)                import os
437: (4)                from unittest.mock import Mock
438: (4)                import matplotlib.pyplot as plt
439: (4)                pause_time = 2 if os.getenv("CI") else 0.5
440: (4)                fig = plt.figure()
441: (4)                plt.pause(pause_time)
442: (4)                timer = fig.canvas.new_timer(0.1)
443: (4)                mock = Mock()
444: (4)                timer.add_callback(mock)
445: (4)                timer.start()
446: (4)                plt.pause(pause_time)
447: (4)                timer.stop()
448: (4)                assert mock.call_count > 1
449: (4)                mock.call_count = 0
450: (4)                timer.single_shot = True
451: (4)                timer.start()
452: (4)                plt.pause(pause_time)
453: (4)                assert mock.call_count == 1
454: (4)                timer.start()
455: (4)                plt.pause(pause_time)
456: (4)                assert mock.call_count == 2
457: (4)                plt.close("all")
458: (0)            @pytest.mark.parametrize("env", _get_testable_interactive_backends())
459: (0)            def test_interactive_timers(env):
460: (4)                if env["MPLBACKEND"] == "gtk3cairo" and os.getenv("CI"):
461: (8)                    pytest.skip("gtk3cairo timers do not work in remote CI")
462: (4)                if env["MPLBACKEND"] == "wx":
463: (8)                    pytest.skip("wx backend is deprecated; tests failed on appveyor")
464: (4)                _run_helper(_impl_test_interactive_timers,
465: (16)                           timeout=_test_timeout, extra_env=env)
466: (0)            def _test_sigint_impl(backend, target_name, kwargs):
467: (4)                import sys
468: (4)                import matplotlib.pyplot as plt
469: (4)                import os
470: (4)                import threading
471: (4)                plt.switch_backend(backend)
472: (4)                def interrupter():
473: (8)                    if sys.platform == 'win32':
474: (12)                       import win32api
475: (12)                       win32api.GenerateConsoleCtrlEvent(0, 0)
476: (8)                    else:
477: (12)                       import signal
478: (12)                       os.kill(os.getpid(), signal.SIGINT)
479: (4)                target = getattr(plt, target_name)
480: (4)                timer = threading.Timer(1, interrupter)
481: (4)                fig = plt.figure()
```

```
482: (4)                fig.canvas.mpl_connect(
483: (8)                    'draw_event',
484: (8)                    lambda *args: print('DRAW', flush=True)
485: (4)                )
486: (4)                fig.canvas.mpl_connect(
487: (8)                    'draw_event',
488: (8)                    lambda *args: timer.start()
489: (4)                )
490: (4)                try:
491: (8)                    target(**kwargs)
492: (4)                except KeyboardInterrupt:
493: (8)                    print('SUCCESS', flush=True)
494: (0)            @pytest.mark.parametrize("env", _get_testable_interactive_backends())
495: (0)            @pytest.mark.parametrize("target, kwargs", [
496: (4)                ('show', {'block': True}),
497: (4)                ('pause', {'interval': 10})
498: (0)            ])
499: (0)            def test_sigint(env, target, kwargs):
500: (4)                backend = env.get("MPLBACKEND")
501: (4)                if not backend.startswith(("qt", "macosx")):
502: (8)                    pytest.skip("SIGINT currently only tested on qt and macosx")
503: (4)                proc = _WaitForStringPopen(
504: (8)                    [sys.executable, "-c",
505: (9)                     inspect.getsource(_test_sigint_impl) +
506: (9)                     f"\n_test_sigint_impl({backend!r}, {target!r}, {kwargs!r})"])
507: (4)                try:
508: (8)                    proc.wait_for('DRAW')
509: (8)                    stdout, _ = proc.communicate(timeout=_test_timeout)
510: (4)                except Exception:
511: (8)                    proc.kill()
512: (8)                    stdout, _ = proc.communicate()
513: (8)                    raise
514: (4)                assert 'SUCCESS' in stdout
515: (0)            def _test_other_signal_before_sigint_impl(backend, target_name, kwargs):
516: (4)                import signal
517: (4)                import matplotlib.pyplot as plt
518: (4)                plt.switch_backend(backend)
519: (4)                target = getattr(plt, target_name)
520: (4)                fig = plt.figure()
521: (4)                fig.canvas.mpl_connect('draw_event', lambda *args: print('DRAW',
flush=True))
522: (4)                timer = fig.canvas.new_timer(interval=1)
523: (4)                timer.single_shot = True
524: (4)                timer.add_callback(print, 'SIGUSR1', flush=True)
525: (4)                def custom_signal_handler(signum, frame):
526: (8)                    timer.start()
527: (4)                signal.signal(signal.SIGUSR1, custom_signal_handler)
528: (4)                try:
529: (8)                    target(**kwargs)
530: (4)                except KeyboardInterrupt:
531: (8)                    print('SUCCESS', flush=True)
532: (0)            @pytest.mark.skipif(sys.platform == 'win32',
533: (20)                               reason='No other signal available to send on Windows')
534: (0)            @pytest.mark.parametrize("env", _get_testable_interactive_backends())
535: (0)            @pytest.mark.parametrize("target, kwargs", [
536: (4)                ('show', {'block': True}),
537: (4)                ('pause', {'interval': 10})
538: (0)            ])
539: (0)            def test_other_signal_before_sigint(env, target, kwargs, request):
540: (4)                backend = env.get("MPLBACKEND")
541: (4)                if not backend.startswith(("qt", "macosx")):
542: (8)                    pytest.skip("SIGINT currently only tested on qt and macosx")
543: (4)                if backend == "macosx":
544: (8)                    request.node.add_marker(pytest.mark.xfail(reason="macosx backend is
buggy"))
545: (4)                if sys.platform == "darwin" and target == "show":
546: (8)                    request.node.add_marker(
547: (12)                       pytest.mark.xfail(reason="Qt backend is buggy on macOS"))
548: (4)                proc = _WaitForStringPopen(
```

```
549: (8)                    [sys.executable, "-c",
550: (9)                     inspect.getsource(_test_other_signal_before_sigint_impl) +
551: (9)                     "\n_test_other_signal_before_sigint_impl("
552: (12)                       f"{backend!r}, {target!r}, {kwargs!r})"])
553: (4)                try:
554: (8)                    proc.wait_for('DRAW')
555: (8)                    os.kill(proc.pid, signal.SIGUSR1)
556: (8)                    proc.wait_for('SIGUSR1')
557: (8)                    os.kill(proc.pid, signal.SIGINT)
558: (8)                    stdout, _ = proc.communicate(timeout=_test_timeout)
559: (4)                except Exception:
560: (8)                    proc.kill()
561: (8)                    stdout, _ = proc.communicate()
562: (8)                    raise
563: (4)                print(stdout)
564: (4)                assert 'SUCCESS' in stdout

----------------------------------------

File 162 - test_backend_svg.py:

1: (0)              import datetime
2: (0)              from io import BytesIO
3: (0)              from pathlib import Path
4: (0)              import xml.etree.ElementTree
5: (0)              import xml.parsers.expat
6: (0)              import pytest
7: (0)              import numpy as np
8: (0)              import matplotlib as mpl
9: (0)              from matplotlib.figure import Figure
10: (0)             from matplotlib.text import Text
11: (0)             import matplotlib.pyplot as plt
12: (0)             from matplotlib.testing.decorators import check_figures_equal,
image_comparison
13: (0)             from matplotlib.testing._markers import needs_usetex
14: (0)             from matplotlib import font_manager as fm
15: (0)             from matplotlib.offsetbox import (OffsetImage, AnnotationBbox)
16: (0)             def test_visibility():
17: (4)                 fig, ax = plt.subplots()
18: (4)                 x = np.linspace(0, 4 * np.pi, 50)
19: (4)                 y = np.sin(x)
20: (4)                 yerr = np.ones_like(y)
21: (4)                 a, b, c = ax.errorbar(x, y, yerr=yerr, fmt='ko')
22: (4)                 for artist in b:
23: (8)                     artist.set_visible(False)
24: (4)                 with BytesIO() as fd:
25: (8)                     fig.savefig(fd, format='svg')
26: (8)                     buf = fd.getvalue()
27: (4)                 parser = xml.parsers.expat.ParserCreate()
28: (4)                 parser.Parse(buf)  # this will raise ExpatError if the svg is invalid
29: (0)             @image_comparison(['fill_black_with_alpha.svg'], remove_text=True)
30: (0)             def test_fill_black_with_alpha():
31: (4)                 fig, ax = plt.subplots()
32: (4)                 ax.scatter(x=[0, 0.1, 1], y=[0, 0, 0], c='k', alpha=0.1, s=10000)
33: (0)             @image_comparison(['noscale'], remove_text=True)
34: (0)             def test_noscale():
35: (4)                 X, Y = np.meshgrid(np.arange(-5, 5, 1), np.arange(-5, 5, 1))
36: (4)                 Z = np.sin(Y ** 2)
37: (4)                 fig, ax = plt.subplots()
38: (4)                 ax.imshow(Z, cmap='gray', interpolation='none')
39: (0)             def test_text_urls():
40: (4)                 fig = plt.figure()
41: (4)                 test_url = "http://test_text_urls.matplotlib.org"
42: (4)                 fig.suptitle("test_text_urls", url=test_url)
43: (4)                 with BytesIO() as fd:
44: (8)                     fig.savefig(fd, format='svg')
45: (8)                     buf = fd.getvalue().decode()
46: (4)                 expected = f'<a xlink:href="{test_url}">'
47: (4)                 assert expected in buf
```

```
48: (0)             @image_comparison(['bold_font_output.svg'])
49: (0)             def test_bold_font_output():
50: (4)                 fig, ax = plt.subplots()
51: (4)                 ax.plot(np.arange(10), np.arange(10))
52: (4)                 ax.set_xlabel('nonbold-xlabel')
53: (4)                 ax.set_ylabel('bold-ylabel', fontweight='bold')
54: (4)                 ax.set_title('bold-title', fontweight='bold')
55: (0)             @image_comparison(['bold_font_output_with_none_fonttype.svg'])
56: (0)             def test_bold_font_output_with_none_fonttype():
57: (4)                 plt.rcParams['svg.fonttype'] = 'none'
58: (4)                 fig, ax = plt.subplots()
59: (4)                 ax.plot(np.arange(10), np.arange(10))
60: (4)                 ax.set_xlabel('nonbold-xlabel')
61: (4)                 ax.set_ylabel('bold-ylabel', fontweight='bold')
62: (4)                 ax.set_title('bold-title', fontweight='bold')
63: (0)             @check_figures_equal(tol=20)
64: (0)             def test_rasterized(fig_test, fig_ref):
65: (4)                 t = np.arange(0, 100) * (2.3)
66: (4)                 x = np.cos(t)
67: (4)                 y = np.sin(t)
68: (4)                 ax_ref = fig_ref.subplots()
69: (4)                 ax_ref.plot(x, y, "-", c="r", lw=10)
70: (4)                 ax_ref.plot(x+1, y, "-", c="b", lw=10)
71: (4)                 ax_test = fig_test.subplots()
72: (4)                 ax_test.plot(x, y, "-", c="r", lw=10, rasterized=True)
73: (4)                 ax_test.plot(x+1, y, "-", c="b", lw=10, rasterized=True)
74: (0)             @check_figures_equal()
75: (0)             def test_rasterized_ordering(fig_test, fig_ref):
76: (4)                 t = np.arange(0, 100) * (2.3)
77: (4)                 x = np.cos(t)
78: (4)                 y = np.sin(t)
79: (4)                 ax_ref = fig_ref.subplots()
80: (4)                 ax_ref.set_xlim(0, 3)
81: (4)                 ax_ref.set_ylim(-1.1, 1.1)
82: (4)                 ax_ref.plot(x, y, "-", c="r", lw=10, rasterized=True)
83: (4)                 ax_ref.plot(x+1, y, "-", c="b", lw=10, rasterized=False)
84: (4)                 ax_ref.plot(x+2, y, "-", c="g", lw=10, rasterized=True)
85: (4)                 ax_ref.plot(x+3, y, "-", c="m", lw=10, rasterized=True)
86: (4)                 ax_test = fig_test.subplots()
87: (4)                 ax_test.set_xlim(0, 3)
88: (4)                 ax_test.set_ylim(-1.1, 1.1)
89: (4)                 ax_test.plot(x, y, "-", c="r", lw=10, rasterized=True, zorder=1.1)
90: (4)                 ax_test.plot(x+2, y, "-", c="g", lw=10, rasterized=True, zorder=1.3)
91: (4)                 ax_test.plot(x+3, y, "-", c="m", lw=10, rasterized=True, zorder=1.4)
92: (4)                 ax_test.plot(x+1, y, "-", c="b", lw=10, rasterized=False, zorder=1.2)
93: (0)             @check_figures_equal(tol=5, extensions=['svg', 'pdf'])
94: (0)             def test_prevent_rasterization(fig_test, fig_ref):
95: (4)                 loc = [0.05, 0.05]
96: (4)                 ax_ref = fig_ref.subplots()
97: (4)                 ax_ref.plot([loc[0]], [loc[1]], marker="x", c="black", zorder=2)
98: (4)                 b = mpl.offsetbox.TextArea("X")
99: (4)                 abox = mpl.offsetbox.AnnotationBbox(b, loc, zorder=2.1)
100: (4)                ax_ref.add_artist(abox)
101: (4)                ax_test = fig_test.subplots()
102: (4)                ax_test.plot([loc[0]], [loc[1]], marker="x", c="black", zorder=2,
103: (17)                            rasterized=True)
104: (4)                b = mpl.offsetbox.TextArea("X")
105: (4)                abox = mpl.offsetbox.AnnotationBbox(b, loc, zorder=2.1)
106: (4)                ax_test.add_artist(abox)
107: (0)            def test_count_bitmaps():
108: (4)                def count_tag(fig, tag):
109: (8)                    with BytesIO() as fd:
110: (12)                       fig.savefig(fd, format='svg')
111: (12)                       buf = fd.getvalue().decode()
112: (8)                    return buf.count(f"<{tag}")
113: (4)                fig1 = plt.figure()
114: (4)                ax1 = fig1.add_subplot(1, 1, 1)
115: (4)                ax1.set_axis_off()
116: (4)                for n in range(5):
```

```
117: (8)                ax1.plot([0, 20], [0, n], "b-", rasterized=False)
118: (4)            assert count_tag(fig1, "image") == 0
119: (4)            assert count_tag(fig1, "path") == 6  # axis patch plus lines
120: (4)            fig2 = plt.figure()
121: (4)            ax2 = fig2.add_subplot(1, 1, 1)
122: (4)            ax2.set_axis_off()
123: (4)            for n in range(5):
124: (8)                ax2.plot([0, 20], [0, n], "b-", rasterized=True)
125: (4)            assert count_tag(fig2, "image") == 1
126: (4)            assert count_tag(fig2, "path") == 1  # axis patch
127: (4)            fig3 = plt.figure()
128: (4)            ax3 = fig3.add_subplot(1, 1, 1)
129: (4)            ax3.set_axis_off()
130: (4)            for n in range(5):
131: (8)                ax3.plot([0, 20], [n, 0], "b-", rasterized=False)
132: (8)                ax3.plot([0, 20], [0, n], "b-", rasterized=True)
133: (4)            assert count_tag(fig3, "image") == 5
134: (4)            assert count_tag(fig3, "path") == 6
135: (4)            fig4 = plt.figure()
136: (4)            ax4 = fig4.add_subplot(1, 1, 1)
137: (4)            ax4.set_axis_off()
138: (4)            ax4.set_rasterized(True)
139: (4)            for n in range(5):
140: (8)                ax4.plot([0, 20], [n, 0], "b-", rasterized=False)
141: (8)                ax4.plot([0, 20], [0, n], "b-", rasterized=True)
142: (4)            assert count_tag(fig4, "image") == 1
143: (4)            assert count_tag(fig4, "path") == 1
144: (4)            fig5 = plt.figure()
145: (4)            fig5.suppressComposite = True
146: (4)            ax5 = fig5.add_subplot(1, 1, 1)
147: (4)            ax5.set_axis_off()
148: (4)            for n in range(5):
149: (8)                ax5.plot([0, 20], [0, n], "b-", rasterized=True)
150: (4)            assert count_tag(fig5, "image") == 5
151: (4)            assert count_tag(fig5, "path") == 1  # axis patch
152: (0)        @mpl.style.context('default')
153: (0)        @needs_usetex
154: (0)        def test_unicode_won():
155: (4)            fig = Figure()
156: (4)            fig.text(.5, .5, r'\textwon', usetex=True)
157: (4)            with BytesIO() as fd:
158: (8)                fig.savefig(fd, format='svg')
159: (8)                buf = fd.getvalue()
160: (4)            tree = xml.etree.ElementTree.fromstring(buf)
161: (4)            ns = 'http://www.w3.org/2000/svg'
162: (4)            won_id = 'SFSS3583-8e'
163: (4)            assert len(tree.findall(f'.//{{{ns}}}path[@d][@id="{won_id}"]')) == 1
164: (4)            assert f'#{won_id}' in tree.find(f'.//{{{ns}}}use').attrib.values()
165: (0)        def test_svgnone_with_data_coordinates():
166: (4)            plt.rcParams.update({'svg.fonttype': 'none', 'font.stretch': 'condensed'})
167: (4)            expected = 'Unlikely to appear by chance'
168: (4)            fig, ax = plt.subplots()
169: (4)            ax.text(np.datetime64('2019-06-30'), 1, expected)
170: (4)            ax.set_xlim(np.datetime64('2019-01-01'), np.datetime64('2019-12-31'))
171: (4)            ax.set_ylim(0, 2)
172: (4)            with BytesIO() as fd:
173: (8)                fig.savefig(fd, format='svg')
174: (8)                fd.seek(0)
175: (8)                buf = fd.read().decode()
176: (4)            assert expected in buf and "condensed" in buf
177: (0)        def test_gid():
178: (4)            """Test that object gid appears in output svg."""
179: (4)            from matplotlib.offsetbox import OffsetBox
180: (4)            from matplotlib.axis import Tick
181: (4)            fig = plt.figure()
182: (4)            ax1 = fig.add_subplot(131)
183: (4)            ax1.imshow([[1., 2.], [2., 3.]], aspect="auto")
184: (4)            ax1.scatter([1, 2, 3], [1, 2, 3], label="myscatter")
185: (4)            ax1.plot([2, 3, 1], label="myplot")
```

```
(4)                 ax1.legend()
(4)                 ax1a = ax1.twinx()
(4)                 ax1a.bar([1, 2, 3], [1, 2, 3])
(4)                 ax2 = fig.add_subplot(132, projection="polar")
(4)                 ax2.plot([0, 1.5, 3], [1, 2, 3])
(4)                 ax3 = fig.add_subplot(133, projection="3d")
(4)                 ax3.plot([1, 2], [1, 2], [1, 2])
(4)                 fig.canvas.draw()
(4)                 gdic = {}
(4)                 for idx, obj in enumerate(fig.findobj(include_self=True)):
(8)                     if obj.get_visible():
(12)                        gid = f"test123{obj.__class__.__name__}_{idx}"
(12)                        gdic[gid] = obj
(12)                        obj.set_gid(gid)
(4)                 with BytesIO() as fd:
(8)                     fig.savefig(fd, format='svg')
(8)                     buf = fd.getvalue().decode()
(4)                 def include(gid, obj):
(8)                     if isinstance(obj, OffsetBox):
(12)                        return False
(8)                     if isinstance(obj, Text):
(12)                        if obj.get_text() == "":
(16)                            return False
(12)                        elif obj.axes is None:
(16)                            return False
(8)                     if isinstance(obj, plt.Line2D):
(12)                        xdata, ydata = obj.get_data()
(12)                        if len(xdata) == len(ydata) == 1:
(16)                            return False
(12)                        elif not hasattr(obj, "axes") or obj.axes is None:
(16)                            return False
(8)                     if isinstance(obj, Tick):
(12)                        loc = obj.get_loc()
(12)                        if loc == 0:
(16)                            return False
(12)                        vi = obj.get_view_interval()
(12)                        if loc < min(vi) or loc > max(vi):
(16)                            return False
(8)                     return True
(4)                 for gid, obj in gdic.items():
(8)                     if include(gid, obj):
(12)                        assert gid in buf
(0)             def test_savefig_tight():
(4)                 plt.savefig(BytesIO(), format="svgz", bbox_inches="tight")
(0)             def test_url():
(4)                 fig, ax = plt.subplots()
(4)                 s = ax.scatter([1, 2, 3], [4, 5, 6])
(4)                 s.set_urls(['https://example.com/foo', 'https://example.com/bar', None])
(4)                 p, = plt.plot([1, 3], [6, 5])
(4)                 p.set_url('https://example.com/baz')
(4)                 b = BytesIO()
(4)                 fig.savefig(b, format='svg')
(4)                 b = b.getvalue()
(4)                 for v in [b'foo', b'bar', b'baz']:
(8)                     assert b'https://example.com/' + v in b
(0)             def test_url_tick(monkeypatch):
(4)                 monkeypatch.setenv('SOURCE_DATE_EPOCH', '19680801')
(4)                 fig1, ax = plt.subplots()
(4)                 ax.scatter([1, 2, 3], [4, 5, 6])
(4)                 for i, tick in enumerate(ax.yaxis.get_major_ticks()):
(8)                     tick.set_url(f'https://example.com/{i}')
(4)                 fig2, ax = plt.subplots()
(4)                 ax.scatter([1, 2, 3], [4, 5, 6])
(4)                 for i, tick in enumerate(ax.yaxis.get_major_ticks()):
(8)                     tick.label1.set_url(f'https://example.com/{i}')
(8)                     tick.label2.set_url(f'https://example.com/{i}')
(4)                 b1 = BytesIO()
(4)                 fig1.savefig(b1, format='svg')
(4)                 b1 = b1.getvalue()
```

```
255: (4)                b2 = BytesIO()
256: (4)                fig2.savefig(b2, format='svg')
257: (4)                b2 = b2.getvalue()
258: (4)                for i in range(len(ax.yaxis.get_major_ticks())):
259: (8)                    assert f'https://example.com/{i}'.encode('ascii') in b1
260: (4)                assert b1 == b2
261: (0)            def test_svg_default_metadata(monkeypatch):
262: (4)                monkeypatch.setenv('SOURCE_DATE_EPOCH', '19680801')
263: (4)                fig, ax = plt.subplots()
264: (4)                with BytesIO() as fd:
265: (8)                    fig.savefig(fd, format='svg')
266: (8)                    buf = fd.getvalue().decode()
267: (4)                assert mpl.__version__ in buf
268: (4)                assert '1970-08-16' in buf
269: (4)                assert 'image/svg+xml' in buf
270: (4)                assert 'StillImage' in buf
271: (4)                with BytesIO() as fd:
272: (8)                    fig.savefig(fd, format='svg', metadata={'Date': None, 'Creator': None,
273: (48)                                                            'Format': None, 'Type': None})
274: (8)                    buf = fd.getvalue().decode()
275: (4)                assert mpl.__version__ not in buf
276: (4)                assert '1970-08-16' not in buf
277: (4)                assert 'image/svg+xml' not in buf
278: (4)                assert 'StillImage' not in buf
279: (0)            def test_svg_clear_default_metadata(monkeypatch):
280: (4)                monkeypatch.setenv('SOURCE_DATE_EPOCH', '19680801')
281: (4)                metadata_contains = {'creator': mpl.__version__, 'date': '1970-08-16',
282: (25)                                     'format': 'image/svg+xml', 'type': 'StillImage'}
283: (4)                SVGNS = '{http://www.w3.org/2000/svg}'
284: (4)                RDFNS = '{http://www.w3.org/1999/02/22-rdf-syntax-ns#}'
285: (4)                CCNS = '{http://creativecommons.org/ns#}'
286: (4)                DCNS = '{http://purl.org/dc/elements/1.1/}'
287: (4)                fig, ax = plt.subplots()
288: (4)                for name in metadata_contains:
289: (8)                    with BytesIO() as fd:
290: (12)                       fig.savefig(fd, format='svg', metadata={name.title(): None})
291: (12)                       buf = fd.getvalue().decode()
292: (8)                    root = xml.etree.ElementTree.fromstring(buf)
293: (8)                    work, = root.findall(f'./{SVGNS}metadata/{RDFNS}RDF/{CCNS}Work')
294: (8)                    for key in metadata_contains:
295: (12)                       data = work.findall(f'./{DCNS}{key}')
296: (12)                       if key == name:
297: (16)                           assert not data
298: (16)                           continue
299: (12)                       data, = data
300: (12)                       xmlstr = xml.etree.ElementTree.tostring(data, encoding="unicode")
301: (12)                       assert metadata_contains[key] in xmlstr
302: (0)            def test_svg_clear_all_metadata():
303: (4)                fig, ax = plt.subplots()
304: (4)                with BytesIO() as fd:
305: (8)                    fig.savefig(fd, format='svg', metadata={'Date': None, 'Creator': None,
306: (48)                                                            'Format': None, 'Type': None})
307: (8)                    buf = fd.getvalue().decode()
308: (4)                SVGNS = '{http://www.w3.org/2000/svg}'
309: (4)                root = xml.etree.ElementTree.fromstring(buf)
310: (4)                assert not root.findall(f'./{SVGNS}metadata')
311: (0)            def test_svg_metadata():
312: (4)                single_value = ['Coverage', 'Identifier', 'Language', 'Relation',
'Source',
313: (20)                               'Title', 'Type']
314: (4)                multi_value = ['Contributor', 'Creator', 'Keywords', 'Publisher',
'Rights']
315: (4)                metadata = {
316: (8)                    'Date': [datetime.date(1968, 8, 1),
317: (17)                            datetime.datetime(1968, 8, 2, 1, 2, 3)],
318: (8)                    'Description': 'description\ntext',
319: (8)                    **{k: f'{k} foo' for k in single_value},
320: (8)                    **{k: [f'{k} bar', f'{k} baz'] for k in multi_value},
321: (4)                }
```

```
322: (4)                fig = plt.figure()
323: (4)                with BytesIO() as fd:
324: (8)                    fig.savefig(fd, format='svg', metadata=metadata)
325: (8)                    buf = fd.getvalue().decode()
326: (4)                SVGNS = '{http://www.w3.org/2000/svg}'
327: (4)                RDFNS = '{http://www.w3.org/1999/02/22-rdf-syntax-ns#}'
328: (4)                CCNS = '{http://creativecommons.org/ns#}'
329: (4)                DCNS = '{http://purl.org/dc/elements/1.1/}'
330: (4)                root = xml.etree.ElementTree.fromstring(buf)
331: (4)                rdf, = root.findall(f'./{SVGNS}metadata/{RDFNS}RDF')
332: (4)                titles = [node.text for node in root.findall(f'./{SVGNS}title')]
333: (4)                assert titles == [metadata['Title']]
334: (4)                types = [node.attrib[f'{RDFNS}resource']
335: (13)                         for node in rdf.findall(f'./{CCNS}Work/{DCNS}type')]
336: (4)                assert types == [metadata['Type']]
337: (4)                for k in ['Description', *single_value]:
338: (8)                    if k == 'Type':
339: (12)                       continue
340: (8)                    values = [node.text
341: (18)                             for node in rdf.findall(f'./{CCNS}Work/{DCNS}{k.lower()}')]
342: (8)                    assert values == [metadata[k]]
343: (4)                for k in multi_value:
344: (8)                    if k == 'Keywords':
345: (12)                       continue
346: (8)                    values = [
347: (12)                       node.text
348: (12)                       for node in rdf.findall(
349: (16)                           f'./{CCNS}Work/{DCNS}{k.lower()}/{CCNS}Agent/{DCNS}title')]
350: (8)                    assert values == metadata[k]
351: (4)                dates = [node.text for node in rdf.findall(f'./{CCNS}Work/{DCNS}date')]
352: (4)                assert dates == ['1968-08-01/1968-08-02T01:02:03']
353: (4)                values = [node.text for node in
354: (14)                         rdf.findall(f'./{CCNS}Work/{DCNS}subject/{RDFNS}Bag/{RDFNS}li')]
355: (4)                assert values == metadata['Keywords']
356: (0)            @image_comparison(["multi_font_aspath.svg"], tol=1.8)
357: (0)            def test_multi_font_type3():
358: (4)                fp = fm.FontProperties(family=["WenQuanYi Zen Hei"])
359: (4)                if Path(fm.findfont(fp)).name != "wqy-zenhei.ttc":
360: (8)                    pytest.skip("Font may be missing")
361: (4)                plt.rc('font', family=['DejaVu Sans', 'WenQuanYi Zen Hei'], size=27)
362: (4)                plt.rc('svg', fonttype='path')
363: (4)                fig = plt.figure()
364: (4)                fig.text(0.15, 0.475, "There are 几个汉字 in between!")
365: (0)            @image_comparison(["multi_font_astext.svg"])
366: (0)            def test_multi_font_type42():
367: (4)                fp = fm.FontProperties(family=["WenQuanYi Zen Hei"])
368: (4)                if Path(fm.findfont(fp)).name != "wqy-zenhei.ttc":
369: (8)                    pytest.skip("Font may be missing")
370: (4)                fig = plt.figure()
371: (4)                plt.rc('svg', fonttype='none')
372: (4)                plt.rc('font', family=['DejaVu Sans', 'WenQuanYi Zen Hei'], size=27)
373: (4)                fig.text(0.15, 0.475, "There are 几个汉字 in between!")
374: (0)            @pytest.mark.parametrize('metadata,error,message', [
375: (4)                ({'Date': 1}, TypeError, "Invalid type for Date metadata. Expected str"),
376: (4)                ({'Date': [1]}, TypeError,
377: (5)                 "Invalid type for Date metadata. Expected iterable"),
378: (4)                ({'Keywords': 1}, TypeError,
379: (5)                 "Invalid type for Keywords metadata. Expected str"),
380: (4)                ({'Keywords': [1]}, TypeError,
381: (5)                 "Invalid type for Keywords metadata. Expected iterable"),
382: (4)                ({'Creator': 1}, TypeError,
383: (5)                 "Invalid type for Creator metadata. Expected str"),
384: (4)                ({'Creator': [1]}, TypeError,
385: (5)                 "Invalid type for Creator metadata. Expected iterable"),
386: (4)                ({'Title': 1}, TypeError,
387: (5)                 "Invalid type for Title metadata. Expected str"),
388: (4)                ({'Format': 1}, TypeError,
389: (5)                 "Invalid type for Format metadata. Expected str"),
390: (4)                ({'Foo': 'Bar'}, ValueError, "Unknown metadata key"),
```

```
(4)                 ])
(0)             def test_svg_incorrect_metadata(metadata, error, message):
(4)                 with pytest.raises(error, match=message), BytesIO() as fd:
(8)                     fig = plt.figure()
(8)                     fig.savefig(fd, format='svg', metadata=metadata)
(0)             def test_svg_escape():
(4)                 fig = plt.figure()
(4)                 fig.text(0.5, 0.5, "<\'\"&>", gid="<\'\"&>")
(4)                 with BytesIO() as fd:
(8)                     fig.savefig(fd, format='svg')
(8)                     buf = fd.getvalue().decode()
(8)                     assert '&lt;&apos;&quot;&amp;&gt;"' in buf
(0)             @pytest.mark.parametrize("font_str", [
(4)                 "'DejaVu Sans', 'WenQuanYi Zen Hei', 'Arial', sans-serif",
(4)                 "'DejaVu Serif', 'WenQuanYi Zen Hei', 'Times New Roman', serif",
(4)                 "'Arial', 'WenQuanYi Zen Hei', cursive",
(4)                 "'Impact', 'WenQuanYi Zen Hei', fantasy",
(4)                 "'DejaVu Sans Mono', 'WenQuanYi Zen Hei', 'Courier New', monospace",
(0)             ])
(0)             @pytest.mark.parametrize("include_generic", [True, False])
(0)             def test_svg_font_string(font_str, include_generic):
(4)                 fp = fm.FontProperties(family=["WenQuanYi Zen Hei"])
(4)                 if Path(fm.findfont(fp)).name != "wqy-zenhei.ttc":
(8)                     pytest.skip("Font may be missing")
(4)                 explicit, *rest, generic = map(
(8)                     lambda x: x.strip("'"), font_str.split(", ")
(4)                 )
(4)                 size = len(generic)
(4)                 if include_generic:
(8)                     rest = rest + [generic]
(4)                 plt.rcParams[f"font.{generic}"] = rest
(4)                 plt.rcParams["font.size"] = size
(4)                 plt.rcParams["svg.fonttype"] = "none"
(4)                 fig, ax = plt.subplots()
(4)                 if generic == "sans-serif":
(8)                     generic_options = ["sans", "sans-serif", "sans serif"]
(4)                 else:
(8)                     generic_options = [generic]
(4)                 for generic_name in generic_options:
(8)                     ax.text(0.5, 0.5, "There are 几个汉字 in between!",
(16)                            family=[explicit, generic_name], ha="center")
(8)                     ax.text(0.5, 0.1, "There are 几个汉字 in between!",
(16)                            family=[explicit, *rest, generic_name], ha="center")
(4)                 ax.axis("off")
(4)                 with BytesIO() as fd:
(8)                     fig.savefig(fd, format="svg")
(8)                     buf = fd.getvalue()
(4)                 tree = xml.etree.ElementTree.fromstring(buf)
(4)                 ns = "http://www.w3.org/2000/svg"
(4)                 text_count = 0
(4)                 for text_element in tree.findall(f".//{{{ns}}}text"):
(8)                     text_count += 1
(8)                     font_info = dict(
(12)                        map(lambda x: x.strip(), _.strip().split(":"))
(12)                        for _ in dict(text_element.items())["style"].split(";")
(8)                     )["font"]
(8)                     assert font_info == f"{size}px {font_str}"
(4)                 assert text_count == len(ax.texts)
(0)             def test_annotationbbox_gid():
(4)                 fig = plt.figure()
(4)                 ax = fig.add_subplot()
(4)                 arr_img = np.ones((32, 32))
(4)                 xy = (0.3, 0.55)
(4)                 imagebox = OffsetImage(arr_img, zoom=0.1)
(4)                 imagebox.image.axes = ax
(4)                 ab = AnnotationBbox(imagebox, xy,
(24)                                    xybox=(120., -80.),
(24)                                    xycoords='data',
(24)                                    boxcoords="offset points",
```

```
460: (24)                                   pad=0.5,
461: (24)                                   arrowprops=dict(
462: (28)                                       arrowstyle="->",
463: (28)                                       connectionstyle="angle,angleA=0,angleB=90,rad=3")
464: (24)                                   )
465: (4)                ab.set_gid("a test for issue 20044")
466: (4)                ax.add_artist(ab)
467: (4)                with BytesIO() as fd:
468: (8)                    fig.savefig(fd, format='svg')
469: (8)                    buf = fd.getvalue().decode('utf-8')
470: (4)                expected = '<g id="a test for issue 20044">'
471: (4)                assert expected in buf
```

File 163 - test_backend_template.py:

```
1: (0)              """
2: (0)              Backend-loading machinery tests, using variations on the template backend.
3: (0)              """
4: (0)              import sys
5: (0)              from types import SimpleNamespace
6: (0)              from unittest.mock import MagicMock
7: (0)              import matplotlib as mpl
8: (0)              from matplotlib import pyplot as plt
9: (0)              from matplotlib.backends import backend_template
10: (0)             from matplotlib.backends.backend_template import (
11: (4)                 FigureCanvasTemplate, FigureManagerTemplate)
12: (0)             def test_load_template():
13: (4)                 mpl.use("template")
14: (4)                 assert type(plt.figure().canvas) == FigureCanvasTemplate
15: (0)             def test_load_old_api(monkeypatch):
16: (4)                 mpl_test_backend = SimpleNamespace(**vars(backend_template))
17: (4)                 mpl_test_backend.new_figure_manager = (
18: (8)                     lambda num, *args, FigureClass=mpl.figure.Figure, **kwargs:
19: (8)                     FigureManagerTemplate(
20: (12)                        FigureCanvasTemplate(FigureClass(*args, **kwargs)), num))
21: (4)                 monkeypatch.setitem(sys.modules, "mpl_test_backend", mpl_test_backend)
22: (4)                 mpl.use("module://mpl_test_backend")
23: (4)                 assert type(plt.figure().canvas) == FigureCanvasTemplate
24: (4)                 plt.draw_if_interactive()
25: (0)             def test_show(monkeypatch):
26: (4)                 mpl_test_backend = SimpleNamespace(**vars(backend_template))
27: (4)                 mock_show = MagicMock()
28: (4)                 monkeypatch.setattr(
29: (8)                     mpl_test_backend.FigureManagerTemplate, "pyplot_show", mock_show)
30: (4)                 monkeypatch.setitem(sys.modules, "mpl_test_backend", mpl_test_backend)
31: (4)                 mpl.use("module://mpl_test_backend")
32: (4)                 plt.show()
33: (4)                 mock_show.assert_called_with()
34: (0)             def test_show_old_global_api(monkeypatch):
35: (4)                 mpl_test_backend = SimpleNamespace(**vars(backend_template))
36: (4)                 mock_show = MagicMock()
37: (4)                 monkeypatch.setattr(mpl_test_backend, "show", mock_show, raising=False)
38: (4)                 monkeypatch.setitem(sys.modules, "mpl_test_backend", mpl_test_backend)
39: (4)                 mpl.use("module://mpl_test_backend")
40: (4)                 plt.show()
41: (4)                 mock_show.assert_called_with()
42: (0)             def test_load_case_sensitive(monkeypatch):
43: (4)                 mpl_test_backend = SimpleNamespace(**vars(backend_template))
44: (4)                 mock_show = MagicMock()
45: (4)                 monkeypatch.setattr(
46: (8)                     mpl_test_backend.FigureManagerTemplate, "pyplot_show", mock_show)
47: (4)                 monkeypatch.setitem(sys.modules, "mpl_Test_Backend", mpl_test_backend)
48: (4)                 mpl.use("module://mpl_Test_Backend")
49: (4)                 plt.show()
50: (4)                 mock_show.assert_called_with()
```

File 164 - test_backend_tk.py:

```
1: (0)              import functools
2: (0)              import importlib
3: (0)              import os
4: (0)              import platform
5: (0)              import subprocess
6: (0)              import sys
7: (0)              import pytest
8: (0)              from matplotlib import _c_internal_utils
9: (0)              from matplotlib.testing import subprocess_run_helper
10: (0)             _test_timeout = 60  # A reasonably safe value for slower architectures.
11: (0)             def _isolated_tk_test(success_count, func=None):
12: (4)                 """
13: (4)                 A decorator to run *func* in a subprocess and assert that it prints
14: (4)                 "success" *success_count* times and nothing on stderr.
15: (4)                 TkAgg tests seem to have interactions between tests, so isolate each test
16: (4)                 in a subprocess. See GH#18261
17: (4)                 """
18: (4)                 if func is None:
19: (8)                     return functools.partial(_isolated_tk_test, success_count)
20: (4)                 if "MPL_TEST_ESCAPE_HATCH" in os.environ:
21: (8)                     return func
22: (4)                 @pytest.mark.skipif(
23: (8)                     not importlib.util.find_spec('tkinter'),
24: (8)                     reason="missing tkinter"
25: (4)                 )
26: (4)                 @pytest.mark.skipif(
27: (8)                     sys.platform == "linux" and not _c_internal_utils.display_is_valid(),
28: (8)                     reason="$DISPLAY and $WAYLAND_DISPLAY are unset"
29: (4)                 )
30: (4)                 @pytest.mark.xfail(  # https://github.com/actions/setup-python/issues/649
31: (8)                     ('TF_BUILD' in os.environ or 'GITHUB_ACTION' in os.environ) and
32: (8)                     sys.platform == 'darwin' and sys.version_info[:2] < (3, 11),
33: (8)                     reason='Tk version mismatch on Azure macOS CI'
34: (4)                 )
35: (4)                 @functools.wraps(func)
36: (4)                 def test_func():
37: (8)                     pytest.importorskip('tkinter')
38: (8)                     try:
39: (12)                        proc = subprocess_run_helper(
40: (16)                            func, timeout=_test_timeout, extra_env=dict(
41: (20)                                MPLBACKEND="TkAgg", MPL_TEST_ESCAPE_HATCH="1"))
42: (8)                     except subprocess.TimeoutExpired:
43: (12)                        pytest.fail("Subprocess timed out")
44: (8)                     except subprocess.CalledProcessError as e:
45: (12)                        pytest.fail("Subprocess failed to test intended behavior\n"
46: (24)                                    + str(e.stderr))
47: (8)                     else:
48: (12)                        ignored_lines = ["OpenGL", "CFMessagePort: bootstrap_register",
49: (29)                                         "/usr/include/servers/bootstrap_defs.h"]
50: (12)                        assert not [line for line in proc.stderr.splitlines()
51: (24)                                    if all(msg not in line for msg in ignored_lines)]
52: (12)                        assert proc.stdout.count("success") == success_count
53: (4)                 return test_func
54: (0)             @_isolated_tk_test(success_count=6)  # len(bad_boxes)
55: (0)             def test_blit():
56: (4)                 import matplotlib.pyplot as plt
57: (4)                 import numpy as np
58: (4)                 import matplotlib.backends.backend_tkagg  # noqa
59: (4)                 from matplotlib.backends import _backend_tk, _tkagg
60: (4)                 fig, ax = plt.subplots()
61: (4)                 photoimage = fig.canvas._tkphoto
62: (4)                 data = np.ones((4, 4, 4), dtype=np.uint8)
63: (4)                 bad_boxes = ((-1, 2, 0, 2),
64: (17)                             (2, 0, 0, 2),
65: (17)                             (1, 6, 0, 2),
66: (17)                             (0, 2, -1, 2),
```

```
 67: (17)                             (0, 2, 2, 0),
 68: (17)                             (0, 2, 1, 6))
 69: (4)                 for bad_box in bad_boxes:
 70: (8)                     try:
 71: (12)                        _tkagg.blit(
 72: (16)                            photoimage.tk.interpaddr(), str(photoimage), data,
 73: (16)                            _tkagg.TK_PHOTO_COMPOSITE_OVERLAY, (0, 1, 2, 3), bad_box)
 74: (8)                     except ValueError:
 75: (12)                        print("success")
 76: (4)                 plt.close(fig)
 77: (4)                 _backend_tk.blit(photoimage, data, (0, 1, 2, 3))
 78: (0)             @_isolated_tk_test(success_count=1)
 79: (0)             def test_figuremanager_preserves_host_mainloop():
 80: (4)                 import tkinter
 81: (4)                 import matplotlib.pyplot as plt
 82: (4)                 success = []
 83: (4)                 def do_plot():
 84: (8)                     plt.figure()
 85: (8)                     plt.plot([1, 2], [3, 5])
 86: (8)                     plt.close()
 87: (8)                     root.after(0, legitimate_quit)
 88: (4)                 def legitimate_quit():
 89: (8)                     root.quit()
 90: (8)                     success.append(True)
 91: (4)                 root = tkinter.Tk()
 92: (4)                 root.after(0, do_plot)
 93: (4)                 root.mainloop()
 94: (4)                 if success:
 95: (8)                     print("success")
 96: (0)             @pytest.mark.skipif(platform.python_implementation() != 'CPython',
 97: (20)                                 reason='PyPy does not support Tkinter threading: '
 98: (27)
'https://foss.heptapod.net/pypy/pypy/-/issues/1929')
 99: (0)             @pytest.mark.flaky(reruns=3)
100: (0)             @_isolated_tk_test(success_count=1)
101: (0)             def test_figuremanager_cleans_own_mainloop():
102: (4)                 import tkinter
103: (4)                 import time
104: (4)                 import matplotlib.pyplot as plt
105: (4)                 import threading
106: (4)                 from matplotlib.cbook import _get_running_interactive_framework
107: (4)                 root = tkinter.Tk()
108: (4)                 plt.plot([1, 2, 3], [1, 2, 5])
109: (4)                 def target():
110: (8)                     while not 'tk' == _get_running_interactive_framework():
111: (12)                        time.sleep(.01)
112: (8)                     plt.close()
113: (8)                     if show_finished_event.wait():
114: (12)                        print('success')
115: (4)                 show_finished_event = threading.Event()
116: (4)                 thread = threading.Thread(target=target, daemon=True)
117: (4)                 thread.start()
118: (4)                 plt.show(block=True)  # Testing if this function hangs.
119: (4)                 show_finished_event.set()
120: (4)                 thread.join()
121: (0)             @pytest.mark.flaky(reruns=3)
122: (0)             @_isolated_tk_test(success_count=0)
123: (0)             def test_never_update():
124: (4)                 import tkinter
125: (4)                 del tkinter.Misc.update
126: (4)                 del tkinter.Misc.update_idletasks
127: (4)                 import matplotlib.pyplot as plt
128: (4)                 fig = plt.figure()
129: (4)                 plt.show(block=False)
130: (4)                 plt.draw()  # Test FigureCanvasTkAgg.
131: (4)                 fig.canvas.toolbar.configure_subplots()  # Test NavigationToolbar2Tk.
132: (4)                 fig.canvas.get_tk_widget().after(100, plt.close, fig)
133: (4)                 plt.show(block=True)
134: (0)             @_isolated_tk_test(success_count=2)
```

```
135: (0)            def test_missing_back_button():
136: (4)                import matplotlib.pyplot as plt
137: (4)                from matplotlib.backends.backend_tkagg import NavigationToolbar2Tk
138: (4)                class Toolbar(NavigationToolbar2Tk):
139: (8)                    toolitems = [t for t in NavigationToolbar2Tk.toolitems if
140: (21)                                t[0] in ('Home', 'Pan', 'Zoom')]
141: (4)                fig = plt.figure()
142: (4)                print("success")
143: (4)                Toolbar(fig.canvas, fig.canvas.manager.window)  # This should not raise.
144: (4)                print("success")
145: (0)            @_isolated_tk_test(success_count=1)
146: (0)            def test_canvas_focus():
147: (4)                import tkinter as tk
148: (4)                import matplotlib.pyplot as plt
149: (4)                success = []
150: (4)                def check_focus():
151: (8)                    tkcanvas = fig.canvas.get_tk_widget()
152: (8)                    if not tkcanvas.winfo_viewable():
153: (12)                       tkcanvas.wait_visibility()
154: (8)                    if tkcanvas.focus_lastfor() == tkcanvas:
155: (12)                       success.append(True)
156: (8)                    plt.close()
157: (8)                    root.destroy()
158: (4)                root = tk.Tk()
159: (4)                fig = plt.figure()
160: (4)                plt.plot([1, 2, 3])
161: (4)                root.after(0, plt.show)
162: (4)                root.after(100, check_focus)
163: (4)                root.mainloop()
164: (4)                if success:
165: (8)                    print("success")
166: (0)            @_isolated_tk_test(success_count=2)
167: (0)            def test_embedding():
168: (4)                import tkinter as tk
169: (4)                from matplotlib.backends.backend_tkagg import (
170: (8)                    FigureCanvasTkAgg, NavigationToolbar2Tk)
171: (4)                from matplotlib.backend_bases import key_press_handler
172: (4)                from matplotlib.figure import Figure
173: (4)                root = tk.Tk()
174: (4)                def test_figure(master):
175: (8)                    fig = Figure()
176: (8)                    ax = fig.add_subplot()
177: (8)                    ax.plot([1, 2, 3])
178: (8)                    canvas = FigureCanvasTkAgg(fig, master=master)
179: (8)                    canvas.draw()
180: (8)                    canvas.mpl_connect("key_press_event", key_press_handler)
181: (8)                    canvas.get_tk_widget().pack(expand=True, fill="both")
182: (8)                    toolbar = NavigationToolbar2Tk(canvas, master, pack_toolbar=False)
183: (8)                    toolbar.pack(expand=True, fill="x")
184: (8)                    canvas.get_tk_widget().forget()
185: (8)                    toolbar.forget()
186: (4)                test_figure(root)
187: (4)                print("success")
188: (4)                root.tk_setPalette(background="sky blue", selectColor="midnight blue",
189: (23)                                  foreground="white")
190: (4)                test_figure(root)
191: (4)                print("success")
```

File 165 - test_backend_tools.py:

```
1: (0)              import pytest
2: (0)              from matplotlib.backend_tools import ToolHelpBase
3: (0)              @pytest.mark.parametrize('rc_shortcut,expected', [
4: (4)                  ('home', 'Home'),
5: (4)                  ('backspace', 'Backspace'),
6: (4)                  ('f1', 'F1'),
7: (4)                  ('ctrl+a', 'Ctrl+A'),
```

```
8: (4)                  ('ctrl+A', 'Ctrl+Shift+A'),
9: (4)                  ('a', 'a'),
10: (4)                 ('A', 'A'),
11: (4)                 ('ctrl+shift+f1', 'Ctrl+Shift+F1'),
12: (4)                 ('1', '1'),
13: (4)                 ('cmd+p', 'Cmd+P'),
14: (4)                 ('cmd+1', 'Cmd+1'),
15: (0)             ])
16: (0)             def test_format_shortcut(rc_shortcut, expected):
17: (4)                 assert ToolHelpBase.format_shortcut(rc_shortcut) == expected

----------------------------------------

File 166 - test_basic.py:

1: (0)              import builtins
2: (0)              import os
3: (0)              import sys
4: (0)              import textwrap
5: (0)              from matplotlib.testing import subprocess_run_for_testing
6: (0)              def test_simple():
7: (4)                  assert 1 + 1 == 2
8: (0)              def test_override_builtins():
9: (4)                  import pylab  # type: ignore
10: (4)                 ok_to_override = {
11: (8)                     '__name__',
12: (8)                     '__doc__',
13: (8)                     '__package__',
14: (8)                     '__loader__',
15: (8)                     '__spec__',
16: (8)                     'any',
17: (8)                     'all',
18: (8)                     'sum',
19: (8)                     'divmod'
20: (4)                 }
21: (4)                 overridden = {key for key in {*dir(pylab)} & {*dir(builtins)}
22: (18)                              if getattr(pylab, key) != getattr(builtins, key)}
23: (4)                 assert overridden <= ok_to_override
24: (0)             def test_lazy_imports():
25: (4)                 source = textwrap.dedent("""
26: (4)                 import sys
27: (4)                 import matplotlib.figure
28: (4)                 import matplotlib.backend_bases
29: (4)                 import matplotlib.pyplot
30: (4)                 assert 'matplotlib._tri' not in sys.modules
31: (4)                 assert 'matplotlib._qhull' not in sys.modules
32: (4)                 assert 'matplotlib._contour' not in sys.modules
33: (4)                 assert 'urllib.request' not in sys.modules
34: (4)                 """)
35: (4)                 subprocess_run_for_testing(
36: (8)                     [sys.executable, '-c', source],
37: (8)                     env={**os.environ, "MPLBACKEND": "", "MATPLOTLIBRC": os.devnull},
38: (8)                     check=True)

----------------------------------------

File 167 - test_backend_webagg.py:

1: (0)              import os
2: (0)              import sys
3: (0)              import pytest
4: (0)              import matplotlib.backends.backend_webagg_core
5: (0)              from matplotlib.testing import subprocess_run_for_testing
6: (0)              @pytest.mark.parametrize("backend", ["webagg", "nbagg"])
7: (0)              def test_webagg_fallback(backend):
8: (4)                  pytest.importorskip("tornado")
9: (4)                  if backend == "nbagg":
10: (8)                     pytest.importorskip("IPython")
11: (4)                 env = dict(os.environ)
```

```
12: (4)                 if sys.platform != "win32":
13: (8)                     env["DISPLAY"] = ""
14: (4)                 env["MPLBACKEND"] = backend
15: (4)                 test_code = (
16: (8)                     "import os;"
17: (8)                     + f"assert os.environ['MPLBACKEND'] == '{backend}';"
18: (8)                     + "import matplotlib.pyplot as plt; "
19: (8)                     + "print(plt.get_backend());"
20: (8)                     f"assert '{backend}' == plt.get_backend().lower();"
21: (4)                 )
22: (4)                 subprocess_run_for_testing([sys.executable, "-c", test_code], env=env,
check=True)
23: (0)             def test_webagg_core_no_toolbar():
24: (4)                 fm = matplotlib.backends.backend_webagg_core.FigureManagerWebAgg
25: (4)                 assert fm._toolbar2_class is None
```

File 168 - test_bbox_tight.py:

```
1: (0)              from io import BytesIO
2: (0)              import platform
3: (0)              import numpy as np
4: (0)              from matplotlib.testing.decorators import image_comparison
5: (0)              import matplotlib.pyplot as plt
6: (0)              import matplotlib.path as mpath
7: (0)              import matplotlib.patches as mpatches
8: (0)              from matplotlib.ticker import FuncFormatter
9: (0)              @image_comparison(['bbox_inches_tight'], remove_text=True,
10: (18)                              savefig_kwarg={'bbox_inches': 'tight'})
11: (0)             def test_bbox_inches_tight():
12: (4)                 data = [[66386, 174296, 75131, 577908, 32015],
13: (12)                        [58230, 381139, 78045, 99308, 160454],
14: (12)                        [89135, 80552, 152558, 497981, 603535],
15: (12)                        [78415, 81858, 150656, 193263, 69638],
16: (12)                        [139361, 331509, 343164, 781380, 52269]]
17: (4)                 col_labels = row_labels = [''] * 5
18: (4)                 rows = len(data)
19: (4)                 ind = np.arange(len(col_labels)) + 0.3  # the x locations for the groups
20: (4)                 cell_text = []
21: (4)                 width = 0.4  # the width of the bars
22: (4)                 yoff = np.zeros(len(col_labels))
23: (4)                 fig, ax = plt.subplots(1, 1)
24: (4)                 for row in range(rows):
25: (8)                     ax.bar(ind, data[row], width, bottom=yoff, align='edge', color='b')
26: (8)                     yoff = yoff + data[row]
27: (8)                     cell_text.append([''])
28: (4)                 plt.xticks([])
29: (4)                 plt.xlim(0, 5)
30: (4)                 plt.legend([''] * 5, loc=(1.2, 0.2))
31: (4)                 fig.legend([''] * 5, bbox_to_anchor=(0, 0.2), loc='lower left')
32: (4)                 cell_text.reverse()
33: (4)                 plt.table(cellText=cell_text, rowLabels=row_labels, colLabels=col_labels,
34: (14)                          loc='bottom')
35: (0)             @image_comparison(['bbox_inches_tight_suptile_legend'],
36: (18)                              savefig_kwarg={'bbox_inches': 'tight'},
37: (18)                              tol=0.02 if platform.machine() == 'arm64' else 0)
38: (0)             def test_bbox_inches_tight_suptile_legend():
39: (4)                 plt.plot(np.arange(10), label='a straight line')
40: (4)                 plt.legend(bbox_to_anchor=(0.9, 1), loc='upper left')
41: (4)                 plt.title('Axis title')
42: (4)                 plt.suptitle('Figure title')
43: (4)                 def y_formatter(y, pos):
44: (8)                     if int(y) == 4:
45: (12)                        return 'The number 4'
46: (8)                     else:
47: (12)                        return str(y)
48: (4)                 plt.gca().yaxis.set_major_formatter(FuncFormatter(y_formatter))
49: (4)                 plt.xlabel('X axis')
```

```
 50: (0)             @image_comparison(['bbox_inches_tight_suptile_non_default.png'],
 51: (18)                              savefig_kwarg={'bbox_inches': 'tight'},
 52: (18)                              tol=0.1)  # large tolerance because only testing clipping.
 53: (0)             def test_bbox_inches_tight_suptitle_non_default():
 54: (4)                 fig, ax = plt.subplots()
 55: (4)                 fig.suptitle('Booo', x=0.5, y=1.1)
 56: (0)             @image_comparison(['bbox_inches_tight_layout.png'], remove_text=True,
 57: (18)                              style='mpl20',
 58: (18)                              savefig_kwarg=dict(bbox_inches='tight',
pad_inches='layout'))
 59: (0)             def test_bbox_inches_tight_layout_constrained():
 60: (4)                 fig, ax = plt.subplots(layout='constrained')
 61: (4)                 fig.get_layout_engine().set(h_pad=0.5)
 62: (4)                 ax.set_aspect('equal')
 63: (0)             def test_bbox_inches_tight_layout_notconstrained(tmp_path):
 64: (4)                 fig, ax = plt.subplots()
 65: (4)                 fig.savefig(tmp_path / 'foo.png', bbox_inches='tight',
pad_inches='layout')
 66: (0)             @image_comparison(['bbox_inches_tight_clipping'],
 67: (18)                              remove_text=True, savefig_kwarg={'bbox_inches': 'tight'})
 68: (0)             def test_bbox_inches_tight_clipping():
 69: (4)                 plt.scatter(np.arange(10), np.arange(10))
 70: (4)                 ax = plt.gca()
 71: (4)                 ax.set_xlim(0, 5)
 72: (4)                 ax.set_ylim(0, 5)
 73: (4)                 patch = mpatches.Rectangle([-50, -50], 100, 100,
 74: (31)                                           transform=ax.transData,
 75: (31)                                           facecolor='blue', alpha=0.5)
 76: (4)                 path = mpath.Path.unit_regular_star(5).deepcopy()
 77: (4)                 path.vertices *= 0.25
 78: (4)                 patch.set_clip_path(path, transform=ax.transAxes)
 79: (4)                 plt.gcf().artists.append(patch)
 80: (0)             @image_comparison(['bbox_inches_tight_raster'],
 81: (18)                              remove_text=True, savefig_kwarg={'bbox_inches': 'tight'})
 82: (0)             def test_bbox_inches_tight_raster():
 83: (4)                 """Test rasterization with tight_layout"""
 84: (4)                 fig, ax = plt.subplots()
 85: (4)                 ax.plot([1.0, 2.0], rasterized=True)
 86: (0)             def test_only_on_non_finite_bbox():
 87: (4)                 fig, ax = plt.subplots()
 88: (4)                 ax.annotate("", xy=(0, float('nan')))
 89: (4)                 ax.set_axis_off()
 90: (4)                 fig.savefig(BytesIO(), bbox_inches='tight', format='png')
 91: (0)             def test_tight_pcolorfast():
 92: (4)                 fig, ax = plt.subplots()
 93: (4)                 ax.pcolorfast(np.arange(4).reshape((2, 2)))
 94: (4)                 ax.set(ylim=(0, .1))
 95: (4)                 buf = BytesIO()
 96: (4)                 fig.savefig(buf, bbox_inches="tight")
 97: (4)                 buf.seek(0)
 98: (4)                 height, width, _ = plt.imread(buf).shape
 99: (4)                 assert width > height
 100: (0)            def test_noop_tight_bbox():
 101: (4)                from PIL import Image
 102: (4)                x_size, y_size = (10, 7)
 103: (4)                dpi = 100
 104: (4)                fig = plt.figure(frameon=False, dpi=dpi, figsize=(x_size/dpi, y_size/dpi))
 105: (4)                ax = fig.add_axes((0, 0, 1, 1))
 106: (4)                ax.set_axis_off()
 107: (4)                ax.xaxis.set_visible(False)
 108: (4)                ax.yaxis.set_visible(False)
 109: (4)                data = np.arange(x_size * y_size).reshape(y_size, x_size)
 110: (4)                ax.imshow(data, rasterized=True)
 111: (4)                fig.savefig(BytesIO(), bbox_inches='tight', pad_inches=0, format='pdf')
 112: (4)                out = BytesIO()
 113: (4)                fig.savefig(out, bbox_inches='tight', pad_inches=0)
 114: (4)                out.seek(0)
 115: (4)                im = np.asarray(Image.open(out))
 116: (4)                assert (im[:, :, 3] == 255).all()
```

```
117: (4)                assert not (im[:, :, :3] == 255).all()
118: (4)                assert im.shape == (7, 10, 4)
119: (0)            @image_comparison(['bbox_inches_fixed_aspect'], extensions=['png'],
120: (18)                             remove_text=True, savefig_kwarg={'bbox_inches': 'tight'})
121: (0)            def test_bbox_inches_fixed_aspect():
122: (4)                with plt.rc_context({'figure.constrained_layout.use': True}):
123: (8)                    fig, ax = plt.subplots()
124: (8)                    ax.plot([0, 1])
125: (8)                    ax.set_xlim(0, 1)
126: (8)                    ax.set_aspect('equal')
```

File 169 - test_category.py:

```
1: (0)              """Catch all for categorical functions"""
2: (0)              import warnings
3: (0)              import pytest
4: (0)              import numpy as np
5: (0)              import matplotlib as mpl
6: (0)              from matplotlib.axes import Axes
7: (0)              import matplotlib.pyplot as plt
8: (0)              import matplotlib.category as cat
9: (0)              from matplotlib.testing.decorators import check_figures_equal
10: (0)             class TestUnitData:
11: (4)                 test_cases = [('single', (["hello world"], [0])),
12: (18)                              ('unicode', (["Здравствуйте мир"], [0])),
13: (18)                              ('mixed', (['A', "np.nan", 'B', "3.14", "мир"],
14: (29)                                         [0, 1, 2, 3, 4]))]
15: (4)                 ids, data = zip(*test_cases)
16: (4)                 @pytest.mark.parametrize("data, locs", data, ids=ids)
17: (4)                 def test_unit(self, data, locs):
18: (8)                     unit = cat.UnitData(data)
19: (8)                     assert list(unit._mapping.keys()) == data
20: (8)                     assert list(unit._mapping.values()) == locs
21: (4)                 def test_update(self):
22: (8)                     data = ['a', 'd']
23: (8)                     locs = [0, 1]
24: (8)                     data_update = ['b', 'd', 'e']
25: (8)                     unique_data = ['a', 'd', 'b', 'e']
26: (8)                     updated_locs = [0, 1, 2, 3]
27: (8)                     unit = cat.UnitData(data)
28: (8)                     assert list(unit._mapping.keys()) == data
29: (8)                     assert list(unit._mapping.values()) == locs
30: (8)                     unit.update(data_update)
31: (8)                     assert list(unit._mapping.keys()) == unique_data
32: (8)                     assert list(unit._mapping.values()) == updated_locs
33: (4)                 failing_test_cases = [("number", 3.14), ("nan", np.nan),
34: (26)                                      ("list", [3.14, 12]), ("mixed type", ["A", 2])]
35: (4)                 fids, fdata = zip(*test_cases)
36: (4)                 @pytest.mark.parametrize("fdata", fdata, ids=fids)
37: (4)                 def test_non_string_fails(self, fdata):
38: (8)                     with pytest.raises(TypeError):
39: (12)                        cat.UnitData(fdata)
40: (4)                 @pytest.mark.parametrize("fdata", fdata, ids=fids)
41: (4)                 def test_non_string_update_fails(self, fdata):
42: (8)                     unitdata = cat.UnitData()
43: (8)                     with pytest.raises(TypeError):
44: (12)                        unitdata.update(fdata)
45: (0)             class FakeAxis:
46: (4)                 def __init__(self, units):
47: (8)                     self.units = units
48: (0)             class TestStrCategoryConverter:
49: (4)                 """
50: (4)                 Based on the pandas conversion and factorization tests:
51: (4)                 ref: /pandas/tseries/tests/test_converter.py
52: (9)                      /pandas/tests/test_algos.py:TestFactorize
53: (4)                 """
54: (4)                 test_cases = [("unicode", ["Здравствуйте мир"]),
```

```
 55: (18)                                ("ascii", ["hello world"]),
 56: (18)                                ("single", ['a', 'b', 'c']),
 57: (18)                                ("integer string", ["1", "2"]),
 58: (18)                                ("single + values>10", ["A", "B", "C", "D", "E", "F", "G",
 59: (42)                                                        "H", "I", "J", "K", "L", "M", "N",
 60: (42)                                                        "O", "P", "Q", "R", "S", "T", "U",
 61: (42)                                                        "V", "W", "X", "Y", "Z"])]
 62: (4)                 ids, values = zip(*test_cases)
 63: (4)                 failing_test_cases = [("mixed", [3.14, 'A', np.inf]),
 64: (26)                                      ("string integer", ['42', 42])]
 65: (4)                 fids, fvalues = zip(*failing_test_cases)
 66: (4)                 @pytest.fixture(autouse=True)
 67: (4)                 def mock_axis(self, request):
 68: (8)                     self.cc = cat.StrCategoryConverter()
 69: (8)                     self.unit = cat.UnitData()
 70: (8)                     self.ax = FakeAxis(self.unit)
 71: (4)                 @pytest.mark.parametrize("vals", values, ids=ids)
 72: (4)                 def test_convert(self, vals):
 73: (8)                     np.testing.assert_allclose(self.cc.convert(vals, self.ax.units,
 74: (51)                                                                self.ax),
 75: (35)                                                 range(len(vals)))
 76: (4)                 @pytest.mark.parametrize("value", ["hi", "мир"], ids=["ascii", "unicode"])
 77: (4)                 def test_convert_one_string(self, value):
 78: (8)                     assert self.cc.convert(value, self.unit, self.ax) == 0
 79: (4)                 @pytest.mark.parametrize("fvals", fvalues, ids=fids)
 80: (4)                 def test_convert_fail(self, fvals):
 81: (8)                     with pytest.raises(TypeError):
 82: (12)                        self.cc.convert(fvals, self.unit, self.ax)
 83: (4)                 def test_axisinfo(self):
 84: (8)                     axis = self.cc.axisinfo(self.unit, self.ax)
 85: (8)                     assert isinstance(axis.majloc, cat.StrCategoryLocator)
 86: (8)                     assert isinstance(axis.majfmt, cat.StrCategoryFormatter)
 87: (4)                 def test_default_units(self):
 88: (8)                     assert isinstance(self.cc.default_units(["a"], self.ax), cat.UnitData)
 89: (0)             PLOT_LIST = [Axes.scatter, Axes.plot, Axes.bar]
 90: (0)             PLOT_IDS = ["scatter", "plot", "bar"]
 91: (0)             class TestStrCategoryLocator:
 92: (4)                 def test_StrCategoryLocator(self):
 93: (8)                     locs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
 94: (8)                     unit = cat.UnitData([str(j) for j in locs])
 95: (8)                     ticks = cat.StrCategoryLocator(unit._mapping)
 96: (8)                     np.testing.assert_array_equal(ticks.tick_values(None, None), locs)
 97: (4)                 @pytest.mark.parametrize("plotter", PLOT_LIST, ids=PLOT_IDS)
 98: (4)                 def test_StrCategoryLocatorPlot(self, plotter):
 99: (8)                     ax = plt.figure().subplots()
100: (8)                     plotter(ax, [1, 2, 3], ["a", "b", "c"])
101: (8)                     np.testing.assert_array_equal(ax.yaxis.major.locator(), range(3))
102: (0)             class TestStrCategoryFormatter:
103: (4)                 test_cases = [("ascii", ["hello", "world", "hi"]),
104: (18)                              ("unicode", ["Здравствуйте", "привет"])]
105: (4)                 ids, cases = zip(*test_cases)
106: (4)                 @pytest.mark.parametrize("ydata", cases, ids=ids)
107: (4)                 def test_StrCategoryFormatter(self, ydata):
108: (8)                     unit = cat.UnitData(ydata)
109: (8)                     labels = cat.StrCategoryFormatter(unit._mapping)
110: (8)                     for i, d in enumerate(ydata):
111: (12)                        assert labels(i, i) == d
112: (12)                        assert labels(i, None) == d
113: (4)                 @pytest.mark.parametrize("ydata", cases, ids=ids)
114: (4)                 @pytest.mark.parametrize("plotter", PLOT_LIST, ids=PLOT_IDS)
115: (4)                 def test_StrCategoryFormatterPlot(self, ydata, plotter):
116: (8)                     ax = plt.figure().subplots()
117: (8)                     plotter(ax, range(len(ydata)), ydata)
118: (8)                     for i, d in enumerate(ydata):
119: (12)                        assert ax.yaxis.major.formatter(i) == d
120: (8)                     assert ax.yaxis.major.formatter(i+1) == ""
121: (0)             def axis_test(axis, labels):
122: (4)                 ticks = list(range(len(labels)))
123: (4)                 np.testing.assert_array_equal(axis.get_majorticklocs(), ticks)
```

```
(4)                 graph_labels = [axis.major.formatter(i, i) for i in ticks]
(4)                 assert graph_labels == [cat.StrCategoryFormatter._text(l) for l in labels]
(4)                 assert list(axis.units._mapping.keys()) == [l for l in labels]
(4)                 assert list(axis.units._mapping.values()) == ticks
(0)             class TestPlotBytes:
(4)                 bytes_cases = [('string list', ['a', 'b', 'c']),
(19)                               ('bytes list', [b'a', b'b', b'c']),
(19)                               ('bytes ndarray', np.array([b'a', b'b', b'c']))]
(4)                 bytes_ids, bytes_data = zip(*bytes_cases)
(4)                 @pytest.mark.parametrize("plotter", PLOT_LIST, ids=PLOT_IDS)
(4)                 @pytest.mark.parametrize("bdata", bytes_data, ids=bytes_ids)
(4)                 def test_plot_bytes(self, plotter, bdata):
(8)                     ax = plt.figure().subplots()
(8)                     counts = np.array([4, 6, 5])
(8)                     plotter(ax, bdata, counts)
(8)                     axis_test(ax.xaxis, bdata)
(0)             class TestPlotNumlike:
(4)                 numlike_cases = [('string list', ['1', '11', '3']),
(21)                                 ('string ndarray', np.array(['1', '11', '3'])),
(21)                                 ('bytes list', [b'1', b'11', b'3']),
(21)                                 ('bytes ndarray', np.array([b'1', b'11', b'3']))]
(4)                 numlike_ids, numlike_data = zip(*numlike_cases)
(4)                 @pytest.mark.parametrize("plotter", PLOT_LIST, ids=PLOT_IDS)
(4)                 @pytest.mark.parametrize("ndata", numlike_data, ids=numlike_ids)
(4)                 def test_plot_numlike(self, plotter, ndata):
(8)                     ax = plt.figure().subplots()
(8)                     counts = np.array([4, 6, 5])
(8)                     plotter(ax, ndata, counts)
(8)                     axis_test(ax.xaxis, ndata)
(0)             class TestPlotTypes:
(4)                 @pytest.mark.parametrize("plotter", PLOT_LIST, ids=PLOT_IDS)
(4)                 def test_plot_unicode(self, plotter):
(8)                     ax = plt.figure().subplots()
(8)                     words = ['Здравствуйте', 'привет']
(8)                     plotter(ax, words, [0, 1])
(8)                     axis_test(ax.xaxis, words)
(4)                 @pytest.fixture
(4)                 def test_data(self):
(8)                     self.x = ["hello", "happy", "world"]
(8)                     self.xy = [2, 6, 3]
(8)                     self.y = ["Python", "is", "fun"]
(8)                     self.yx = [3, 4, 5]
(4)                 @pytest.mark.usefixtures("test_data")
(4)                 @pytest.mark.parametrize("plotter", PLOT_LIST, ids=PLOT_IDS)
(4)                 def test_plot_xaxis(self, test_data, plotter):
(8)                     ax = plt.figure().subplots()
(8)                     plotter(ax, self.x, self.xy)
(8)                     axis_test(ax.xaxis, self.x)
(4)                 @pytest.mark.usefixtures("test_data")
(4)                 @pytest.mark.parametrize("plotter", PLOT_LIST, ids=PLOT_IDS)
(4)                 def test_plot_yaxis(self, test_data, plotter):
(8)                     ax = plt.figure().subplots()
(8)                     plotter(ax, self.yx, self.y)
(8)                     axis_test(ax.yaxis, self.y)
(4)                 @pytest.mark.usefixtures("test_data")
(4)                 @pytest.mark.parametrize("plotter", PLOT_LIST, ids=PLOT_IDS)
(4)                 def test_plot_xyaxis(self, test_data, plotter):
(8)                     ax = plt.figure().subplots()
(8)                     plotter(ax, self.x, self.y)
(8)                     axis_test(ax.xaxis, self.x)
(8)                     axis_test(ax.yaxis, self.y)
(4)                 @pytest.mark.parametrize("plotter", PLOT_LIST, ids=PLOT_IDS)
(4)                 def test_update_plot(self, plotter):
(8)                     ax = plt.figure().subplots()
(8)                     plotter(ax, ['a', 'b'], ['e', 'g'])
(8)                     plotter(ax, ['a', 'b', 'd'], ['f', 'a', 'b'])
(8)                     plotter(ax, ['b', 'c', 'd'], ['g', 'e', 'd'])
(8)                     axis_test(ax.xaxis, ['a', 'b', 'd', 'c'])
(8)                     axis_test(ax.yaxis, ['e', 'g', 'f', 'a', 'b', 'd'])
```

```
193: (4)                failing_test_cases = [("mixed", ['A', 3.14]),
194: (26)                                     ("number integer", ['1', 1]),
195: (26)                                     ("string integer", ['42', 42]),
196: (26)                                     ("missing", ['12', np.nan])]
197: (4)                fids, fvalues = zip(*failing_test_cases)
198: (4)                plotters = [Axes.scatter, Axes.bar,
199: (16)                           pytest.param(Axes.plot, marks=pytest.mark.xfail)]
200: (4)                @pytest.mark.parametrize("plotter", plotters)
201: (4)                @pytest.mark.parametrize("xdata", fvalues, ids=fids)
202: (4)                def test_mixed_type_exception(self, plotter, xdata):
203: (8)                    ax = plt.figure().subplots()
204: (8)                    with pytest.raises(TypeError):
205: (12)                       plotter(ax, xdata, [1, 2])
206: (4)                @pytest.mark.parametrize("plotter", plotters)
207: (4)                @pytest.mark.parametrize("xdata", fvalues, ids=fids)
208: (4)                def test_mixed_type_update_exception(self, plotter, xdata):
209: (8)                    ax = plt.figure().subplots()
210: (8)                    with pytest.raises(TypeError):
211: (12)                       plotter(ax, [0, 3], [1, 3])
212: (12)                       plotter(ax, xdata, [1, 2])
213: (0)            @mpl.style.context('default')
214: (0)            @check_figures_equal(extensions=["png"])
215: (0)            def test_overriding_units_in_plot(fig_test, fig_ref):
216: (4)                from datetime import datetime
217: (4)                t0 = datetime(2018, 3, 1)
218: (4)                t1 = datetime(2018, 3, 2)
219: (4)                t2 = datetime(2018, 3, 3)
220: (4)                t3 = datetime(2018, 3, 4)
221: (4)                ax_test = fig_test.subplots()
222: (4)                ax_ref = fig_ref.subplots()
223: (4)                for ax, kwargs in zip([ax_test, ax_ref],
224: (26)                                     ({}, dict(xunits=None, yunits=None))):
225: (8)                    ax.plot([t0, t1], ["V1", "V2"], **kwargs)
226: (8)                    x_units = ax.xaxis.units
227: (8)                    y_units = ax.yaxis.units
228: (8)                    ax.plot([t2, t3], ["V1", "V2"], **kwargs)
229: (8)                    assert x_units is ax.xaxis.units
230: (8)                    assert y_units is ax.yaxis.units
231: (0)            def test_no_deprecation_on_empty_data():
232: (4)                """
233: (4)                Smoke test to check that no deprecation warning is emitted. See #22640.
234: (4)                """
235: (4)                f, ax = plt.subplots()
236: (4)                ax.xaxis.update_units(["a", "b"])
237: (4)                ax.plot([], [])
238: (0)            def test_hist():
239: (4)                fig, ax = plt.subplots()
240: (4)                n, bins, patches = ax.hist(['a', 'b', 'a', 'c', 'ff'])
241: (4)                assert n.shape == (10,)
242: (4)                np.testing.assert_allclose(n, [2., 0., 0., 1., 0., 0., 1., 0., 0., 1.])
243: (0)            def test_set_lim():
244: (4)                f, ax = plt.subplots()
245: (4)                ax.plot(["a", "b", "c", "d"], [1, 2, 3, 4])
246: (4)                with warnings.catch_warnings():
247: (8)                    ax.set_xlim("b", "c")
```

File 170 - test_cbook.py:

```
1: (0)              from __future__ import annotations
2: (0)              import sys
3: (0)              import itertools
4: (0)              import pickle
5: (0)              from typing import Any
6: (0)              from unittest.mock import patch, Mock
7: (0)              from datetime import datetime, date, timedelta
8: (0)              import numpy as np
9: (0)              from numpy.testing import (assert_array_equal, assert_approx_equal,
```

```
(27)                                assert_array_almost_equal)
(0)             import pytest
(0)             from matplotlib import _api, cbook
(0)             import matplotlib.colors as mcolors
(0)             from matplotlib.cbook import delete_masked_points, strip_math
(0)             from types import ModuleType
(0)             class Test_delete_masked_points:
(4)                 def test_bad_first_arg(self):
(8)                     with pytest.raises(ValueError):
(12)                        delete_masked_points('a string', np.arange(1.0, 7.0))
(4)                 def test_string_seq(self):
(8)                     a1 = ['a', 'b', 'c', 'd', 'e', 'f']
(8)                     a2 = [1, 2, 3, np.nan, np.nan, 6]
(8)                     result1, result2 = delete_masked_points(a1, a2)
(8)                     ind = [0, 1, 2, 5]
(8)                     assert_array_equal(result1, np.array(a1)[ind])
(8)                     assert_array_equal(result2, np.array(a2)[ind])
(4)                 def test_datetime(self):
(8)                     dates = [datetime(2008, 1, 1), datetime(2008, 1, 2),
(17)                             datetime(2008, 1, 3), datetime(2008, 1, 4),
(17)                             datetime(2008, 1, 5), datetime(2008, 1, 6)]
(8)                     a_masked = np.ma.array([1, 2, 3, np.nan, np.nan, 6],
(31)                                           mask=[False, False, True, True, False, False])
(8)                     actual = delete_masked_points(dates, a_masked)
(8)                     ind = [0, 1, 5]
(8)                     assert_array_equal(actual[0], np.array(dates)[ind])
(8)                     assert_array_equal(actual[1], a_masked[ind].compressed())
(4)                 def test_rgba(self):
(8)                     a_masked = np.ma.array([1, 2, 3, np.nan, np.nan, 6],
(31)                                           mask=[False, False, True, True, False, False])
(8)                     a_rgba = mcolors.to_rgba_array(['r', 'g', 'b', 'c', 'm', 'y'])
(8)                     actual = delete_masked_points(a_masked, a_rgba)
(8)                     ind = [0, 1, 5]
(8)                     assert_array_equal(actual[0], a_masked[ind].compressed())
(8)                     assert_array_equal(actual[1], a_rgba[ind])
(0)             class Test_boxplot_stats:
(4)                 def setup_method(self):
(8)                     np.random.seed(937)
(8)                     self.nrows = 37
(8)                     self.ncols = 4
(8)                     self.data = np.random.lognormal(size=(self.nrows, self.ncols),
(40)                                                    mean=1.5, sigma=1.75)
(8)                     self.known_keys = sorted([
(12)                        'mean', 'med', 'q1', 'q3', 'iqr',
(12)                        'cilo', 'cihi', 'whislo', 'whishi',
(12)                        'fliers', 'label'
(8)                     ])
(8)                     self.std_results = cbook.boxplot_stats(self.data)
(8)                     self.known_nonbootstrapped_res = {
(12)                        'cihi': 6.8161283264444847,
(12)                        'cilo': -0.1489815330368689,
(12)                        'iqr': 13.492709959447094,
(12)                        'mean': 13.00447442387868,
(12)                        'med': 3.3335733967038079,
(12)                        'fliers': np.array([
(16)                            92.55467075,  87.03819018,  42.23204914,  39.29390996
(12)                        ]),
(12)                        'q1': 1.3597529879465153,
(12)                        'q3': 14.85246294739361,
(12)                        'whishi': 27.899688243699629,
(12)                        'whislo': 0.042143774965502923
(8)                     }
(8)                     self.known_bootstrapped_ci = {
(12)                        'cihi': 8.939577523357828,
(12)                        'cilo': 1.8692703958676578,
(8)                     }
(8)                     self.known_whis3_res = {
(12)                        'whishi': 42.232049135969874,
(12)                        'whislo': 0.042143774965502923,
```

```
(12)                        'fliers': np.array([92.55467075, 87.03819018]),
(8)                     }
(8)                     self.known_res_percentiles = {
(12)                        'whislo':   0.1933685896907924,
(12)                        'whishi':  42.232049135969874
(8)                     }
(8)                     self.known_res_range = {
(12)                        'whislo': 0.042143774965502923,
(12)                        'whishi': 92.554670752188699
(8)                     }
(4)                 def test_form_main_list(self):
(8)                     assert isinstance(self.std_results, list)
(4)                 def test_form_each_dict(self):
(8)                     for res in self.std_results:
(12)                        assert isinstance(res, dict)
(4)                 def test_form_dict_keys(self):
(8)                     for res in self.std_results:
(12)                        assert set(res) <= set(self.known_keys)
(4)                 def test_results_baseline(self):
(8)                     res = self.std_results[0]
(8)                     for key, value in self.known_nonbootstrapped_res.items():
(12)                        assert_array_almost_equal(res[key], value)
(4)                 def test_results_bootstrapped(self):
(8)                     results = cbook.boxplot_stats(self.data, bootstrap=10000)
(8)                     res = results[0]
(8)                     for key, value in self.known_bootstrapped_ci.items():
(12)                        assert_approx_equal(res[key], value)
(4)                 def test_results_whiskers_float(self):
(8)                     results = cbook.boxplot_stats(self.data, whis=3)
(8)                     res = results[0]
(8)                     for key, value in self.known_whis3_res.items():
(12)                        assert_array_almost_equal(res[key], value)
(4)                 def test_results_whiskers_range(self):
(8)                     results = cbook.boxplot_stats(self.data, whis=[0, 100])
(8)                     res = results[0]
(8)                     for key, value in self.known_res_range.items():
(12)                        assert_array_almost_equal(res[key], value)
(4)                 def test_results_whiskers_percentiles(self):
(8)                     results = cbook.boxplot_stats(self.data, whis=[5, 95])
(8)                     res = results[0]
(8)                     for key, value in self.known_res_percentiles.items():
(12)                        assert_array_almost_equal(res[key], value)
(4)                 def test_results_withlabels(self):
(8)                     labels = ['Test1', 2, 'Aardvark', 4]
(8)                     results = cbook.boxplot_stats(self.data, labels=labels)
(8)                     for lab, res in zip(labels, results):
(12)                        assert res['label'] == lab
(8)                     results = cbook.boxplot_stats(self.data)
(8)                     for res in results:
(12)                        assert 'label' not in res
(4)                 def test_label_error(self):
(8)                     labels = [1, 2]
(8)                     with pytest.raises(ValueError):
(12)                        cbook.boxplot_stats(self.data, labels=labels)
(4)                 def test_bad_dims(self):
(8)                     data = np.random.normal(size=(34, 34, 34))
(8)                     with pytest.raises(ValueError):
(12)                        cbook.boxplot_stats(data)
(4)                 def test_boxplot_stats_autorange_false(self):
(8)                     x = np.zeros(shape=140)
(8)                     x = np.hstack([-25, x, 25])
(8)                     bstats_false = cbook.boxplot_stats(x, autorange=False)
(8)                     bstats_true = cbook.boxplot_stats(x, autorange=True)
(8)                     assert bstats_false[0]['whislo'] == 0
(8)                     assert bstats_false[0]['whishi'] == 0
(8)                     assert_array_almost_equal(bstats_false[0]['fliers'], [-25, 25])
(8)                     assert bstats_true[0]['whislo'] == -25
(8)                     assert bstats_true[0]['whishi'] == 25
(8)                     assert_array_almost_equal(bstats_true[0]['fliers'], [])
```

```
(0)             class Test_callback_registry:
(4)                 def setup_method(self):
(8)                     self.signal = 'test'
(8)                     self.callbacks = cbook.CallbackRegistry()
(4)                 def connect(self, s, func, pickle):
(8)                     if pickle:
(12)                        return self.callbacks.connect(s, func)
(8)                     else:
(12)                        return self.callbacks._connect_picklable(s, func)
(4)                 def disconnect(self, cid):
(8)                     return self.callbacks.disconnect(cid)
(4)                 def count(self):
(8)                     count1 = len(self.callbacks._func_cid_map.get(self.signal, []))
(8)                     count2 = len(self.callbacks.callbacks.get(self.signal))
(8)                     assert count1 == count2
(8)                     return count1
(4)                 def is_empty(self):
(8)                     np.testing.break_cycles()
(8)                     assert self.callbacks._func_cid_map == {}
(8)                     assert self.callbacks.callbacks == {}
(8)                     assert self.callbacks._pickled_cids == set()
(4)                 def is_not_empty(self):
(8)                     np.testing.break_cycles()
(8)                     assert self.callbacks._func_cid_map != {}
(8)                     assert self.callbacks.callbacks != {}
(4)                 def test_cid_restore(self):
(8)                     cb = cbook.CallbackRegistry()
(8)                     cb.connect('a', lambda: None)
(8)                     cb2 = pickle.loads(pickle.dumps(cb))
(8)                     cid = cb2.connect('c', lambda: None)
(8)                     assert cid == 1
(4)                 @pytest.mark.parametrize('pickle', [True, False])
(4)                 def test_callback_complete(self, pickle):
(8)                     self.is_empty()
(8)                     mini_me = Test_callback_registry()
(8)                     cid1 = self.connect(self.signal, mini_me.dummy, pickle)
(8)                     assert type(cid1) is int
(8)                     self.is_not_empty()
(8)                     cid2 = self.connect(self.signal, mini_me.dummy, pickle)
(8)                     assert cid1 == cid2
(8)                     self.is_not_empty()
(8)                     assert len(self.callbacks._func_cid_map) == 1
(8)                     assert len(self.callbacks.callbacks) == 1
(8)                     del mini_me
(8)                     self.is_empty()
(4)                 @pytest.mark.parametrize('pickle', [True, False])
(4)                 def test_callback_disconnect(self, pickle):
(8)                     self.is_empty()
(8)                     mini_me = Test_callback_registry()
(8)                     cid1 = self.connect(self.signal, mini_me.dummy, pickle)
(8)                     assert type(cid1) is int
(8)                     self.is_not_empty()
(8)                     self.disconnect(cid1)
(8)                     self.is_empty()
(4)                 @pytest.mark.parametrize('pickle', [True, False])
(4)                 def test_callback_wrong_disconnect(self, pickle):
(8)                     self.is_empty()
(8)                     mini_me = Test_callback_registry()
(8)                     cid1 = self.connect(self.signal, mini_me.dummy, pickle)
(8)                     assert type(cid1) is int
(8)                     self.is_not_empty()
(8)                     self.disconnect("foo")
(8)                     self.is_not_empty()
(4)                 @pytest.mark.parametrize('pickle', [True, False])
(4)                 def test_registration_on_non_empty_registry(self, pickle):
(8)                     self.is_empty()
(8)                     mini_me = Test_callback_registry()
(8)                     self.connect(self.signal, mini_me.dummy, pickle)
(8)                     mini_me2 = Test_callback_registry()
```

```
(8)                 self.connect(self.signal, mini_me2.dummy, pickle)
(8)                 mini_me2 = Test_callback_registry()
(8)                 self.connect(self.signal, mini_me2.dummy, pickle)
(8)                 self.is_not_empty()
(8)                 assert self.count() == 2
(8)                 mini_me = None
(8)                 mini_me2 = None
(8)                 self.is_empty()
(4)             def dummy(self):
(8)                 pass
(4)             def test_pickling(self):
(8)                 assert hasattr(pickle.loads(pickle.dumps(cbook.CallbackRegistry())),
(23)                               "callbacks")
(0)         def test_callbackregistry_default_exception_handler(capsys, monkeypatch):
(4)             cb = cbook.CallbackRegistry()
(4)             cb.connect("foo", lambda: None)
(4)             monkeypatch.setattr(
(8)                 cbook, "_get_running_interactive_framework", lambda: None)
(4)             with pytest.raises(TypeError):
(8)                 cb.process("foo", "argument mismatch")
(4)             outerr = capsys.readouterr()
(4)             assert outerr.out == outerr.err == ""
(4)             monkeypatch.setattr(
(8)                 cbook, "_get_running_interactive_framework", lambda: "not-none")
(4)             cb.process("foo", "argument mismatch")  # No error in that case.
(4)             outerr = capsys.readouterr()
(4)             assert outerr.out == ""
(4)             assert "takes 0 positional arguments but 1 was given" in outerr.err
(0)         def raising_cb_reg(func):
(4)             class TestException(Exception):
(8)                 pass
(4)             def raise_runtime_error():
(8)                 raise RuntimeError
(4)             def raise_value_error():
(8)                 raise ValueError
(4)             def transformer(excp):
(8)                 if isinstance(excp, RuntimeError):
(12)                    raise TestException
(8)                 raise excp
(4)             cb_old = cbook.CallbackRegistry(exception_handler=None)
(4)             cb_old.connect('foo', raise_runtime_error)
(4)             cb_filt = cbook.CallbackRegistry(exception_handler=transformer)
(4)             cb_filt.connect('foo', raise_runtime_error)
(4)             cb_filt_pass = cbook.CallbackRegistry(exception_handler=transformer)
(4)             cb_filt_pass.connect('foo', raise_value_error)
(4)             return pytest.mark.parametrize('cb, excp',
(35)                                           [[cb_old, RuntimeError],
(36)                                            [cb_filt, TestException],
(36)                                            [cb_filt_pass, ValueError]])(func)
(0)         @raising_cb_reg
(0)         def test_callbackregistry_custom_exception_handler(monkeypatch, cb, excp):
(4)             monkeypatch.setattr(
(8)                 cbook, "_get_running_interactive_framework", lambda: None)
(4)             with pytest.raises(excp):
(8)                 cb.process('foo')
(0)         def test_callbackregistry_signals():
(4)             cr = cbook.CallbackRegistry(signals=["foo"])
(4)             results = []
(4)             def cb(x): results.append(x)
(4)             cr.connect("foo", cb)
(4)             with pytest.raises(ValueError):
(8)                 cr.connect("bar", cb)
(4)             cr.process("foo", 1)
(4)             with pytest.raises(ValueError):
(8)                 cr.process("bar", 1)
(4)             assert results == [1]
(0)         def test_callbackregistry_blocking():
(4)             def raise_handler(excp):
(8)                 raise excp
```

```
(4)     cb = cbook.CallbackRegistry(exception_handler=raise_handler)
(4)     def test_func1():
(8)         raise ValueError("1 should be blocked")
(4)     def test_func2():
(8)         raise ValueError("2 should be blocked")
(4)     cb.connect("test1", test_func1)
(4)     cb.connect("test2", test_func2)
(4)     with cb.blocked():
(8)         cb.process("test1")
(8)         cb.process("test2")
(4)     with cb.blocked(signal="test1"):
(8)         cb.process("test1")
(8)         with pytest.raises(ValueError, match="2 should be blocked"):
(12)            cb.process("test2")
(4)     with pytest.raises(ValueError, match="1 should be blocked"):
(8)         cb.process("test1")
(4)     with pytest.raises(ValueError, match="2 should be blocked"):
(8)         cb.process("test2")
(0)  @pytest.mark.parametrize('line, result', [
(4)     ('a : no_comment', 'a : no_comment'),
(4)     ('a : "quoted str"', 'a : "quoted str"'),
(4)     ('a : "quoted str" # comment', 'a : "quoted str"'),
(4)     ('a : "#000000"', 'a : "#000000"'),
(4)     ('a : "#000000" # comment', 'a : "#000000"'),
(4)     ('a : ["#000000", "#FFFFFF"]', 'a : ["#000000", "#FFFFFF"]'),
(4)     ('a : ["#000000", "#FFFFFF"] # comment', 'a : ["#000000", "#FFFFFF"]'),
(4)     ('a : val  # a comment "with quotes"', 'a : val'),
(4)     ('# only comment "with quotes" xx', ''),
(0)  ])
(0)  def test_strip_comment(line, result):
(4)     """Strip everything from the first unquoted #."""
(4)     assert cbook._strip_comment(line) == result
(0)  def test_strip_comment_invalid():
(4)     with pytest.raises(ValueError, match="Missing closing quote"):
(8)         cbook._strip_comment('grid.color: "aa')
(0)  def test_sanitize_sequence():
(4)     d = {'a': 1, 'b': 2, 'c': 3}
(4)     k = ['a', 'b', 'c']
(4)     v = [1, 2, 3]
(4)     i = [('a', 1), ('b', 2), ('c', 3)]
(4)     assert k == sorted(cbook.sanitize_sequence(d.keys()))
(4)     assert v == sorted(cbook.sanitize_sequence(d.values()))
(4)     assert i == sorted(cbook.sanitize_sequence(d.items()))
(4)     assert i == cbook.sanitize_sequence(i)
(4)     assert k == cbook.sanitize_sequence(k)
(0)  fail_mapping: tuple[tuple[dict, dict], ...] = (
(4)     ({'a': 1, 'b': 2}, {'alias_mapping': {'a': ['b']}}),
(4)     ({'a': 1, 'b': 2}, {'alias_mapping': {'a': ['a', 'b']}}),
(0)  )
(0)  pass_mapping: tuple[tuple[Any, dict, dict], ...] = (
(4)     (None, {}, {}),
(4)     ({'a': 1, 'b': 2}, {'a': 1, 'b': 2}, {}),
(4)     ({'b': 2}, {'a': 2}, {'alias_mapping': {'a': ['a', 'b']}}),
(0)  )
(0)  @pytest.mark.parametrize('inp, kwargs_to_norm', fail_mapping)
(0)  def test_normalize_kwargs_fail(inp, kwargs_to_norm):
(4)     with pytest.raises(TypeError), \
(9)          _api.suppress_matplotlib_deprecation_warning():
(8)         cbook.normalize_kwargs(inp, **kwargs_to_norm)
(0)  @pytest.mark.parametrize('inp, expected, kwargs_to_norm',
(25)                                pass_mapping)
(0)  def test_normalize_kwargs_pass(inp, expected, kwargs_to_norm):
(4)     with _api.suppress_matplotlib_deprecation_warning():
(8)         assert expected == cbook.normalize_kwargs(inp, **kwargs_to_norm)
(0)  def test_warn_external_frame_embedded_python():
(4)     with patch.object(cbook, "sys") as mock_sys:
(8)         mock_sys._getframe = Mock(return_value=None)
(8)         with pytest.warns(UserWarning, match=r"\Adummy\Z"):
(12)            _api.warn_external("dummy")
```

```
(0)             def test_to_prestep():
(4)                 x = np.arange(4)
(4)                 y1 = np.arange(4)
(4)                 y2 = np.arange(4)[::-1]
(4)                 xs, y1s, y2s = cbook.pts_to_prestep(x, y1, y2)
(4)                 x_target = np.asarray([0, 0, 1, 1, 2, 2, 3], dtype=float)
(4)                 y1_target = np.asarray([0, 1, 1, 2, 2, 3, 3], dtype=float)
(4)                 y2_target = np.asarray([3, 2, 2, 1, 1, 0, 0], dtype=float)
(4)                 assert_array_equal(x_target, xs)
(4)                 assert_array_equal(y1_target, y1s)
(4)                 assert_array_equal(y2_target, y2s)
(4)                 xs, y1s = cbook.pts_to_prestep(x, y1)
(4)                 assert_array_equal(x_target, xs)
(4)                 assert_array_equal(y1_target, y1s)
(0)             def test_to_prestep_empty():
(4)                 steps = cbook.pts_to_prestep([], [])
(4)                 assert steps.shape == (2, 0)
(0)             def test_to_poststep():
(4)                 x = np.arange(4)
(4)                 y1 = np.arange(4)
(4)                 y2 = np.arange(4)[::-1]
(4)                 xs, y1s, y2s = cbook.pts_to_poststep(x, y1, y2)
(4)                 x_target = np.asarray([0, 1, 1, 2, 2, 3, 3], dtype=float)
(4)                 y1_target = np.asarray([0, 0, 1, 1, 2, 2, 3], dtype=float)
(4)                 y2_target = np.asarray([3, 3, 2, 2, 1, 1, 0], dtype=float)
(4)                 assert_array_equal(x_target, xs)
(4)                 assert_array_equal(y1_target, y1s)
(4)                 assert_array_equal(y2_target, y2s)
(4)                 xs, y1s = cbook.pts_to_poststep(x, y1)
(4)                 assert_array_equal(x_target, xs)
(4)                 assert_array_equal(y1_target, y1s)
(0)             def test_to_poststep_empty():
(4)                 steps = cbook.pts_to_poststep([], [])
(4)                 assert steps.shape == (2, 0)
(0)             def test_to_midstep():
(4)                 x = np.arange(4)
(4)                 y1 = np.arange(4)
(4)                 y2 = np.arange(4)[::-1]
(4)                 xs, y1s, y2s = cbook.pts_to_midstep(x, y1, y2)
(4)                 x_target = np.asarray([0, .5, .5, 1.5, 1.5, 2.5, 2.5, 3], dtype=float)
(4)                 y1_target = np.asarray([0, 0, 1, 1, 2, 2, 3, 3], dtype=float)
(4)                 y2_target = np.asarray([3, 3, 2, 2, 1, 1, 0, 0], dtype=float)
(4)                 assert_array_equal(x_target, xs)
(4)                 assert_array_equal(y1_target, y1s)
(4)                 assert_array_equal(y2_target, y2s)
(4)                 xs, y1s = cbook.pts_to_midstep(x, y1)
(4)                 assert_array_equal(x_target, xs)
(4)                 assert_array_equal(y1_target, y1s)
(0)             def test_to_midstep_empty():
(4)                 steps = cbook.pts_to_midstep([], [])
(4)                 assert steps.shape == (2, 0)
(0)             @pytest.mark.parametrize(
(4)                 "args",
(4)                 [(np.arange(12).reshape(3, 4), 'a'),
(5)                  (np.arange(12), 'a'),
(5)                  (np.arange(12), np.arange(3))])
(0)             def test_step_fails(args):
(4)                 with pytest.raises(ValueError):
(8)                     cbook.pts_to_prestep(*args)
(0)             def test_grouper():
(4)                 class Dummy:
(8)                     pass
(4)                 a, b, c, d, e = objs = [Dummy() for _ in range(5)]
(4)                 g = cbook.Grouper()
(4)                 g.join(*objs)
(4)                 assert set(list(g)[0]) == set(objs)
(4)                 assert set(g.get_siblings(a)) == set(objs)
(4)                 for other in objs[1:]:
(8)                     assert g.joined(a, other)
```

```
(4)                 g.remove(a)
(4)                 for other in objs[1:]:
(8)                     assert not g.joined(a, other)
(4)                 for A, B in itertools.product(objs[1:], objs[1:]):
(8)                     assert g.joined(A, B)
(0)             def test_grouper_private():
(4)                 class Dummy:
(8)                     pass
(4)                 objs = [Dummy() for _ in range(5)]
(4)                 g = cbook.Grouper()
(4)                 g.join(*objs)
(4)                 mapping = g._mapping
(4)                 for o in objs:
(8)                     assert o in mapping
(4)                 base_set = mapping[objs[0]]
(4)                 for o in objs[1:]:
(8)                     assert mapping[o] is base_set
(0)             def test_flatiter():
(4)                 x = np.arange(5)
(4)                 it = x.flat
(4)                 assert 0 == next(it)
(4)                 assert 1 == next(it)
(4)                 ret = cbook._safe_first_finite(it)
(4)                 assert ret == 0
(4)                 assert 0 == next(it)
(4)                 assert 1 == next(it)
(0)             def test__safe_first_finite_all_nan():
(4)                 arr = np.full(2, np.nan)
(4)                 ret = cbook._safe_first_finite(arr)
(4)                 assert np.isnan(ret)
(0)             def test__safe_first_finite_all_inf():
(4)                 arr = np.full(2, np.inf)
(4)                 ret = cbook._safe_first_finite(arr)
(4)                 assert np.isinf(ret)
(0)             def test_reshape2d():
(4)                 class Dummy:
(8)                     pass
(4)                 xnew = cbook._reshape_2D([], 'x')
(4)                 assert np.shape(xnew) == (1, 0)
(4)                 x = [Dummy() for _ in range(5)]
(4)                 xnew = cbook._reshape_2D(x, 'x')
(4)                 assert np.shape(xnew) == (1, 5)
(4)                 x = np.arange(5)
(4)                 xnew = cbook._reshape_2D(x, 'x')
(4)                 assert np.shape(xnew) == (1, 5)
(4)                 x = [[Dummy() for _ in range(5)] for _ in range(3)]
(4)                 xnew = cbook._reshape_2D(x, 'x')
(4)                 assert np.shape(xnew) == (3, 5)
(4)                 x = np.random.rand(3, 5)
(4)                 xnew = cbook._reshape_2D(x, 'x')
(4)                 assert np.shape(xnew) == (5, 3)
(4)                 x = [[1], [2], [3]]
(4)                 xnew = cbook._reshape_2D(x, 'x')
(4)                 assert isinstance(xnew, list)
(4)                 assert isinstance(xnew[0], np.ndarray) and xnew[0].shape == (1,)
(4)                 assert isinstance(xnew[1], np.ndarray) and xnew[1].shape == (1,)
(4)                 assert isinstance(xnew[2], np.ndarray) and xnew[2].shape == (1,)
(4)                 x = [np.array(0), np.array(1), np.array(2)]
(4)                 xnew = cbook._reshape_2D(x, 'x')
(4)                 assert isinstance(xnew, list)
(4)                 assert len(xnew) == 1
(4)                 assert isinstance(xnew[0], np.ndarray) and xnew[0].shape == (3,)
(4)                 x = [[1, 2, 3], [3, 4], [2]]
(4)                 xnew = cbook._reshape_2D(x, 'x')
(4)                 assert isinstance(xnew, list)
(4)                 assert isinstance(xnew[0], np.ndarray) and xnew[0].shape == (3,)
(4)                 assert isinstance(xnew[1], np.ndarray) and xnew[1].shape == (2,)
(4)                 assert isinstance(xnew[2], np.ndarray) and xnew[2].shape == (1,)
(4)                 class ArraySubclass(np.ndarray):
```

```
(8)                     def __iter__(self):
(12)                        for value in super().__iter__():
(16)                            yield np.array(value)
(8)                     def __getitem__(self, item):
(12)                        return np.array(super().__getitem__(item))
(4)                 v = np.arange(10, dtype=float)
(4)                 x = ArraySubclass((10,), dtype=float, buffer=v.data)
(4)                 xnew = cbook._reshape_2D(x, 'x')
(4)                 assert len(xnew) == 1
(4)                 assert isinstance(xnew[0], ArraySubclass)
(4)                 x = ['a', 'b', 'c', 'c', 'dd', 'e', 'f', 'ff', 'f']
(4)                 xnew = cbook._reshape_2D(x, 'x')
(4)                 assert len(xnew[0]) == len(x)
(4)                 assert isinstance(xnew[0], np.ndarray)
(0)             def test_reshape2d_pandas(pd):
(4)                 X = np.arange(30).reshape(10, 3)
(4)                 x = pd.DataFrame(X, columns=["a", "b", "c"])
(4)                 Xnew = cbook._reshape_2D(x, 'x')
(4)                 for x, xnew in zip(X.T, Xnew):
(8)                     np.testing.assert_array_equal(x, xnew)
(0)             def test_reshape2d_xarray(xr):
(4)                 X = np.arange(30).reshape(10, 3)
(4)                 x = xr.DataArray(X, dims=["x", "y"])
(4)                 Xnew = cbook._reshape_2D(x, 'x')
(4)                 for x, xnew in zip(X.T, Xnew):
(8)                     np.testing.assert_array_equal(x, xnew)
(0)             def test_index_of_pandas(pd):
(4)                 X = np.arange(30).reshape(10, 3)
(4)                 x = pd.DataFrame(X, columns=["a", "b", "c"])
(4)                 Idx, Xnew = cbook.index_of(x)
(4)                 np.testing.assert_array_equal(X, Xnew)
(4)                 IdxRef = np.arange(10)
(4)                 np.testing.assert_array_equal(Idx, IdxRef)
(0)             def test_index_of_xarray(xr):
(4)                 X = np.arange(30).reshape(10, 3)
(4)                 x = xr.DataArray(X, dims=["x", "y"])
(4)                 Idx, Xnew = cbook.index_of(x)
(4)                 np.testing.assert_array_equal(X, Xnew)
(4)                 IdxRef = np.arange(10)
(4)                 np.testing.assert_array_equal(Idx, IdxRef)
(0)             def test_contiguous_regions():
(4)                 a, b, c = 3, 4, 5
(4)                 mask = [True]*a + [False]*b + [True]*c
(4)                 expected = [(0, a), (a+b, a+b+c)]
(4)                 assert cbook.contiguous_regions(mask) == expected
(4)                 d, e = 6, 7
(4)                 mask = mask + [False]*e
(4)                 assert cbook.contiguous_regions(mask) == expected
(4)                 mask = [False]*d + mask[:-e]
(4)                 expected = [(d, d+a), (d+a+b, d+a+b+c)]
(4)                 assert cbook.contiguous_regions(mask) == expected
(4)                 mask = mask + [False]*e
(4)                 assert cbook.contiguous_regions(mask) == expected
(4)                 assert cbook.contiguous_regions([False]*5) == []
(4)                 assert cbook.contiguous_regions([]) == []
(0)             def test_safe_first_element_pandas_series(pd):
(4)                 s = pd.Series(range(5), index=range(10, 15))
(4)                 actual = cbook._safe_first_finite(s)
(4)                 assert actual == 0
(0)             def test_warn_external(recwarn):
(4)                 _api.warn_external("oops")
(4)                 assert len(recwarn) == 1
(4)                 assert recwarn[0].filename == __file__
(0)             def test_array_patch_perimeters():
(4)                 def check(x, rstride, cstride):
(8)                     rows, cols = x.shape
(8)                     row_inds = [*range(0, rows-1, rstride), rows-1]
(8)                     col_inds = [*range(0, cols-1, cstride), cols-1]
(8)                     polys = []
```

```
(8)                     for rs, rs_next in zip(row_inds[:-1], row_inds[1:]):
(12)                        for cs, cs_next in zip(col_inds[:-1], col_inds[1:]):
(16)                            ps = cbook._array_perimeter(x[rs:rs_next+1, cs:cs_next+1]).T
(16)                            polys.append(ps)
(8)                     polys = np.asarray(polys)
(8)                     assert np.array_equal(polys,
(30)                                          cbook._array_patch_perimeters(
(34)                                              x, rstride=rstride, cstride=cstride))
(4)                 def divisors(n):
(8)                     return [i for i in range(1, n + 1) if n % i == 0]
(4)                 for rows, cols in [(5, 5), (7, 14), (13, 9)]:
(8)                     x = np.arange(rows * cols).reshape(rows, cols)
(8)                     for rstride, cstride in itertools.product(divisors(rows - 1),
(50)                                                              divisors(cols - 1)):
(12)                        check(x, rstride=rstride, cstride=cstride)
(0)             def test_setattr_cm():
(4)                 class A:
(8)                     cls_level = object()
(8)                     override = object()
(8)                     def __init__(self):
(12)                        self.aardvark = 'aardvark'
(12)                        self.override = 'override'
(12)                        self._p = 'p'
(8)                     def meth(self):
(12)                        ...
(8)                     @classmethod
(8)                     def classy(cls):
(12)                        ...
(8)                     @staticmethod
(8)                     def static():
(12)                        ...
(8)                     @property
(8)                     def prop(self):
(12)                        return self._p
(8)                     @prop.setter
(8)                     def prop(self, val):
(12)                        self._p = val
(4)                 class B(A):
(8)                     ...
(4)                 other = A()
(4)                 def verify_pre_post_state(obj):
(8)                     assert obj.meth is not obj.meth
(8)                     assert obj.aardvark is obj.aardvark
(8)                     assert a.aardvark == 'aardvark'
(8)                     assert obj.prop is obj.prop
(8)                     assert obj.cls_level is A.cls_level
(8)                     assert obj.override == 'override'
(8)                     assert not hasattr(obj, 'extra')
(8)                     assert obj.prop == 'p'
(8)                     assert obj.monkey == other.meth
(8)                     assert obj.cls_level is A.cls_level
(8)                     assert 'cls_level' not in obj.__dict__
(8)                     assert 'classy' not in obj.__dict__
(8)                     assert 'static' not in obj.__dict__
(4)                 a = B()
(4)                 a.monkey = other.meth
(4)                 verify_pre_post_state(a)
(4)                 with cbook._setattr_cm(
(12)                        a, prop='squirrel',
(12)                        aardvark='moose', meth=lambda: None,
(12)                        override='boo', extra='extra',
(12)                        monkey=lambda: None, cls_level='bob',
(12)                        classy='classy', static='static'):
(8)                     assert a.meth is a.meth
(8)                     assert a.aardvark is a.aardvark
(8)                     assert a.aardvark == 'moose'
(8)                     assert a.override == 'boo'
(8)                     assert a.extra == 'extra'
(8)                     assert a.prop == 'squirrel'
```

```
(8)                 assert a.monkey != other.meth
(8)                 assert a.cls_level == 'bob'
(8)                 assert a.classy == 'classy'
(8)                 assert a.static == 'static'
(4)             verify_pre_post_state(a)
(0)         def test_format_approx():
(4)             f = cbook._format_approx
(4)             assert f(0, 1) == '0'
(4)             assert f(0, 2) == '0'
(4)             assert f(0, 3) == '0'
(4)             assert f(-0.0123, 1) == '-0'
(4)             assert f(1e-7, 5) == '0'
(4)             assert f(0.0012345600001, 5) == '0.00123'
(4)             assert f(-0.0012345600001, 5) == '-0.00123'
(4)             assert f(0.0012345600001, 8) == f(0.0012345600001, 10) == '0.00123456'
(0)         def test_safe_first_element_with_none():
(4)             datetime_lst = [date.today() + timedelta(days=i) for i in range(10)]
(4)             datetime_lst[0] = None
(4)             actual = cbook._safe_first_finite(datetime_lst)
(4)             assert actual is not None and actual == datetime_lst[1]
(0)         def test_strip_math():
(4)             assert strip_math(r'1 \times 2') == r'1 \times 2'
(4)             assert strip_math(r'$1 \times 2$') == '1 x 2'
(4)             assert strip_math(r'$\rm{hi}$') == 'hi'
(0)         @pytest.mark.parametrize('fmt, value, result', [
(4)             ('%.2f m', 0.2, '0.20 m'),
(4)             ('{:.2f} m', 0.2, '0.20 m'),
(4)             ('{} m', 0.2, '0.2 m'),
(4)             ('const', 0.2, 'const'),
(4)             ('%d or {}', 0.2, '0 or {}'),
(4)             ('{{{:,.0f}}}', 2e5, '{200,000}'),
(4)             ('{:.2%}', 2/3, '66.67%'),
(4)             ('$%g', 2.54, '$2.54'),
(0)         ])
(0)         def test_auto_format_str(fmt, value, result):
(4)             """Apply *value* to the format string *fmt*."""
(4)             assert cbook._auto_format_str(fmt, value) == result
(4)             assert cbook._auto_format_str(fmt, np.float64(value)) == result
(0)         def test_unpack_to_numpy_from_torch():
(4)             """
(4)             Test that torch tensors are converted to NumPy arrays.
(4)             We don't want to create a dependency on torch in the test suite, so we
mock it.
(4)             """
(4)             class Tensor:
(8)                 def __init__(self, data):
(12)                    self.data = data
(8)                 def __array__(self):
(12)                    return self.data
(4)             torch = ModuleType('torch')
(4)             torch.Tensor = Tensor
(4)             sys.modules['torch'] = torch
(4)             data = np.arange(10)
(4)             torch_tensor = torch.Tensor(data)
(4)             result = cbook._unpack_to_numpy(torch_tensor)
(4)             assert result is torch_tensor.__array__()
(0)         def test_unpack_to_numpy_from_jax():
(4)             """
(4)             Test that jax arrays are converted to NumPy arrays.
(4)             We don't want to create a dependency on jax in the test suite, so we mock
it.
(4)             """
(4)             class Array:
(8)                 def __init__(self, data):
(12)                    self.data = data
(8)                 def __array__(self):
(12)                    return self.data
(4)             jax = ModuleType('jax')
(4)             jax.Array = Array
```

```
698: (4)                sys.modules['jax'] = jax
699: (4)                data = np.arange(10)
700: (4)                jax_array = jax.Array(data)
701: (4)                result = cbook._unpack_to_numpy(jax_array)
702: (4)                assert result is jax_array.__array__()
```

File 171 - test_collections.py:

```
1: (0)              from datetime import datetime
2: (0)              import io
3: (0)              import itertools
4: (0)              import platform
5: (0)              import re
6: (0)              from types import SimpleNamespace
7: (0)              import numpy as np
8: (0)              from numpy.testing import assert_array_equal, assert_array_almost_equal
9: (0)              import pytest
10: (0)             import matplotlib as mpl
11: (0)             import matplotlib.pyplot as plt
12: (0)             import matplotlib.collections as mcollections
13: (0)             import matplotlib.colors as mcolors
14: (0)             import matplotlib.path as mpath
15: (0)             import matplotlib.transforms as mtransforms
16: (0)             from matplotlib.collections import (Collection, LineCollection,
17: (36)                                                EventCollection, PolyCollection)
18: (0)             from matplotlib.testing.decorators import check_figures_equal,
image_comparison
19: (0)             @pytest.fixture(params=["pcolormesh", "pcolor"])
20: (0)             def pcfunc(request):
21: (4)                 return request.param
22: (0)             def generate_EventCollection_plot():
23: (4)                 """Generate the initial collection and plot it."""
24: (4)                 positions = np.array([0., 1., 2., 3., 5., 8., 13., 21.])
25: (4)                 extra_positions = np.array([34., 55., 89.])
26: (4)                 orientation = 'horizontal'
27: (4)                 lineoffset = 1
28: (4)                 linelength = .5
29: (4)                 linewidth = 2
30: (4)                 color = [1, 0, 0, 1]
31: (4)                 linestyle = 'solid'
32: (4)                 antialiased = True
33: (4)                 coll = EventCollection(positions,
34: (27)                                       orientation=orientation,
35: (27)                                       lineoffset=lineoffset,
36: (27)                                       linelength=linelength,
37: (27)                                       linewidth=linewidth,
38: (27)                                       color=color,
39: (27)                                       linestyle=linestyle,
40: (27)                                       antialiased=antialiased
41: (27)                                       )
42: (4)                 fig, ax = plt.subplots()
43: (4)                 ax.add_collection(coll)
44: (4)                 ax.set_title('EventCollection: default')
45: (4)                 props = {'positions': positions,
46: (13)                         'extra_positions': extra_positions,
47: (13)                         'orientation': orientation,
48: (13)                         'lineoffset': lineoffset,
49: (13)                         'linelength': linelength,
50: (13)                         'linewidth': linewidth,
51: (13)                         'color': color,
52: (13)                         'linestyle': linestyle,
53: (13)                         'antialiased': antialiased
54: (13)                         }
55: (4)                 ax.set_xlim(-1, 22)
56: (4)                 ax.set_ylim(0, 2)
57: (4)                 return ax, coll, props
58: (0)             @image_comparison(['EventCollection_plot__default'])
```

```
(0)             def test__EventCollection__get_props():
(4)                 _, coll, props = generate_EventCollection_plot()
(4)                 check_segments(coll,
(19)                               props['positions'],
(19)                               props['linelength'],
(19)                               props['lineoffset'],
(19)                               props['orientation'])
(4)                 np.testing.assert_array_equal(props['positions'], coll.get_positions())
(4)                 assert props['orientation'] == coll.get_orientation()
(4)                 assert coll.is_horizontal()
(4)                 assert props['linelength'] == coll.get_linelength()
(4)                 assert props['lineoffset'] == coll.get_lineoffset()
(4)                 assert coll.get_linestyle() == [(0, None)]
(4)                 for color in [coll.get_color(), *coll.get_colors()]:
(8)                     np.testing.assert_array_equal(color, props['color'])
(0)             @image_comparison(['EventCollection_plot__set_positions'])
(0)             def test__EventCollection__set_positions():
(4)                 splt, coll, props = generate_EventCollection_plot()
(4)                 new_positions = np.hstack([props['positions'], props['extra_positions']])
(4)                 coll.set_positions(new_positions)
(4)                 np.testing.assert_array_equal(new_positions, coll.get_positions())
(4)                 check_segments(coll, new_positions,
(19)                               props['linelength'],
(19)                               props['lineoffset'],
(19)                               props['orientation'])
(4)                 splt.set_title('EventCollection: set_positions')
(4)                 splt.set_xlim(-1, 90)
(0)             @image_comparison(['EventCollection_plot__add_positions'])
(0)             def test__EventCollection__add_positions():
(4)                 splt, coll, props = generate_EventCollection_plot()
(4)                 new_positions = np.hstack([props['positions'],
(31)                                           props['extra_positions'][0]])
(4)                 coll.switch_orientation()  # Test adding in the vertical orientation, too.
(4)                 coll.add_positions(props['extra_positions'][0])
(4)                 coll.switch_orientation()
(4)                 np.testing.assert_array_equal(new_positions, coll.get_positions())
(4)                 check_segments(coll,
(19)                               new_positions,
(19)                               props['linelength'],
(19)                               props['lineoffset'],
(19)                               props['orientation'])
(4)                 splt.set_title('EventCollection: add_positions')
(4)                 splt.set_xlim(-1, 35)
(0)             @image_comparison(['EventCollection_plot__append_positions'])
(0)             def test__EventCollection__append_positions():
(4)                 splt, coll, props = generate_EventCollection_plot()
(4)                 new_positions = np.hstack([props['positions'],
(31)                                           props['extra_positions'][2]])
(4)                 coll.append_positions(props['extra_positions'][2])
(4)                 np.testing.assert_array_equal(new_positions, coll.get_positions())
(4)                 check_segments(coll,
(19)                               new_positions,
(19)                               props['linelength'],
(19)                               props['lineoffset'],
(19)                               props['orientation'])
(4)                 splt.set_title('EventCollection: append_positions')
(4)                 splt.set_xlim(-1, 90)
(0)             @image_comparison(['EventCollection_plot__extend_positions'])
(0)             def test__EventCollection__extend_positions():
(4)                 splt, coll, props = generate_EventCollection_plot()
(4)                 new_positions = np.hstack([props['positions'],
(31)                                           props['extra_positions'][1:]])
(4)                 coll.extend_positions(props['extra_positions'][1:])
(4)                 np.testing.assert_array_equal(new_positions, coll.get_positions())
(4)                 check_segments(coll,
(19)                               new_positions,
(19)                               props['linelength'],
(19)                               props['lineoffset'],
(19)                               props['orientation'])
```

```
(4)                 splt.set_title('EventCollection: extend_positions')
(4)                 splt.set_xlim(-1, 90)
(0)             @image_comparison(['EventCollection_plot__switch_orientation'])
(0)             def test__EventCollection__switch_orientation():
(4)                 splt, coll, props = generate_EventCollection_plot()
(4)                 new_orientation = 'vertical'
(4)                 coll.switch_orientation()
(4)                 assert new_orientation == coll.get_orientation()
(4)                 assert not coll.is_horizontal()
(4)                 new_positions = coll.get_positions()
(4)                 check_segments(coll,
(19)                               new_positions,
(19)                               props['linelength'],
(19)                               props['lineoffset'], new_orientation)
(4)                 splt.set_title('EventCollection: switch_orientation')
(4)                 splt.set_ylim(-1, 22)
(4)                 splt.set_xlim(0, 2)
(0)             @image_comparison(['EventCollection_plot__switch_orientation__2x'])
(0)             def test__EventCollection__switch_orientation_2x():
(4)                 """
(4)                 Check that calling switch_orientation twice sets the orientation back to
(4)                 the default.
(4)                 """
(4)                 splt, coll, props = generate_EventCollection_plot()
(4)                 coll.switch_orientation()
(4)                 coll.switch_orientation()
(4)                 new_positions = coll.get_positions()
(4)                 assert props['orientation'] == coll.get_orientation()
(4)                 assert coll.is_horizontal()
(4)                 np.testing.assert_array_equal(props['positions'], new_positions)
(4)                 check_segments(coll,
(19)                               new_positions,
(19)                               props['linelength'],
(19)                               props['lineoffset'],
(19)                               props['orientation'])
(4)                 splt.set_title('EventCollection: switch_orientation 2x')
(0)             @image_comparison(['EventCollection_plot__set_orientation'])
(0)             def test__EventCollection__set_orientation():
(4)                 splt, coll, props = generate_EventCollection_plot()
(4)                 new_orientation = 'vertical'
(4)                 coll.set_orientation(new_orientation)
(4)                 assert new_orientation == coll.get_orientation()
(4)                 assert not coll.is_horizontal()
(4)                 check_segments(coll,
(19)                               props['positions'],
(19)                               props['linelength'],
(19)                               props['lineoffset'],
(19)                               new_orientation)
(4)                 splt.set_title('EventCollection: set_orientation')
(4)                 splt.set_ylim(-1, 22)
(4)                 splt.set_xlim(0, 2)
(0)             @image_comparison(['EventCollection_plot__set_linelength'])
(0)             def test__EventCollection__set_linelength():
(4)                 splt, coll, props = generate_EventCollection_plot()
(4)                 new_linelength = 15
(4)                 coll.set_linelength(new_linelength)
(4)                 assert new_linelength == coll.get_linelength()
(4)                 check_segments(coll,
(19)                               props['positions'],
(19)                               new_linelength,
(19)                               props['lineoffset'],
(19)                               props['orientation'])
(4)                 splt.set_title('EventCollection: set_linelength')
(4)                 splt.set_ylim(-20, 20)
(0)             @image_comparison(['EventCollection_plot__set_lineoffset'])
(0)             def test__EventCollection__set_lineoffset():
(4)                 splt, coll, props = generate_EventCollection_plot()
(4)                 new_lineoffset = -5.
(4)                 coll.set_lineoffset(new_lineoffset)
```

```
197: (4)                assert new_lineoffset == coll.get_lineoffset()
198: (4)                check_segments(coll,
199: (19)                               props['positions'],
200: (19)                               props['linelength'],
201: (19)                               new_lineoffset,
202: (19)                               props['orientation'])
203: (4)                splt.set_title('EventCollection: set_lineoffset')
204: (4)                splt.set_ylim(-6, -4)
205: (0)            @image_comparison([
206: (4)                'EventCollection_plot__set_linestyle',
207: (4)                'EventCollection_plot__set_linestyle',
208: (4)                'EventCollection_plot__set_linewidth',
209: (0)            ])
210: (0)            def test__EventCollection__set_prop():
211: (4)                for prop, value, expected in [
212: (12)                       ('linestyle', 'dashed', [(0, (6.0, 6.0))]),
213: (12)                       ('linestyle', (0, (6., 6.)), [(0, (6.0, 6.0))]),
214: (12)                       ('linewidth', 5, 5),
215: (4)                ]:
216: (8)                    splt, coll, _ = generate_EventCollection_plot()
217: (8)                    coll.set(**{prop: value})
218: (8)                    assert plt.getp(coll, prop) == expected
219: (8)                    splt.set_title(f'EventCollection: set_{prop}')
220: (0)            @image_comparison(['EventCollection_plot__set_color'])
221: (0)            def test__EventCollection__set_color():
222: (4)                splt, coll, _ = generate_EventCollection_plot()
223: (4)                new_color = np.array([0, 1, 1, 1])
224: (4)                coll.set_color(new_color)
225: (4)                for color in [coll.get_color(), *coll.get_colors()]:
226: (8)                    np.testing.assert_array_equal(color, new_color)
227: (4)                splt.set_title('EventCollection: set_color')
228: (0)            def check_segments(coll, positions, linelength, lineoffset, orientation):
229: (4)                """
230: (4)                Test helper checking that all values in the segment are correct, given a
231: (4)                particular set of inputs.
232: (4)                """
233: (4)                segments = coll.get_segments()
234: (4)                if (orientation.lower() == 'horizontal'
235: (12)                       or orientation.lower() == 'none' or orientation is None):
236: (8)                    pos1 = 1
237: (8)                    pos2 = 0
238: (4)                elif orientation.lower() == 'vertical':
239: (8)                    pos1 = 0
240: (8)                    pos2 = 1
241: (4)                else:
242: (8)                    raise ValueError("orientation must be 'horizontal' or 'vertical'")
243: (4)                for i, segment in enumerate(segments):
244: (8)                    assert segment[0, pos1] == lineoffset + linelength / 2
245: (8)                    assert segment[1, pos1] == lineoffset - linelength / 2
246: (8)                    assert segment[0, pos2] == positions[i]
247: (8)                    assert segment[1, pos2] == positions[i]
248: (0)            def test_collection_norm_autoscale():
249: (4)                lines = np.arange(24).reshape((4, 3, 2))
250: (4)                coll = mcollections.LineCollection(lines, array=np.arange(4))
251: (4)                assert coll.norm(2) == 2 / 3
252: (4)                coll.set_array(np.arange(4) + 5)
253: (4)                assert coll.norm(2) == 2 / 3
254: (0)            def test_null_collection_datalim():
255: (4)                col = mcollections.PathCollection([])
256: (4)                col_data_lim = col.get_datalim(mtransforms.IdentityTransform())
257: (4)                assert_array_equal(col_data_lim.get_points(),
258: (23)                                  mtransforms.Bbox.null().get_points())
259: (0)            def test_no_offsets_datalim():
260: (4)                ax = plt.axes()
261: (4)                coll = mcollections.PathCollection([mpath.Path([(0, 0), (1, 0)])])
262: (4)                ax.add_collection(coll)
263: (4)                coll_data_lim = coll.get_datalim(mtransforms.IdentityTransform())
264: (4)                assert_array_equal(coll_data_lim.get_points(),
265: (23)                                  mtransforms.Bbox.null().get_points())
```

```
(0)             def test_add_collection():
(4)                 plt.figure()
(4)                 ax = plt.axes()
(4)                 ax.scatter([0, 1], [0, 1])
(4)                 bounds = ax.dataLim.bounds
(4)                 ax.scatter([], [])
(4)                 assert ax.dataLim.bounds == bounds
(0)             @mpl.style.context('mpl20')
(0)             @check_figures_equal(extensions=['png'])
(0)             def test_collection_log_datalim(fig_test, fig_ref):
(4)                 x_vals = [4.38462e-6, 5.54929e-6, 7.02332e-6, 8.88889e-6, 1.12500e-5,
(14)                          1.42383e-5, 1.80203e-5, 2.28070e-5, 2.88651e-5, 3.65324e-5,
(14)                          4.62363e-5, 5.85178e-5, 7.40616e-5, 9.37342e-5, 1.18632e-4]
(4)                 y_vals = [0.0, 0.1, 0.182, 0.332, 0.604, 1.1, 2.0, 3.64, 6.64, 12.1, 22.0,
(14)                          39.6, 71.3]
(4)                 x, y = np.meshgrid(x_vals, y_vals)
(4)                 x = x.flatten()
(4)                 y = y.flatten()
(4)                 ax_test = fig_test.subplots()
(4)                 ax_test.set_xscale('log')
(4)                 ax_test.set_yscale('log')
(4)                 ax_test.margins = 0
(4)                 ax_test.scatter(x, y)
(4)                 ax_ref = fig_ref.subplots()
(4)                 ax_ref.set_xscale('log')
(4)                 ax_ref.set_yscale('log')
(4)                 ax_ref.plot(x, y, marker="o", ls="")
(0)             def test_quiver_limits():
(4)                 ax = plt.axes()
(4)                 x, y = np.arange(8), np.arange(10)
(4)                 u = v = np.linspace(0, 10, 80).reshape(10, 8)
(4)                 q = plt.quiver(x, y, u, v)
(4)                 assert q.get_datalim(ax.transData).bounds == (0., 0., 7., 9.)
(4)                 plt.figure()
(4)                 ax = plt.axes()
(4)                 x = np.linspace(-5, 10, 20)
(4)                 y = np.linspace(-2, 4, 10)
(4)                 y, x = np.meshgrid(y, x)
(4)                 trans = mtransforms.Affine2D().translate(25, 32) + ax.transData
(4)                 plt.quiver(x, y, np.sin(x), np.cos(y), transform=trans)
(4)                 assert ax.dataLim.bounds == (20.0, 30.0, 15.0, 6.0)
(0)             def test_barb_limits():
(4)                 ax = plt.axes()
(4)                 x = np.linspace(-5, 10, 20)
(4)                 y = np.linspace(-2, 4, 10)
(4)                 y, x = np.meshgrid(y, x)
(4)                 trans = mtransforms.Affine2D().translate(25, 32) + ax.transData
(4)                 plt.barbs(x, y, np.sin(x), np.cos(y), transform=trans)
(4)                 assert_array_almost_equal(ax.dataLim.bounds, (20, 30, 15, 6),
(30)                                          decimal=1)
(0)             @image_comparison(['EllipseCollection_test_image.png'], remove_text=True,
(18)                              tol=0.021 if platform.machine() == 'arm64' else 0)
(0)             def test_EllipseCollection():
(4)                 fig, ax = plt.subplots()
(4)                 x = np.arange(4)
(4)                 y = np.arange(3)
(4)                 X, Y = np.meshgrid(x, y)
(4)                 XY = np.vstack((X.ravel(), Y.ravel())).T
(4)                 ww = X / x[-1]
(4)                 hh = Y / y[-1]
(4)                 aa = np.ones_like(ww) * 20  # first axis is 20 degrees CCW from x axis
(4)                 ec = mcollections.EllipseCollection(
(8)                     ww, hh, aa, units='x', offsets=XY, offset_transform=ax.transData,
(8)                     facecolors='none')
(4)                 ax.add_collection(ec)
(4)                 ax.autoscale_view()
(0)             def test_EllipseCollection_setter_getter():
(4)                 rng = np.random.default_rng(0)
(4)                 widths = (2, )
```

```
335: (4)                heights = (3, )
336: (4)                angles = (45, )
337: (4)                offsets = rng.random((10, 2)) * 10
338: (4)                fig, ax = plt.subplots()
339: (4)                ec = mcollections.EllipseCollection(
340: (8)                    widths=widths,
341: (8)                    heights=heights,
342: (8)                    angles=angles,
343: (8)                    offsets=offsets,
344: (8)                    units='x',
345: (8)                    offset_transform=ax.transData,
346: (8)                    )
347: (4)                assert_array_almost_equal(ec._widths, np.array(widths).ravel() * 0.5)
348: (4)                assert_array_almost_equal(ec._heights, np.array(heights).ravel() * 0.5)
349: (4)                assert_array_almost_equal(ec._angles, np.deg2rad(angles).ravel())
350: (4)                assert_array_almost_equal(ec.get_widths(), widths)
351: (4)                assert_array_almost_equal(ec.get_heights(), heights)
352: (4)                assert_array_almost_equal(ec.get_angles(), angles)
353: (4)                ax.add_collection(ec)
354: (4)                ax.set_xlim(-2, 12)
355: (4)                ax.set_ylim(-2, 12)
356: (4)                new_widths = rng.random((10, 2)) * 2
357: (4)                new_heights = rng.random((10, 2)) * 3
358: (4)                new_angles = rng.random((10, 2)) * 180
359: (4)                ec.set(widths=new_widths, heights=new_heights, angles=new_angles)
360: (4)                assert_array_almost_equal(ec.get_widths(), new_widths.ravel())
361: (4)                assert_array_almost_equal(ec.get_heights(), new_heights.ravel())
362: (4)                assert_array_almost_equal(ec.get_angles(), new_angles.ravel())
363: (0)            @image_comparison(['polycollection_close.png'], remove_text=True,
style='mpl20')
364: (0)            def test_polycollection_close():
365: (4)                from mpl_toolkits.mplot3d import Axes3D  # type: ignore
366: (4)                plt.rcParams['axes3d.automargin'] = True
367: (4)                vertsQuad = [
368: (8)                    [[0., 0.], [0., 1.], [1., 1.], [1., 0.]],
369: (8)                    [[0., 1.], [2., 3.], [2., 2.], [1., 1.]],
370: (8)                    [[2., 2.], [2., 3.], [4., 1.], [3., 1.]],
371: (8)                    [[3., 0.], [3., 1.], [4., 1.], [4., 0.]]]
372: (4)                fig = plt.figure()
373: (4)                ax = fig.add_axes(Axes3D(fig))
374: (4)                colors = ['r', 'g', 'b', 'y', 'k']
375: (4)                zpos = list(range(5))
376: (4)                poly = mcollections.PolyCollection(
377: (8)                    vertsQuad * len(zpos), linewidth=0.25)
378: (4)                poly.set_alpha(0.7)
379: (4)                zs = []
380: (4)                cs = []
381: (4)                for z, c in zip(zpos, colors):
382: (8)                    zs.extend([z] * len(vertsQuad))
383: (8)                    cs.extend([c] * len(vertsQuad))
384: (4)                poly.set_color(cs)
385: (4)                ax.add_collection3d(poly, zs=zs, zdir='y')
386: (4)                ax.set_xlim3d(0, 4)
387: (4)                ax.set_zlim3d(0, 3)
388: (4)                ax.set_ylim3d(0, 4)
389: (0)            @image_comparison(['regularpolycollection_rotate.png'], remove_text=True)
390: (0)            def test_regularpolycollection_rotate():
391: (4)                xx, yy = np.mgrid[:10, :10]
392: (4)                xy_points = np.transpose([xx.flatten(), yy.flatten()])
393: (4)                rotations = np.linspace(0, 2*np.pi, len(xy_points))
394: (4)                fig, ax = plt.subplots()
395: (4)                for xy, alpha in zip(xy_points, rotations):
396: (8)                    col = mcollections.RegularPolyCollection(
397: (12)                       4, sizes=(100,), rotation=alpha,
398: (12)                       offsets=[xy], offset_transform=ax.transData)
399: (8)                    ax.add_collection(col, autolim=True)
400: (4)                ax.autoscale_view()
401: (0)            @image_comparison(['regularpolycollection_scale.png'], remove_text=True)
402: (0)            def test_regularpolycollection_scale():
```

```
(4)     class SquareCollection(mcollections.RegularPolyCollection):
(8)         def __init__(self, **kwargs):
(12)            super().__init__(4, rotation=np.pi/4., **kwargs)
(8)         def get_transform(self):
(12)            """Return transform scaling circle areas to data space."""
(12)            ax = self.axes
(12)            pts2pixels = 72.0 / ax.figure.dpi
(12)            scale_x = pts2pixels * ax.bbox.width / ax.viewLim.width
(12)            scale_y = pts2pixels * ax.bbox.height / ax.viewLim.height
(12)            return mtransforms.Affine2D().scale(scale_x, scale_y)
(4)     fig, ax = plt.subplots()
(4)     xy = [(0, 0)]
(4)     circle_areas = [np.pi / 2]
(4)     squares = SquareCollection(
(8)         sizes=circle_areas, offsets=xy, offset_transform=ax.transData)
(4)     ax.add_collection(squares, autolim=True)
(4)     ax.axis([-1, 1, -1, 1])
(0) def test_picking():
(4)     fig, ax = plt.subplots()
(4)     col = ax.scatter([0], [0], [1000], picker=True)
(4)     fig.savefig(io.BytesIO(), dpi=fig.dpi)
(4)     mouse_event = SimpleNamespace(x=325, y=240)
(4)     found, indices = col.contains(mouse_event)
(4)     assert found
(4)     assert_array_equal(indices['ind'], [0])
(0) def test_quadmesh_contains():
(4)     x = np.arange(4)
(4)     X = x[:, None] * x[None, :]
(4)     fig, ax = plt.subplots()
(4)     mesh = ax.pcolormesh(X)
(4)     fig.draw_without_rendering()
(4)     xdata, ydata = 0.5, 0.5
(4)     x, y = mesh.get_transform().transform((xdata, ydata))
(4)     mouse_event = SimpleNamespace(xdata=xdata, ydata=ydata, x=x, y=y)
(4)     found, indices = mesh.contains(mouse_event)
(4)     assert found
(4)     assert_array_equal(indices['ind'], [0])
(4)     xdata, ydata = 1.5, 1.5
(4)     x, y = mesh.get_transform().transform((xdata, ydata))
(4)     mouse_event = SimpleNamespace(xdata=xdata, ydata=ydata, x=x, y=y)
(4)     found, indices = mesh.contains(mouse_event)
(4)     assert found
(4)     assert_array_equal(indices['ind'], [5])
(0) def test_quadmesh_contains_concave():
(4)     x = [[0, -1], [1, 0]]
(4)     y = [[0, 1], [1, -1]]
(4)     fig, ax = plt.subplots()
(4)     mesh = ax.pcolormesh(x, y, [[0]])
(4)     fig.draw_without_rendering()
(4)     points = [(-0.5, 0.25, True),  # left wing
(14)              (0, 0.25, False),  # between the two wings
(14)              (0.5, 0.25, True),  # right wing
(14)              (0, -0.25, True),  # main body
(14)              ]
(4)     for point in points:
(8)         xdata, ydata, expected = point
(8)         x, y = mesh.get_transform().transform((xdata, ydata))
(8)         mouse_event = SimpleNamespace(xdata=xdata, ydata=ydata, x=x, y=y)
(8)         found, indices = mesh.contains(mouse_event)
(8)         assert found is expected
(0) def test_quadmesh_cursor_data():
(4)     x = np.arange(4)
(4)     X = x[:, None] * x[None, :]
(4)     fig, ax = plt.subplots()
(4)     mesh = ax.pcolormesh(X)
(4)     mesh._A = None
(4)     fig.draw_without_rendering()
(4)     xdata, ydata = 0.5, 0.5
(4)     x, y = mesh.get_transform().transform((xdata, ydata))
```

```
472: (4)                mouse_event = SimpleNamespace(xdata=xdata, ydata=ydata, x=x, y=y)
473: (4)                assert mesh.get_cursor_data(mouse_event) is None
474: (4)                mesh.set_array(np.ones(X.shape))
475: (4)                assert_array_equal(mesh.get_cursor_data(mouse_event), [1])
476: (0)            def test_quadmesh_cursor_data_multiple_points():
477: (4)                x = [1, 2, 1, 2]
478: (4)                fig, ax = plt.subplots()
479: (4)                mesh = ax.pcolormesh(x, x, np.ones((3, 3)))
480: (4)                fig.draw_without_rendering()
481: (4)                xdata, ydata = 1.5, 1.5
482: (4)                x, y = mesh.get_transform().transform((xdata, ydata))
483: (4)                mouse_event = SimpleNamespace(xdata=xdata, ydata=ydata, x=x, y=y)
484: (4)                assert_array_equal(mesh.get_cursor_data(mouse_event), np.ones(9))
485: (0)            def test_linestyle_single_dashes():
486: (4)                plt.scatter([0, 1, 2], [0, 1, 2], linestyle=(0., [2., 2.]))
487: (4)                plt.draw()
488: (0)            @image_comparison(['size_in_xy.png'], remove_text=True)
489: (0)            def test_size_in_xy():
490: (4)                fig, ax = plt.subplots()
491: (4)                widths, heights, angles = (10, 10), 10, 0
492: (4)                widths = 10, 10
493: (4)                coords = [(10, 10), (15, 15)]
494: (4)                e = mcollections.EllipseCollection(
495: (8)                    widths, heights, angles, units='xy',
496: (8)                    offsets=coords, offset_transform=ax.transData)
497: (4)                ax.add_collection(e)
498: (4)                ax.set_xlim(0, 30)
499: (4)                ax.set_ylim(0, 30)
500: (0)            def test_pandas_indexing(pd):
501: (4)                index = [11, 12, 13]
502: (4)                ec = fc = pd.Series(['red', 'blue', 'green'], index=index)
503: (4)                lw = pd.Series([1, 2, 3], index=index)
504: (4)                ls = pd.Series(['solid', 'dashed', 'dashdot'], index=index)
505: (4)                aa = pd.Series([True, False, True], index=index)
506: (4)                Collection(edgecolors=ec)
507: (4)                Collection(facecolors=fc)
508: (4)                Collection(linewidths=lw)
509: (4)                Collection(linestyles=ls)
510: (4)                Collection(antialiaseds=aa)
511: (0)            @mpl.style.context('default')
512: (0)            def test_lslw_bcast():
513: (4)                col = mcollections.PathCollection([])
514: (4)                col.set_linestyles(['-', '-'])
515: (4)                col.set_linewidths([1, 2, 3])
516: (4)                assert col.get_linestyles() == [(0, None)] * 6
517: (4)                assert col.get_linewidths() == [1, 2, 3] * 2
518: (4)                col.set_linestyles(['-', '-', '-'])
519: (4)                assert col.get_linestyles() == [(0, None)] * 3
520: (4)                assert (col.get_linewidths() == [1, 2, 3]).all()
521: (0)            def test_set_wrong_linestyle():
522: (4)                c = Collection()
523: (4)                with pytest.raises(ValueError, match="Do not know how to convert
'fuzzy'"):
524: (8)                    c.set_linestyle('fuzzy')
525: (0)            @mpl.style.context('default')
526: (0)            def test_capstyle():
527: (4)                col = mcollections.PathCollection([])
528: (4)                assert col.get_capstyle() is None
529: (4)                col = mcollections.PathCollection([], capstyle='round')
530: (4)                assert col.get_capstyle() == 'round'
531: (4)                col.set_capstyle('butt')
532: (4)                assert col.get_capstyle() == 'butt'
533: (0)            @mpl.style.context('default')
534: (0)            def test_joinstyle():
535: (4)                col = mcollections.PathCollection([])
536: (4)                assert col.get_joinstyle() is None
537: (4)                col = mcollections.PathCollection([], joinstyle='round')
538: (4)                assert col.get_joinstyle() == 'round'
539: (4)                col.set_joinstyle('miter')
```

```
(4)                 assert col.get_joinstyle() == 'miter'
(0)             @image_comparison(['cap_and_joinstyle.png'])
(0)             def test_cap_and_joinstyle_image():
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_xlim([-0.5, 1.5])
(4)                 ax.set_ylim([-0.5, 2.5])
(4)                 x = np.array([0.0, 1.0, 0.5])
(4)                 ys = np.array([[0.0], [0.5], [1.0]]) + np.array([[0.0, 0.0, 1.0]])
(4)                 segs = np.zeros((3, 3, 2))
(4)                 segs[:, :, 0] = x
(4)                 segs[:, :, 1] = ys
(4)                 line_segments = LineCollection(segs, linewidth=[10, 15, 20])
(4)                 line_segments.set_capstyle("round")
(4)                 line_segments.set_joinstyle("miter")
(4)                 ax.add_collection(line_segments)
(4)                 ax.set_title('Line collection with customized caps and joinstyle')
(0)             @image_comparison(['scatter_post_alpha.png'],
(18)                            remove_text=True, style='default')
(0)             def test_scatter_post_alpha():
(4)                 fig, ax = plt.subplots()
(4)                 sc = ax.scatter(range(5), range(5), c=range(5))
(4)                 sc.set_alpha(.1)
(0)             def test_scatter_alpha_array():
(4)                 x = np.arange(5)
(4)                 alpha = x / 5
(4)                 fig, (ax0, ax1) = plt.subplots(2)
(4)                 sc0 = ax0.scatter(x, x, c=x, alpha=alpha)
(4)                 sc1 = ax1.scatter(x, x, c=x)
(4)                 sc1.set_alpha(alpha)
(4)                 plt.draw()
(4)                 assert_array_equal(sc0.get_facecolors()[:, -1], alpha)
(4)                 assert_array_equal(sc1.get_facecolors()[:, -1], alpha)
(4)                 fig, (ax0, ax1) = plt.subplots(2)
(4)                 sc0 = ax0.scatter(x, x, color=['r', 'g', 'b', 'c', 'm'], alpha=alpha)
(4)                 sc1 = ax1.scatter(x, x, color='r', alpha=alpha)
(4)                 plt.draw()
(4)                 assert_array_equal(sc0.get_facecolors()[:, -1], alpha)
(4)                 assert_array_equal(sc1.get_facecolors()[:, -1], alpha)
(4)                 fig, (ax0, ax1) = plt.subplots(2)
(4)                 sc0 = ax0.scatter(x, x, color=['r', 'g', 'b', 'c', 'm'])
(4)                 sc0.set_alpha(alpha)
(4)                 sc1 = ax1.scatter(x, x, color='r')
(4)                 sc1.set_alpha(alpha)
(4)                 plt.draw()
(4)                 assert_array_equal(sc0.get_facecolors()[:, -1], alpha)
(4)                 assert_array_equal(sc1.get_facecolors()[:, -1], alpha)
(0)             def test_pathcollection_legend_elements():
(4)                 np.random.seed(19680801)
(4)                 x, y = np.random.rand(2, 10)
(4)                 y = np.random.rand(10)
(4)                 c = np.random.randint(0, 5, size=10)
(4)                 s = np.random.randint(10, 300, size=10)
(4)                 fig, ax = plt.subplots()
(4)                 sc = ax.scatter(x, y, c=c, s=s, cmap="jet", marker="o", linewidths=0)
(4)                 h, l = sc.legend_elements(fmt="{x:g}")
(4)                 assert len(h) == 5
(4)                 assert l == ["0", "1", "2", "3", "4"]
(4)                 colors = np.array([line.get_color() for line in h])
(4)                 colors2 = sc.cmap(np.arange(5)/4)
(4)                 assert_array_equal(colors, colors2)
(4)                 l1 = ax.legend(h, l, loc=1)
(4)                 h2, lab2 = sc.legend_elements(num=9)
(4)                 assert len(h2) == 9
(4)                 l2 = ax.legend(h2, lab2, loc=2)
(4)                 h, l = sc.legend_elements(prop="sizes", alpha=0.5, color="red")
(4)                 assert all(line.get_alpha() == 0.5 for line in h)
(4)                 assert all(line.get_markerfacecolor() == "red" for line in h)
(4)                 l3 = ax.legend(h, l, loc=4)
(4)                 h, l = sc.legend_elements(prop="sizes", num=4, fmt="{x:.2f}",
```

```
(30)                                    func=lambda x: 2*x)
(4)                 actsizes = [line.get_markersize() for line in h]
(4)                 labeledsizes = np.sqrt(np.array(l, float) / 2)
(4)                 assert_array_almost_equal(actsizes, labeledsizes)
(4)                 l4 = ax.legend(h, l, loc=3)
(4)                 loc = mpl.ticker.MaxNLocator(nbins=9, min_n_ticks=9-1,
(33)                                                steps=[1, 2, 2.5, 3, 5, 6, 8, 10])
(4)                 h5, lab5 = sc.legend_elements(num=loc)
(4)                 assert len(h2) == len(h5)
(4)                 levels = [-1, 0, 55.4, 260]
(4)                 h6, lab6 = sc.legend_elements(num=levels, prop="sizes", fmt="{x:g}")
(4)                 assert [float(l) for l in lab6] == levels[2:]
(4)                 for l in [l1, l2, l3, l4]:
(8)                     ax.add_artist(l)
(4)                 fig.canvas.draw()
(0)             def test_EventCollection_nosort():
(4)                 arr = np.array([3, 2, 1, 10])
(4)                 coll = EventCollection(arr)
(4)                 np.testing.assert_array_equal(arr, np.array([3, 2, 1, 10]))
(0)             def test_collection_set_verts_array():
(4)                 verts = np.arange(80, dtype=np.double).reshape(10, 4, 2)
(4)                 col_arr = PolyCollection(verts)
(4)                 col_list = PolyCollection(list(verts))
(4)                 assert len(col_arr._paths) == len(col_list._paths)
(4)                 for ap, lp in zip(col_arr._paths, col_list._paths):
(8)                     assert np.array_equal(ap._vertices, lp._vertices)
(8)                     assert np.array_equal(ap._codes, lp._codes)
(4)                 verts_tuple = np.empty(10, dtype=object)
(4)                 verts_tuple[:] = [tuple(tuple(y) for y in x) for x in verts]
(4)                 col_arr_tuple = PolyCollection(verts_tuple)
(4)                 assert len(col_arr._paths) == len(col_arr_tuple._paths)
(4)                 for ap, atp in zip(col_arr._paths, col_arr_tuple._paths):
(8)                     assert np.array_equal(ap._vertices, atp._vertices)
(8)                     assert np.array_equal(ap._codes, atp._codes)
(0)             def test_collection_set_array():
(4)                 vals = [*range(10)]
(4)                 c = Collection()
(4)                 c.set_array(vals)
(4)                 with pytest.raises(TypeError, match="^Image data of dtype"):
(8)                     c.set_array("wrong_input")
(4)                 vals[5] = 45
(4)                 assert np.not_equal(vals, c.get_array()).any()
(0)             def test_blended_collection_autolim():
(4)                 a = [1, 2, 4]
(4)                 height = .2
(4)                 xy_pairs = np.column_stack([np.repeat(a, 2), np.tile([0, height],
len(a))])
(4)                 line_segs = xy_pairs.reshape([len(a), 2, 2])
(4)                 f, ax = plt.subplots()
(4)                 trans = mtransforms.blended_transform_factory(ax.transData, ax.transAxes)
(4)                 ax.add_collection(LineCollection(line_segs, transform=trans))
(4)                 ax.autoscale_view(scalex=True, scaley=False)
(4)                 np.testing.assert_allclose(ax.get_xlim(), [1., 4.])
(0)             def test_singleton_autolim():
(4)                 fig, ax = plt.subplots()
(4)                 ax.scatter(0, 0)
(4)                 np.testing.assert_allclose(ax.get_ylim(), [-0.06, 0.06])
(4)                 np.testing.assert_allclose(ax.get_xlim(), [-0.06, 0.06])
(0)             @pytest.mark.parametrize("transform, expected", [
(4)                 ("transData", (-0.5, 3.5)),
(4)                 ("transAxes", (2.8, 3.2)),
(0)             ])
(0)             def test_autolim_with_zeros(transform, expected):
(4)                 fig, ax = plt.subplots()
(4)                 ax.scatter(0, 0, transform=getattr(ax, transform))
(4)                 ax.scatter(3, 3)
(4)                 np.testing.assert_allclose(ax.get_ylim(), expected)
(4)                 np.testing.assert_allclose(ax.get_xlim(), expected)
(0)             def test_quadmesh_set_array_validation(pcfunc):
```

```
677: (4)                x = np.arange(11)
678: (4)                y = np.arange(8)
679: (4)                z = np.random.random((7, 10))
680: (4)                fig, ax = plt.subplots()
681: (4)                coll = getattr(ax, pcfunc)(x, y, z)
682: (4)                with pytest.raises(ValueError, match=re.escape(
683: (12)                       "For X (11) and Y (8) with flat shading, A should have shape "
684: (12)                       "(7, 10, 3) or (7, 10, 4) or (7, 10) or (70,), not (10, 7)")):
685: (8)                    coll.set_array(z.reshape(10, 7))
686: (4)                z = np.arange(54).reshape((6, 9))
687: (4)                with pytest.raises(ValueError, match=re.escape(
688: (12)                       "For X (11) and Y (8) with flat shading, A should have shape "
689: (12)                       "(7, 10, 3) or (7, 10, 4) or (7, 10) or (70,), not (6, 9)")):
690: (8)                    coll.set_array(z)
691: (4)                with pytest.raises(ValueError, match=re.escape(
692: (12)                       "For X (11) and Y (8) with flat shading, A should have shape "
693: (12)                       "(7, 10, 3) or (7, 10, 4) or (7, 10) or (70,), not (54,)")):
694: (8)                    coll.set_array(z.ravel())
695: (4)                z = np.ones((9, 6, 3))  # RGB with wrong X/Y dims
696: (4)                with pytest.raises(ValueError, match=re.escape(
697: (12)                       "For X (11) and Y (8) with flat shading, A should have shape "
698: (12)                       "(7, 10, 3) or (7, 10, 4) or (7, 10) or (70,), not (9, 6, 3)")):
699: (8)                    coll.set_array(z)
700: (4)                z = np.ones((9, 6, 4))  # RGBA with wrong X/Y dims
701: (4)                with pytest.raises(ValueError, match=re.escape(
702: (12)                       "For X (11) and Y (8) with flat shading, A should have shape "
703: (12)                       "(7, 10, 3) or (7, 10, 4) or (7, 10) or (70,), not (9, 6, 4)")):
704: (8)                    coll.set_array(z)
705: (4)                z = np.ones((7, 10, 2))  # Right X/Y dims, bad 3rd dim
706: (4)                with pytest.raises(ValueError, match=re.escape(
707: (12)                       "For X (11) and Y (8) with flat shading, A should have shape "
708: (12)                       "(7, 10, 3) or (7, 10, 4) or (7, 10) or (70,), not (7, 10, 2)")):
709: (8)                    coll.set_array(z)
710: (4)                x = np.arange(10)
711: (4)                y = np.arange(7)
712: (4)                z = np.random.random((7, 10))
713: (4)                fig, ax = plt.subplots()
714: (4)                coll = ax.pcolormesh(x, y, z, shading='gouraud')
715: (0)            def test_polyquadmesh_masked_vertices_array():
716: (4)                xx, yy = np.meshgrid([0, 1, 2], [0, 1, 2, 3])
717: (4)                zz = (xx*yy)[:-1, :-1]
718: (4)                quadmesh = plt.pcolormesh(xx, yy, zz)
719: (4)                quadmesh.update_scalarmappable()
720: (4)                quadmesh_fc = quadmesh.get_facecolor()[1:, :]
721: (4)                xx = np.ma.masked_where((xx == 0) & (yy == 0), xx)
722: (4)                polymesh = plt.pcolor(xx, yy, zz)
723: (4)                polymesh.update_scalarmappable()
724: (4)                assert len(polymesh.get_paths()) == 5
725: (4)                assert_array_equal(quadmesh_fc, polymesh.get_facecolor())
726: (4)                yy = np.ma.masked_where((xx == 0) & (yy == 0), yy)
727: (4)                polymesh = plt.pcolor(xx, yy, zz)
728: (4)                polymesh.update_scalarmappable()
729: (4)                assert len(polymesh.get_paths()) == 5
730: (4)                assert_array_equal(quadmesh_fc, polymesh.get_facecolor())
731: (4)                zz = np.ma.masked_where((xx[:-1, :-1] == 0) & (yy[:-1, :-1] == 0), zz)
732: (4)                polymesh = plt.pcolor(zz)
733: (4)                polymesh.update_scalarmappable()
734: (4)                assert len(polymesh.get_paths()) == 5
735: (4)                assert_array_equal(quadmesh_fc, polymesh.get_facecolor())
736: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning,
737: (22)                                  match="Setting a PolyQuadMesh"):
738: (8)                    polymesh.set_array(np.ones(5))
739: (4)                zz = np.arange(6).reshape((3, 2))
740: (4)                polymesh.set_array(zz)
741: (4)                polymesh.update_scalarmappable()
742: (4)                assert len(polymesh.get_paths()) == 6
743: (4)                zz = np.ma.masked_less(zz, 2)
744: (4)                polymesh.set_array(zz)
745: (4)                polymesh.update_scalarmappable()
```

```
(4)                 assert len(polymesh.get_paths()) == 4
(0)             def test_quadmesh_get_coordinates(pcfunc):
(4)                 x = [0, 1, 2]
(4)                 y = [2, 4, 6]
(4)                 z = np.ones(shape=(2, 2))
(4)                 xx, yy = np.meshgrid(x, y)
(4)                 coll = getattr(plt, pcfunc)(xx, yy, z)
(4)                 coords = np.stack([xx.T, yy.T]).T
(4)                 assert_array_equal(coll.get_coordinates(), coords)
(0)             def test_quadmesh_set_array():
(4)                 x = np.arange(4)
(4)                 y = np.arange(4)
(4)                 z = np.arange(9).reshape((3, 3))
(4)                 fig, ax = plt.subplots()
(4)                 coll = ax.pcolormesh(x, y, np.ones(z.shape))
(4)                 coll.set_array(z)
(4)                 fig.canvas.draw()
(4)                 assert np.array_equal(coll.get_array(), z)
(4)                 coll.set_array(np.ones(9))
(4)                 fig.canvas.draw()
(4)                 assert np.array_equal(coll.get_array(), np.ones(9))
(4)                 z = np.arange(16).reshape((4, 4))
(4)                 fig, ax = plt.subplots()
(4)                 coll = ax.pcolormesh(x, y, np.ones(z.shape), shading='gouraud')
(4)                 coll.set_array(z)
(4)                 fig.canvas.draw()
(4)                 assert np.array_equal(coll.get_array(), z)
(4)                 coll.set_array(np.ones(16))
(4)                 fig.canvas.draw()
(4)                 assert np.array_equal(coll.get_array(), np.ones(16))
(0)             def test_quadmesh_vmin_vmax(pcfunc):
(4)                 fig, ax = plt.subplots()
(4)                 cmap = mpl.colormaps['plasma']
(4)                 norm = mpl.colors.Normalize(vmin=0, vmax=1)
(4)                 coll = getattr(ax, pcfunc)([[1]], cmap=cmap, norm=norm)
(4)                 fig.canvas.draw()
(4)                 assert np.array_equal(coll.get_facecolors()[0, :], cmap(norm(1)))
(4)                 norm.vmin, norm.vmax = 1, 2
(4)                 fig.canvas.draw()
(4)                 assert np.array_equal(coll.get_facecolors()[0, :], cmap(norm(1)))
(0)             def test_quadmesh_alpha_array(pcfunc):
(4)                 x = np.arange(4)
(4)                 y = np.arange(4)
(4)                 z = np.arange(9).reshape((3, 3))
(4)                 alpha = z / z.max()
(4)                 alpha_flat = alpha.ravel()
(4)                 fig, (ax0, ax1) = plt.subplots(2)
(4)                 coll1 = getattr(ax0, pcfunc)(x, y, z, alpha=alpha)
(4)                 coll2 = getattr(ax0, pcfunc)(x, y, z)
(4)                 coll2.set_alpha(alpha)
(4)                 plt.draw()
(4)                 assert_array_equal(coll1.get_facecolors()[:, -1], alpha_flat)
(4)                 assert_array_equal(coll2.get_facecolors()[:, -1], alpha_flat)
(4)                 fig, (ax0, ax1) = plt.subplots(2)
(4)                 coll1 = getattr(ax0, pcfunc)(x, y, z, alpha=alpha)
(4)                 coll2 = getattr(ax1, pcfunc)(x, y, z)
(4)                 coll2.set_alpha(alpha)
(4)                 plt.draw()
(4)                 assert_array_equal(coll1.get_facecolors()[:, -1], alpha_flat)
(4)                 assert_array_equal(coll2.get_facecolors()[:, -1], alpha_flat)
(0)             def test_alpha_validation(pcfunc):
(4)                 fig, ax = plt.subplots()
(4)                 pc = getattr(ax, pcfunc)(np.arange(12).reshape((3, 4)))
(4)                 with pytest.raises(ValueError, match="^Data array shape"):
(8)                     pc.set_alpha([0.5, 0.6])
(8)                     pc.update_scalarmappable()
(0)             def test_legend_inverse_size_label_relationship():
(4)                 """
(4)                 Ensure legend markers scale appropriately when label and size are
```

```
(4)                 inversely related.
(4)                 Here label = 5 / size
(4)                 """
(4)                 np.random.seed(19680801)
(4)                 X = np.random.random(50)
(4)                 Y = np.random.random(50)
(4)                 C = 1 - np.random.random(50)
(4)                 S = 5 / C
(4)                 legend_sizes = [0.2, 0.4, 0.6, 0.8]
(4)                 fig, ax = plt.subplots()
(4)                 sc = ax.scatter(X, Y, s=S)
(4)                 handles, labels = sc.legend_elements(
(6)                   prop='sizes', num=legend_sizes, func=lambda s: 5 / s
(4)                 )
(4)                 handle_sizes = [x.get_markersize() for x in handles]
(4)                 handle_sizes = [5 / x**2 for x in handle_sizes]
(4)                 assert_array_almost_equal(handle_sizes, legend_sizes, decimal=1)
(0)             @mpl.style.context('default')
(0)             def test_color_logic(pcfunc):
(4)                 pcfunc = getattr(plt, pcfunc)
(4)                 z = np.arange(12).reshape(3, 4)
(4)                 pc = pcfunc(z, edgecolors='red', facecolors='none')
(4)                 pc.update_scalarmappable()  # This is called in draw().
(4)                 face_default = mcolors.to_rgba_array(pc._get_default_facecolor())
(4)                 mapped = pc.get_cmap()(pc.norm(z.ravel()))
(4)                 assert mcolors.same_color(pc.get_edgecolor(), 'red')
(4)                 pc = pcfunc(z)
(4)                 pc.set_facecolor('none')
(4)                 pc.set_edgecolor('red')
(4)                 pc.update_scalarmappable()
(4)                 assert mcolors.same_color(pc.get_facecolor(), 'none')
(4)                 assert mcolors.same_color(pc.get_edgecolor(), [[1, 0, 0, 1]])
(4)                 pc.set_alpha(0.5)
(4)                 pc.update_scalarmappable()
(4)                 assert mcolors.same_color(pc.get_edgecolor(), [[1, 0, 0, 0.5]])
(4)                 pc.set_alpha(None)  # restore default alpha
(4)                 pc.update_scalarmappable()
(4)                 assert mcolors.same_color(pc.get_edgecolor(), [[1, 0, 0, 1]])
(4)                 pc.set_edgecolor(None)
(4)                 pc.update_scalarmappable()
(4)                 assert np.array_equal(pc.get_edgecolor(), mapped)
(4)                 pc.set_facecolor(None)  # restore default for facecolor
(4)                 pc.update_scalarmappable()
(4)                 assert np.array_equal(pc.get_facecolor(), mapped)
(4)                 assert mcolors.same_color(pc.get_edgecolor(), 'none')
(4)                 pc.set_array(None)
(4)                 pc.update_scalarmappable()
(4)                 assert mcolors.same_color(pc.get_edgecolor(), 'none')
(4)                 assert mcolors.same_color(pc.get_facecolor(), face_default)  # not mapped
(4)                 pc.set_array(z)
(4)                 pc.update_scalarmappable()
(4)                 assert np.array_equal(pc.get_facecolor(), mapped)
(4)                 assert mcolors.same_color(pc.get_edgecolor(), 'none')
(4)                 pc = pcfunc(z, edgecolors=(1, 0, 0), facecolors=(0, 1, 0))
(4)                 pc.update_scalarmappable()
(4)                 assert np.array_equal(pc.get_facecolor(), mapped)
(4)                 assert mcolors.same_color(pc.get_edgecolor(), [[1, 0, 0, 1]])
(4)                 pc = pcfunc(z, edgecolors=(1, 0, 0), facecolors=np.ones((12, 3)))
(4)                 pc.update_scalarmappable()
(4)                 assert np.array_equal(pc.get_facecolor(), mapped)
(4)                 assert mcolors.same_color(pc.get_edgecolor(), [[1, 0, 0, 1]])
(4)                 pc.set_array(None)
(4)                 pc.update_scalarmappable()
(4)                 assert mcolors.same_color(pc.get_facecolor(), np.ones((12, 3)))
(4)                 assert mcolors.same_color(pc.get_edgecolor(), [[1, 0, 0, 1]])
(4)                 pc = pcfunc(z, edgecolors=(1, 0, 0), facecolors=np.ones((12, 4)))
(4)                 pc.update_scalarmappable()
(4)                 assert np.array_equal(pc.get_facecolor(), mapped)
(4)                 assert mcolors.same_color(pc.get_edgecolor(), [[1, 0, 0, 1]])
```

```
(4)                 pc.set_array(None)
(4)                 pc.update_scalarmappable()
(4)                 assert mcolors.same_color(pc.get_facecolor(), np.ones((12, 4)))
(4)                 assert mcolors.same_color(pc.get_edgecolor(), [[1, 0, 0, 1]])
(0)             def test_LineCollection_args():
(4)                 lc = LineCollection(None, linewidth=2.2, edgecolor='r',
(24)                                    zorder=3, facecolors=[0, 1, 0, 1])
(4)                 assert lc.get_linewidth()[0] == 2.2
(4)                 assert mcolors.same_color(lc.get_edgecolor(), 'r')
(4)                 assert lc.get_zorder() == 3
(4)                 assert mcolors.same_color(lc.get_facecolor(), [[0, 1, 0, 1]])
(4)                 lc = LineCollection(None, facecolor=None)
(4)                 assert mcolors.same_color(lc.get_facecolor(), 'none')
(0)             def test_array_dimensions(pcfunc):
(4)                 z = np.arange(12).reshape(3, 4)
(4)                 pc = getattr(plt, pcfunc)(z)
(4)                 pc.set_array(z.ravel())
(4)                 pc.update_scalarmappable()
(4)                 pc.set_array(z)
(4)                 pc.update_scalarmappable()
(4)                 z = np.arange(36, dtype=np.uint8).reshape(3, 4, 3)
(4)                 pc.set_array(z)
(4)                 pc.update_scalarmappable()
(0)             def test_get_segments():
(4)                 segments = np.tile(np.linspace(0, 1, 256), (2, 1)).T
(4)                 lc = LineCollection([segments])
(4)                 readback, = lc.get_segments()
(4)                 assert np.all(segments == readback)
(0)             def test_set_offsets_late():
(4)                 identity = mtransforms.IdentityTransform()
(4)                 sizes = [2]
(4)                 null = mcollections.CircleCollection(sizes=sizes)
(4)                 init = mcollections.CircleCollection(sizes=sizes, offsets=(10, 10))
(4)                 late = mcollections.CircleCollection(sizes=sizes)
(4)                 late.set_offsets((10, 10))
(4)                 null_bounds = null.get_datalim(identity).bounds
(4)                 init_bounds = init.get_datalim(identity).bounds
(4)                 late_bounds = late.get_datalim(identity).bounds
(4)                 assert null_bounds != init_bounds
(4)                 assert init_bounds == late_bounds
(0)             def test_set_offset_transform():
(4)                 skew = mtransforms.Affine2D().skew(2, 2)
(4)                 init = mcollections.Collection(offset_transform=skew)
(4)                 late = mcollections.Collection()
(4)                 late.set_offset_transform(skew)
(4)                 assert skew == init.get_offset_transform() == late.get_offset_transform()
(0)             def test_set_offset_units():
(4)                 x = np.linspace(0, 10, 5)
(4)                 y = np.sin(x)
(4)                 d = x * np.timedelta64(24, 'h') + np.datetime64('2021-11-29')
(4)                 sc = plt.scatter(d, y)
(4)                 off0 = sc.get_offsets()
(4)                 sc.set_offsets(list(zip(d, y)))
(4)                 np.testing.assert_allclose(off0, sc.get_offsets())
(4)                 fig, ax = plt.subplots()
(4)                 sc = ax.scatter(y, d)
(4)                 off0 = sc.get_offsets()
(4)                 sc.set_offsets(list(zip(y, d)))
(4)                 np.testing.assert_allclose(off0, sc.get_offsets())
(0)             @image_comparison(baseline_images=["test_check_masked_offsets"],
(18)                              extensions=["png"], remove_text=True, style="mpl20")
(0)             def test_check_masked_offsets():
(4)                 unmasked_x = [
(8)                     datetime(2022, 12, 15, 4, 49, 52),
(8)                     datetime(2022, 12, 15, 4, 49, 53),
(8)                     datetime(2022, 12, 15, 4, 49, 54),
(8)                     datetime(2022, 12, 15, 4, 49, 55),
(8)                     datetime(2022, 12, 15, 4, 49, 56),
(4)                 ]
```

```
953: (4)                masked_y = np.ma.array([1, 2, 3, 4, 5], mask=[0, 1, 1, 0, 0])
954: (4)                fig, ax = plt.subplots()
955: (4)                ax.scatter(unmasked_x, masked_y)
956: (0)            @check_figures_equal(extensions=["png"])
957: (0)            def test_masked_set_offsets(fig_ref, fig_test):
958: (4)                x = np.ma.array([1, 2, 3, 4, 5], mask=[0, 0, 1, 1, 0])
959: (4)                y = np.arange(1, 6)
960: (4)                ax_test = fig_test.add_subplot()
961: (4)                scat = ax_test.scatter(x, y)
962: (4)                scat.set_offsets(np.ma.column_stack([x, y]))
963: (4)                ax_test.set_xticks([])
964: (4)                ax_test.set_yticks([])
965: (4)                ax_ref = fig_ref.add_subplot()
966: (4)                ax_ref.scatter([1, 2, 5], [1, 2, 5])
967: (4)                ax_ref.set_xticks([])
968: (4)                ax_ref.set_yticks([])
969: (0)            def test_check_offsets_dtype():
970: (4)                x = np.ma.array([1, 2, 3, 4, 5], mask=[0, 0, 1, 1, 0])
971: (4)                y = np.arange(1, 6)
972: (4)                fig, ax = plt.subplots()
973: (4)                scat = ax.scatter(x, y)
974: (4)                masked_offsets = np.ma.column_stack([x, y])
975: (4)                scat.set_offsets(masked_offsets)
976: (4)                assert isinstance(scat.get_offsets(), type(masked_offsets))
977: (4)                unmasked_offsets = np.column_stack([x, y])
978: (4)                scat.set_offsets(unmasked_offsets)
979: (4)                assert isinstance(scat.get_offsets(), type(unmasked_offsets))
980: (0)            @pytest.mark.parametrize('gapcolor', ['orange', ['r', 'k']])
981: (0)            @check_figures_equal(extensions=['png'])
982: (0)            @mpl.rc_context({'lines.linewidth': 20})
983: (0)            def test_striped_lines(fig_test, fig_ref, gapcolor):
984: (4)                ax_test = fig_test.add_subplot(111)
985: (4)                ax_ref = fig_ref.add_subplot(111)
986: (4)                for ax in [ax_test, ax_ref]:
987: (8)                    ax.set_xlim(0, 6)
988: (8)                    ax.set_ylim(0, 1)
989: (4)                x = range(1, 6)
990: (4)                linestyles = [':', '-', '--']
991: (4)                ax_test.vlines(x, 0, 1, linestyle=linestyles, gapcolor=gapcolor,
alpha=0.5)
992: (4)                if isinstance(gapcolor, str):
993: (8)                    gapcolor = [gapcolor]
994: (4)                for x, gcol, ls in zip(x, itertools.cycle(gapcolor),
995: (27)                                      itertools.cycle(linestyles)):
996: (8)                    ax_ref.axvline(x, 0, 1, linestyle=ls, gapcolor=gcol, alpha=0.5)

----------------------------------------

File 172 - test_colorbar.py:

1: (0)              import platform
2: (0)              import numpy as np
3: (0)              import pytest
4: (0)              from matplotlib import cm
5: (0)              import matplotlib.colors as mcolors
6: (0)              import matplotlib as mpl
7: (0)              from matplotlib import rc_context
8: (0)              from matplotlib.testing.decorators import image_comparison
9: (0)              import matplotlib.pyplot as plt
10: (0)             from matplotlib.colors import (
11: (4)                 BoundaryNorm, LogNorm, PowerNorm, Normalize, NoNorm
12: (0)             )
13: (0)             from matplotlib.colorbar import Colorbar
14: (0)             from matplotlib.ticker import FixedLocator, LogFormatter, StrMethodFormatter
15: (0)             from matplotlib.testing.decorators import check_figures_equal
16: (0)             def _get_cmap_norms():
17: (4)                 """
18: (4)                 Define a colormap and appropriate norms for each of the four
19: (4)                 possible settings of the extend keyword.
```

```
 20: (4)                 Helper function for _colorbar_extension_shape and
 21: (4)                 colorbar_extension_length.
 22: (4)                 """
 23: (4)                 cmap = mpl.colormaps["RdBu"].resampled(5)
 24: (4)                 clevs = [-5., -2.5, -.5, .5, 1.5, 3.5]
 25: (4)                 norms = dict()
 26: (4)                 norms['neither'] = BoundaryNorm(clevs, len(clevs) - 1)
 27: (4)                 norms['min'] = BoundaryNorm([-10] + clevs[1:], len(clevs) - 1)
 28: (4)                 norms['max'] = BoundaryNorm(clevs[:-1] + [10], len(clevs) - 1)
 29: (4)                 norms['both'] = BoundaryNorm([-10] + clevs[1:-1] + [10], len(clevs) - 1)
 30: (4)                 return cmap, norms
 31: (0)             def _colorbar_extension_shape(spacing):
 32: (4)                 """
 33: (4)                 Produce 4 colorbars with rectangular extensions for either uniform
 34: (4)                 or proportional spacing.
 35: (4)                 Helper function for test_colorbar_extension_shape.
 36: (4)                 """
 37: (4)                 cmap, norms = _get_cmap_norms()
 38: (4)                 fig = plt.figure()
 39: (4)                 fig.subplots_adjust(hspace=4)
 40: (4)                 for i, extension_type in enumerate(('neither', 'min', 'max', 'both')):
 41: (8)                     norm = norms[extension_type]
 42: (8)                     boundaries = values = norm.boundaries
 43: (8)                     values = values[:-1]
 44: (8)                     cax = fig.add_subplot(4, 1, i + 1)
 45: (8)                     Colorbar(cax, cmap=cmap, norm=norm,
 46: (17)                             boundaries=boundaries, values=values,
 47: (17)                             extend=extension_type, extendrect=True,
 48: (17)                             orientation='horizontal', spacing=spacing)
 49: (8)                     cax.tick_params(left=False, labelleft=False,
 50: (24)                                     bottom=False, labelbottom=False)
 51: (4)                 return fig
 52: (0)             def _colorbar_extension_length(spacing):
 53: (4)                 """
 54: (4)                 Produce 12 colorbars with variable length extensions for either
 55: (4)                 uniform or proportional spacing.
 56: (4)                 Helper function for test_colorbar_extension_length.
 57: (4)                 """
 58: (4)                 cmap, norms = _get_cmap_norms()
 59: (4)                 fig = plt.figure()
 60: (4)                 fig.subplots_adjust(hspace=.6)
 61: (4)                 for i, extension_type in enumerate(('neither', 'min', 'max', 'both')):
 62: (8)                     norm = norms[extension_type]
 63: (8)                     boundaries = values = norm.boundaries
 64: (8)                     values = values[:-1]
 65: (8)                     for j, extendfrac in enumerate((None, 'auto', 0.1)):
 66: (12)                        cax = fig.add_subplot(12, 1, i*3 + j + 1)
 67: (12)                        Colorbar(cax, cmap=cmap, norm=norm,
 68: (21)                                 boundaries=boundaries, values=values,
 69: (21)                                 extend=extension_type, extendfrac=extendfrac,
 70: (21)                                 orientation='horizontal', spacing=spacing)
 71: (12)                        cax.tick_params(left=False, labelleft=False,
 72: (30)                                        bottom=False, labelbottom=False)
 73: (4)                 return fig
 74: (0)             @image_comparison(['colorbar_extensions_shape_uniform.png',
 75: (19)                               'colorbar_extensions_shape_proportional.png'])
 76: (0)             def test_colorbar_extension_shape():
 77: (4)                 """Test rectangular colorbar extensions."""
 78: (4)                 plt.rcParams['pcolormesh.snap'] = False
 79: (4)                 _colorbar_extension_shape('uniform')
 80: (4)                 _colorbar_extension_shape('proportional')
 81: (0)             @image_comparison(['colorbar_extensions_uniform.png',
 82: (19)                               'colorbar_extensions_proportional.png'],
 83: (18)                              tol=1.0)
 84: (0)             def test_colorbar_extension_length():
 85: (4)                 """Test variable length colorbar extensions."""
 86: (4)                 plt.rcParams['pcolormesh.snap'] = False
 87: (4)                 _colorbar_extension_length('uniform')
 88: (4)                 _colorbar_extension_length('proportional')
```

```
 89: (0)             @pytest.mark.parametrize("orientation", ["horizontal", "vertical"])
 90: (0)             @pytest.mark.parametrize("extend,expected", [("min", (0, 0, 0, 1)),
 91: (45)                                                         ("max", (1, 1, 1, 1)),
 92: (45)                                                         ("both", (1, 1, 1, 1))])
 93: (0)             def test_colorbar_extension_inverted_axis(orientation, extend, expected):
 94: (4)                 """Test extension color with an inverted axis"""
 95: (4)                 data = np.arange(12).reshape(3, 4)
 96: (4)                 fig, ax = plt.subplots()
 97: (4)                 cmap = mpl.colormaps["viridis"].with_extremes(under=(0, 0, 0, 1),
 98: (50)                                                              over=(1, 1, 1, 1))
 99: (4)                 im = ax.imshow(data, cmap=cmap)
 100: (4)                cbar = fig.colorbar(im, orientation=orientation, extend=extend)
 101: (4)                if orientation == "horizontal":
 102: (8)                    cbar.ax.invert_xaxis()
 103: (4)                else:
 104: (8)                    cbar.ax.invert_yaxis()
 105: (4)                assert cbar._extend_patches[0].get_facecolor() == expected
 106: (4)                if extend == "both":
 107: (8)                    assert len(cbar._extend_patches) == 2
 108: (8)                    assert cbar._extend_patches[1].get_facecolor() == (0, 0, 0, 1)
 109: (4)                else:
 110: (8)                    assert len(cbar._extend_patches) == 1
 111: (0)            @pytest.mark.parametrize('use_gridspec', [True, False])
 112: (0)            @image_comparison(['cbar_with_orientation',
 113: (19)                              'cbar_locationing',
 114: (19)                              'double_cbar',
 115: (19)                              'cbar_sharing',
 116: (19)                              ],
 117: (18)                             extensions=['png'], remove_text=True,
 118: (18)                             savefig_kwarg={'dpi': 40})
 119: (0)            def test_colorbar_positioning(use_gridspec):
 120: (4)                plt.rcParams['pcolormesh.snap'] = False
 121: (4)                data = np.arange(1200).reshape(30, 40)
 122: (4)                levels = [0, 200, 400, 600, 800, 1000, 1200]
 123: (4)                plt.figure()
 124: (4)                plt.contourf(data, levels=levels)
 125: (4)                plt.colorbar(orientation='horizontal', use_gridspec=use_gridspec)
 126: (4)                locations = ['left', 'right', 'top', 'bottom']
 127: (4)                plt.figure()
 128: (4)                for i, location in enumerate(locations):
 129: (8)                    plt.subplot(2, 2, i + 1)
 130: (8)                    plt.contourf(data, levels=levels)
 131: (8)                    plt.colorbar(location=location, use_gridspec=use_gridspec)
 132: (4)                plt.figure()
 133: (4)                data_2nd = np.array([[2, 3, 2, 3], [1.5, 2, 2, 3], [2, 3, 3, 4]])
 134: (4)                data_2nd = np.repeat(np.repeat(data_2nd, 10, axis=1), 10, axis=0)
 135: (4)                color_mappable = plt.contourf(data, levels=levels, extend='both')
 136: (4)                hatch_mappable = plt.contourf(data_2nd, levels=[1, 2, 3], colors='none',
 137: (34)                                             hatches=['/', 'o', '+'], extend='max')
 138: (4)                plt.contour(hatch_mappable, colors='black')
 139: (4)                plt.colorbar(color_mappable, location='left', label='variable 1',
 140: (17)                            use_gridspec=use_gridspec)
 141: (4)                plt.colorbar(hatch_mappable, location='right', label='variable 2',
 142: (17)                            use_gridspec=use_gridspec)
 143: (4)                plt.figure()
 144: (4)                ax1 = plt.subplot(211, anchor='NE', aspect='equal')
 145: (4)                plt.contourf(data, levels=levels)
 146: (4)                ax2 = plt.subplot(223)
 147: (4)                plt.contourf(data, levels=levels)
 148: (4)                ax3 = plt.subplot(224)
 149: (4)                plt.contourf(data, levels=levels)
 150: (4)                plt.colorbar(ax=[ax2, ax3, ax1], location='right', pad=0.0, shrink=0.5,
 151: (17)                            panchor=False, use_gridspec=use_gridspec)
 152: (4)                plt.colorbar(ax=[ax2, ax3, ax1], location='left', shrink=0.5,
 153: (17)                            panchor=False, use_gridspec=use_gridspec)
 154: (4)                plt.colorbar(ax=[ax1], location='bottom', panchor=False,
 155: (17)                            anchor=(0.8, 0.5), shrink=0.6, use_gridspec=use_gridspec)
 156: (0)            def test_colorbar_single_ax_panchor_false():
 157: (4)                ax = plt.subplot(111, anchor='N')
```

```
158: (4)                plt.imshow([[0, 1]])
159: (4)                plt.colorbar(panchor=False)
160: (4)                assert ax.get_anchor() == 'N'
161: (0)            @pytest.mark.parametrize('constrained', [False, True],
162: (25)                                    ids=['standard', 'constrained'])
163: (0)            def test_colorbar_single_ax_panchor_east(constrained):
164: (4)                fig = plt.figure(constrained_layout=constrained)
165: (4)                ax = fig.add_subplot(111, anchor='N')
166: (4)                plt.imshow([[0, 1]])
167: (4)                plt.colorbar(panchor='E')
168: (4)                assert ax.get_anchor() == 'E'
169: (0)            @image_comparison(['contour_colorbar.png'], remove_text=True,
170: (18)                             tol=0 if platform.machine() == 'x86_64' else 0.054)
171: (0)            def test_contour_colorbar():
172: (4)                fig, ax = plt.subplots(figsize=(4, 2))
173: (4)                data = np.arange(1200).reshape(30, 40) - 500
174: (4)                levels = np.array([0, 200, 400, 600, 800, 1000, 1200]) - 500
175: (4)                CS = ax.contour(data, levels=levels, extend='both')
176: (4)                fig.colorbar(CS, orientation='horizontal', extend='both')
177: (4)                fig.colorbar(CS, orientation='vertical')
178: (0)            @image_comparison(['cbar_with_subplots_adjust.png'], remove_text=True,
179: (18)                             savefig_kwarg={'dpi': 40})
180: (0)            def test_gridspec_make_colorbar():
181: (4)                plt.figure()
182: (4)                data = np.arange(1200).reshape(30, 40)
183: (4)                levels = [0, 200, 400, 600, 800, 1000, 1200]
184: (4)                plt.subplot(121)
185: (4)                plt.contourf(data, levels=levels)
186: (4)                plt.colorbar(use_gridspec=True, orientation='vertical')
187: (4)                plt.subplot(122)
188: (4)                plt.contourf(data, levels=levels)
189: (4)                plt.colorbar(use_gridspec=True, orientation='horizontal')
190: (4)                plt.subplots_adjust(top=0.95, right=0.95, bottom=0.2, hspace=0.25)
191: (0)            @image_comparison(['colorbar_single_scatter.png'], remove_text=True,
192: (18)                             savefig_kwarg={'dpi': 40})
193: (0)            def test_colorbar_single_scatter():
194: (4)                plt.figure()
195: (4)                x = y = [0]
196: (4)                z = [50]
197: (4)                cmap = mpl.colormaps['jet'].resampled(16)
198: (4)                cs = plt.scatter(x, y, z, c=z, cmap=cmap)
199: (4)                plt.colorbar(cs)
200: (0)            @pytest.mark.parametrize('use_gridspec', [True, False])
201: (0)            @pytest.mark.parametrize('nested_gridspecs', [True, False])
202: (0)            def test_remove_from_figure(nested_gridspecs, use_gridspec):
203: (4)                """Test `remove` with the specified ``use_gridspec`` setting."""
204: (4)                fig = plt.figure()
205: (4)                if nested_gridspecs:
206: (8)                    gs = fig.add_gridspec(2, 2)[1, 1].subgridspec(2, 2)
207: (8)                    ax = fig.add_subplot(gs[1, 1])
208: (4)                else:
209: (8)                    ax = fig.add_subplot()
210: (4)                sc = ax.scatter([1, 2], [3, 4])
211: (4)                sc.set_array(np.array([5, 6]))
212: (4)                pre_position = ax.get_position()
213: (4)                cb = fig.colorbar(sc, use_gridspec=use_gridspec)
214: (4)                fig.subplots_adjust()
215: (4)                cb.remove()
216: (4)                fig.subplots_adjust()
217: (4)                post_position = ax.get_position()
218: (4)                assert (pre_position.get_points() == post_position.get_points()).all()
219: (0)            def test_remove_from_figure_cl():
220: (4)                """Test `remove` with constrained_layout."""
221: (4)                fig, ax = plt.subplots(constrained_layout=True)
222: (4)                sc = ax.scatter([1, 2], [3, 4])
223: (4)                sc.set_array(np.array([5, 6]))
224: (4)                fig.draw_without_rendering()
225: (4)                pre_position = ax.get_position()
226: (4)                cb = fig.colorbar(sc)
```

```
227: (4)                cb.remove()
228: (4)                fig.draw_without_rendering()
229: (4)                post_position = ax.get_position()
230: (4)                np.testing.assert_allclose(pre_position.get_points(),
231: (31)                                           post_position.get_points())
232: (0)            def test_colorbarbase():
233: (4)                ax = plt.gca()
234: (4)                Colorbar(ax, cmap=plt.cm.bone)
235: (0)            def test_parentless_mappable():
236: (4)                pc = mpl.collections.PatchCollection([], cmap=plt.get_cmap('viridis'),
array=[])
237: (4)                with pytest.raises(ValueError, match='Unable to determine Axes to steal'):
238: (8)                    plt.colorbar(pc)
239: (0)            @image_comparison(['colorbar_closed_patch.png'], remove_text=True)
240: (0)            def test_colorbar_closed_patch():
241: (4)                plt.rcParams['pcolormesh.snap'] = False
242: (4)                fig = plt.figure(figsize=(8, 6))
243: (4)                ax1 = fig.add_axes([0.05, 0.85, 0.9, 0.1])
244: (4)                ax2 = fig.add_axes([0.1, 0.65, 0.75, 0.1])
245: (4)                ax3 = fig.add_axes([0.05, 0.45, 0.9, 0.1])
246: (4)                ax4 = fig.add_axes([0.05, 0.25, 0.9, 0.1])
247: (4)                ax5 = fig.add_axes([0.05, 0.05, 0.9, 0.1])
248: (4)                cmap = mpl.colormaps["RdBu"].resampled(5)
249: (4)                im = ax1.pcolormesh(np.linspace(0, 10, 16).reshape((4, 4)), cmap=cmap)
250: (4)                values = np.linspace(0, 10, 5)
251: (4)                cbar_kw = dict(orientation='horizontal', values=values, ticks=[])
252: (4)                with rc_context({'axes.linewidth': 16}):
253: (8)                    plt.colorbar(im, cax=ax2, extend='both', extendfrac=0.5, **cbar_kw)
254: (8)                    plt.colorbar(im, cax=ax3, extend='both', **cbar_kw)
255: (8)                    plt.colorbar(im, cax=ax4, extend='both', extendrect=True, **cbar_kw)
256: (8)                    plt.colorbar(im, cax=ax5, extend='neither', **cbar_kw)
257: (0)            def test_colorbar_ticks():
258: (4)                fig, ax = plt.subplots()
259: (4)                x = np.arange(-3.0, 4.001)
260: (4)                y = np.arange(-4.0, 3.001)
261: (4)                X, Y = np.meshgrid(x, y)
262: (4)                Z = X * Y
263: (4)                clevs = np.array([-12, -5, 0, 5, 12], dtype=float)
264: (4)                colors = ['r', 'g', 'b', 'c']
265: (4)                cs = ax.contourf(X, Y, Z, clevs, colors=colors, extend='neither')
266: (4)                cbar = fig.colorbar(cs, ax=ax, orientation='horizontal', ticks=clevs)
267: (4)                assert len(cbar.ax.xaxis.get_ticklocs()) == len(clevs)
268: (0)            def test_colorbar_minorticks_on_off():
269: (4)                np.random.seed(seed=12345)
270: (4)                data = np.random.randn(20, 20)
271: (4)                with rc_context({'_internal.classic_mode': False}):
272: (8)                    fig, ax = plt.subplots()
273: (8)                    im = ax.pcolormesh(data, vmin=-2.3, vmax=3.3)
274: (8)                    cbar = fig.colorbar(im, extend='both')
275: (8)                    cbar.minorticks_on()
276: (8)                    np.testing.assert_almost_equal(
277: (12)                       cbar.ax.yaxis.get_minorticklocs(),
278: (12)                       [-2.2, -1.8, -1.6, -1.4, -1.2, -0.8, -0.6, -0.4, -0.2,
279: (13)                        0.2, 0.4, 0.6, 0.8, 1.2, 1.4, 1.6, 1.8, 2.2, 2.4, 2.6, 2.8, 3.2])
280: (8)                    cbar.minorticks_off()
281: (8)                    np.testing.assert_almost_equal(cbar.ax.yaxis.get_minorticklocs(), [])
282: (8)                    im.set_clim(vmin=-1.2, vmax=1.2)
283: (8)                    cbar.minorticks_on()
284: (8)                    np.testing.assert_almost_equal(
285: (12)                       cbar.ax.yaxis.get_minorticklocs(),
286: (12)                       [-1.2, -1.1, -0.9, -0.8, -0.7, -0.6, -0.4, -0.3, -0.2, -0.1,
287: (13)                        0.1, 0.2, 0.3, 0.4, 0.6, 0.7, 0.8, 0.9, 1.1, 1.2])
288: (4)                data = np.random.uniform(low=1, high=10, size=(20, 20))
289: (4)                fig, ax = plt.subplots()
290: (4)                im = ax.pcolormesh(data, norm=LogNorm())
291: (4)                cbar = fig.colorbar(im)
292: (4)                fig.canvas.draw()
293: (4)                default_minorticklocks = cbar.ax.yaxis.get_minorticklocs()
294: (4)                cbar.minorticks_off()
```

```
295: (4)                np.testing.assert_equal(cbar.ax.yaxis.get_minorticklocs(), [])
296: (4)                cbar.minorticks_on()
297: (4)                np.testing.assert_equal(cbar.ax.yaxis.get_minorticklocs(),
298: (28)                                       default_minorticklocks)
299: (4)                cbar.minorticks_off()
300: (4)                cbar.set_ticks([3, 5, 7, 9])
301: (4)                np.testing.assert_equal(cbar.ax.yaxis.get_minorticklocs(), [])
302: (0)            def test_cbar_minorticks_for_rc_xyminortickvisible():
303: (4)                """
304: (4)                issue gh-16468.
305: (4)                Making sure that minor ticks on the colorbar are turned on
306: (4)                (internally) using the cbar.minorticks_on() method when
307: (4)                rcParams['xtick.minor.visible'] = True (for horizontal cbar)
308: (4)                rcParams['ytick.minor.visible'] = True (for vertical cbar).
309: (4)                Using cbar.minorticks_on() ensures that the minor ticks
310: (4)                don't overflow into the extend regions of the colorbar.
311: (4)                """
312: (4)                plt.rcParams['ytick.minor.visible'] = True
313: (4)                plt.rcParams['xtick.minor.visible'] = True
314: (4)                vmin, vmax = 0.4, 2.6
315: (4)                fig, ax = plt.subplots()
316: (4)                im = ax.pcolormesh([[1, 2]], vmin=vmin, vmax=vmax)
317: (4)                cbar = fig.colorbar(im, extend='both', orientation='vertical')
318: (4)                assert cbar.ax.yaxis.get_minorticklocs()[0] >= vmin
319: (4)                assert cbar.ax.yaxis.get_minorticklocs()[-1] <= vmax
320: (4)                cbar = fig.colorbar(im, extend='both', orientation='horizontal')
321: (4)                assert cbar.ax.xaxis.get_minorticklocs()[0] >= vmin
322: (4)                assert cbar.ax.xaxis.get_minorticklocs()[-1] <= vmax
323: (0)            def test_colorbar_autoticks():
324: (4)                with rc_context({'_internal.classic_mode': False}):
325: (8)                    fig, ax = plt.subplots(2, 1)
326: (8)                    x = np.arange(-3.0, 4.001)
327: (8)                    y = np.arange(-4.0, 3.001)
328: (8)                    X, Y = np.meshgrid(x, y)
329: (8)                    Z = X * Y
330: (8)                    Z = Z[:-1, :-1]
331: (8)                    pcm = ax[0].pcolormesh(X, Y, Z)
332: (8)                    cbar = fig.colorbar(pcm, ax=ax[0], extend='both',
333: (28)                                       orientation='vertical')
334: (8)                    pcm = ax[1].pcolormesh(X, Y, Z)
335: (8)                    cbar2 = fig.colorbar(pcm, ax=ax[1], extend='both',
336: (29)                                        orientation='vertical', shrink=0.4)
337: (8)                    np.testing.assert_almost_equal(cbar.ax.yaxis.get_ticklocs(),
338: (39)                                                  np.arange(-15, 16, 5))
339: (8)                    np.testing.assert_almost_equal(cbar2.ax.yaxis.get_ticklocs(),
340: (39)                                                  np.arange(-20, 21, 10))
341: (0)            def test_colorbar_autotickslog():
342: (4)                with rc_context({'_internal.classic_mode': False}):
343: (8)                    fig, ax = plt.subplots(2, 1)
344: (8)                    x = np.arange(-3.0, 4.001)
345: (8)                    y = np.arange(-4.0, 3.001)
346: (8)                    X, Y = np.meshgrid(x, y)
347: (8)                    Z = X * Y
348: (8)                    Z = Z[:-1, :-1]
349: (8)                    pcm = ax[0].pcolormesh(X, Y, 10**Z, norm=LogNorm())
350: (8)                    cbar = fig.colorbar(pcm, ax=ax[0], extend='both',
351: (28)                                       orientation='vertical')
352: (8)                    pcm = ax[1].pcolormesh(X, Y, 10**Z, norm=LogNorm())
353: (8)                    cbar2 = fig.colorbar(pcm, ax=ax[1], extend='both',
354: (29)                                        orientation='vertical', shrink=0.4)
355: (8)                    np.testing.assert_almost_equal(cbar.ax.yaxis.get_ticklocs(),
356: (39)                                                  10**np.arange(-16., 16.2, 4.))
357: (8)                    np.testing.assert_almost_equal(cbar2.ax.yaxis.get_ticklocs(),
358: (39)                                                  10**np.arange(-24., 25., 12.))
359: (0)            def test_colorbar_get_ticks():
360: (4)                plt.figure()
361: (4)                data = np.arange(1200).reshape(30, 40)
362: (4)                levels = [0, 200, 400, 600, 800, 1000, 1200]
363: (4)                plt.contourf(data, levels=levels)
```

```
(4)                 userTicks = plt.colorbar(ticks=[0, 600, 1200])
(4)                 assert userTicks.get_ticks().tolist() == [0, 600, 1200]
(4)                 userTicks.set_ticks([600, 700, 800])
(4)                 assert userTicks.get_ticks().tolist() == [600, 700, 800]
(4)                 defTicks = plt.colorbar(orientation='horizontal')
(4)                 np.testing.assert_allclose(defTicks.get_ticks().tolist(), levels)
(4)                 fig, ax = plt.subplots()
(4)                 x = np.arange(-3.0, 4.001)
(4)                 y = np.arange(-4.0, 3.001)
(4)                 X, Y = np.meshgrid(x, y)
(4)                 Z = X * Y
(4)                 Z = Z[:-1, :-1]
(4)                 pcm = ax.pcolormesh(X, Y, Z)
(4)                 cbar = fig.colorbar(pcm, ax=ax, extend='both',
(24)                                    orientation='vertical')
(4)                 ticks = cbar.get_ticks()
(4)                 np.testing.assert_allclose(ticks, np.arange(-15, 16, 5))
(4)                 assert len(cbar.get_ticks(minor=True)) == 0
(0)             @pytest.mark.parametrize("extend", ['both', 'min', 'max'])
(0)             def test_colorbar_lognorm_extension(extend):
(4)                 f, ax = plt.subplots()
(4)                 cb = Colorbar(ax, norm=LogNorm(vmin=0.1, vmax=1000.0),
(18)                              orientation='vertical', extend=extend)
(4)                 assert cb._values[0] >= 0.0
(0)             def test_colorbar_powernorm_extension():
(4)                 fig, ax = plt.subplots()
(4)                 Colorbar(ax, norm=PowerNorm(gamma=0.5, vmin=0.0, vmax=1.0),
(13)                         orientation='vertical', extend='both')
(0)             def test_colorbar_axes_kw():
(4)                 plt.figure()
(4)                 plt.imshow([[1, 2], [3, 4]])
(4)                 plt.colorbar(orientation='horizontal', fraction=0.2, pad=0.2, shrink=0.5,
(17)                             aspect=10, anchor=(0., 0.), panchor=(0., 1.))
(0)             def test_colorbar_log_minortick_labels():
(4)                 with rc_context({'_internal.classic_mode': False}):
(8)                     fig, ax = plt.subplots()
(8)                     pcm = ax.imshow([[10000, 50000]], norm=LogNorm())
(8)                     cb = fig.colorbar(pcm)
(8)                     fig.canvas.draw()
(8)                     lb = [l.get_text() for l in cb.ax.yaxis.get_ticklabels(which='both')]
(8)                     expected = [r'$\mathdefault{10^{4}}$',
(20)                                r'$\mathdefault{2\times10^{4}}$',
(20)                                r'$\mathdefault{3\times10^{4}}$',
(20)                                r'$\mathdefault{4\times10^{4}}$']
(8)                     for exp in expected:
(12)                        assert exp in lb
(0)             def test_colorbar_renorm():
(4)                 x, y = np.ogrid[-4:4:31j, -4:4:31j]
(4)                 z = 120000*np.exp(-x**2 - y**2)
(4)                 fig, ax = plt.subplots()
(4)                 im = ax.imshow(z)
(4)                 cbar = fig.colorbar(im)
(4)                 np.testing.assert_allclose(cbar.ax.yaxis.get_majorticklocs(),
(31)                                           np.arange(0, 120000.1, 20000))
(4)                 cbar.set_ticks([1, 2, 3])
(4)                 assert isinstance(cbar.locator, FixedLocator)
(4)                 norm = LogNorm(z.min(), z.max())
(4)                 im.set_norm(norm)
(4)                 np.testing.assert_allclose(cbar.ax.yaxis.get_majorticklocs(),
(31)                                           np.logspace(-10, 7, 18))
(4)                 assert np.isclose(cbar.vmin, z.min())
(4)                 cbar.set_ticks([1, 2, 3])
(4)                 assert isinstance(cbar.locator, FixedLocator)
(4)                 np.testing.assert_allclose(cbar.ax.yaxis.get_majorticklocs(),
(31)                                           [1.0, 2.0, 3.0])
(4)                 norm = LogNorm(z.min() * 1000, z.max() * 1000)
(4)                 im.set_norm(norm)
(4)                 assert np.isclose(cbar.vmin, z.min() * 1000)
(4)                 assert np.isclose(cbar.vmax, z.max() * 1000)
```

```
(0)            @pytest.mark.parametrize('fmt', ['%4.2e', '{x:.2e}'])
(0)            def test_colorbar_format(fmt):
(4)                x, y = np.ogrid[-4:4:31j, -4:4:31j]
(4)                z = 120000*np.exp(-x**2 - y**2)
(4)                fig, ax = plt.subplots()
(4)                im = ax.imshow(z)
(4)                cbar = fig.colorbar(im, format=fmt)
(4)                fig.canvas.draw()
(4)                assert cbar.ax.yaxis.get_ticklabels()[4].get_text() == '8.00e+04'
(4)                im.set_clim([4, 200])
(4)                fig.canvas.draw()
(4)                assert cbar.ax.yaxis.get_ticklabels()[4].get_text() == '2.00e+02'
(4)                im.set_norm(LogNorm(vmin=0.1, vmax=10))
(4)                fig.canvas.draw()
(4)                assert (cbar.ax.yaxis.get_ticklabels()[0].get_text() ==
(12)                       '$\\mathdefault{10^{-2}}$')
(0)            def test_colorbar_scale_reset():
(4)                x, y = np.ogrid[-4:4:31j, -4:4:31j]
(4)                z = 120000*np.exp(-x**2 - y**2)
(4)                fig, ax = plt.subplots()
(4)                pcm = ax.pcolormesh(z, cmap='RdBu_r', rasterized=True)
(4)                cbar = fig.colorbar(pcm, ax=ax)
(4)                cbar.outline.set_edgecolor('red')
(4)                assert cbar.ax.yaxis.get_scale() == 'linear'
(4)                pcm.set_norm(LogNorm(vmin=1, vmax=100))
(4)                assert cbar.ax.yaxis.get_scale() == 'log'
(4)                pcm.set_norm(Normalize(vmin=-20, vmax=20))
(4)                assert cbar.ax.yaxis.get_scale() == 'linear'
(4)                assert cbar.outline.get_edgecolor() == mcolors.to_rgba('red')
(4)                pcm.norm = LogNorm()
(4)                assert pcm.norm.vmin == z.min()
(4)                assert pcm.norm.vmax == z.max()
(0)            def test_colorbar_get_ticks_2():
(4)                plt.rcParams['_internal.classic_mode'] = False
(4)                fig, ax = plt.subplots()
(4)                pc = ax.pcolormesh([[.05, .95]])
(4)                cb = fig.colorbar(pc)
(4)                np.testing.assert_allclose(cb.get_ticks(), [0., 0.2, 0.4, 0.6, 0.8, 1.0])
(0)            def test_colorbar_inverted_ticks():
(4)                fig, axs = plt.subplots(2)
(4)                ax = axs[0]
(4)                pc = ax.pcolormesh(10**np.arange(1, 5).reshape(2, 2), norm=LogNorm())
(4)                cbar = fig.colorbar(pc, ax=ax, extend='both')
(4)                ticks = cbar.get_ticks()
(4)                cbar.ax.invert_yaxis()
(4)                np.testing.assert_allclose(ticks, cbar.get_ticks())
(4)                ax = axs[1]
(4)                pc = ax.pcolormesh(np.arange(1, 5).reshape(2, 2))
(4)                cbar = fig.colorbar(pc, ax=ax, extend='both')
(4)                cbar.minorticks_on()
(4)                ticks = cbar.get_ticks()
(4)                minorticks = cbar.get_ticks(minor=True)
(4)                assert isinstance(minorticks, np.ndarray)
(4)                cbar.ax.invert_yaxis()
(4)                np.testing.assert_allclose(ticks, cbar.get_ticks())
(4)                np.testing.assert_allclose(minorticks, cbar.get_ticks(minor=True))
(0)            def test_mappable_no_alpha():
(4)                fig, ax = plt.subplots()
(4)                sm = cm.ScalarMappable(norm=mcolors.Normalize(), cmap='viridis')
(4)                fig.colorbar(sm, ax=ax)
(4)                sm.set_cmap('plasma')
(4)                plt.draw()
(0)            def test_mappable_2d_alpha():
(4)                fig, ax = plt.subplots()
(4)                x = np.arange(1, 5).reshape(2, 2)/4
(4)                pc = ax.pcolormesh(x, alpha=x)
(4)                cb = fig.colorbar(pc, ax=ax)
(4)                assert cb.alpha is None
(4)                assert pc.get_alpha() is x
```

```
(4)                 fig.draw_without_rendering()
(0)             def test_colorbar_label():
(4)                 """
(4)                 Test the label parameter. It should just be mapped to the xlabel/ylabel of
(4)                 the axes, depending on the orientation.
(4)                 """
(4)                 fig, ax = plt.subplots()
(4)                 im = ax.imshow([[1, 2], [3, 4]])
(4)                 cbar = fig.colorbar(im, label='cbar')
(4)                 assert cbar.ax.get_ylabel() == 'cbar'
(4)                 cbar.set_label(None)
(4)                 assert cbar.ax.get_ylabel() == ''
(4)                 cbar.set_label('cbar 2')
(4)                 assert cbar.ax.get_ylabel() == 'cbar 2'
(4)                 cbar2 = fig.colorbar(im, label=None)
(4)                 assert cbar2.ax.get_ylabel() == ''
(4)                 cbar3 = fig.colorbar(im, orientation='horizontal', label='horizontal
cbar')
(4)                 assert cbar3.ax.get_xlabel() == 'horizontal cbar'
(0)             @image_comparison(['colorbar_keeping_xlabel.png'], style='mpl20')
(0)             def test_keeping_xlabel():
(4)                 arr = np.arange(25).reshape((5, 5))
(4)                 fig, ax = plt.subplots()
(4)                 im = ax.imshow(arr)
(4)                 cbar = plt.colorbar(im)
(4)                 cbar.ax.set_xlabel('Visible Xlabel')
(4)                 cbar.set_label('YLabel')
(0)             @pytest.mark.parametrize("clim", [(-20000, 20000), (-32768, 0)])
(0)             def test_colorbar_int(clim):
(4)                 fig, ax = plt.subplots()
(4)                 im = ax.imshow([[*map(np.int16, clim)]])
(4)                 fig.colorbar(im)
(4)                 assert (im.norm.vmin, im.norm.vmax) == clim
(0)             def test_anchored_cbar_position_using_specgrid():
(4)                 data = np.arange(1200).reshape(30, 40)
(4)                 levels = [0, 200, 400, 600, 800, 1000, 1200]
(4)                 shrink = 0.5
(4)                 anchor_y = 0.3
(4)                 fig, ax = plt.subplots()
(4)                 cs = ax.contourf(data, levels=levels)
(4)                 cbar = plt.colorbar(
(12)                        cs, ax=ax, use_gridspec=True,
(12)                        location='right', anchor=(1, anchor_y), shrink=shrink)
(4)                 x0, y0, x1, y1 = ax.get_position().extents
(4)                 cx0, cy0, cx1, cy1 = cbar.ax.get_position().extents
(4)                 p0 = (y1 - y0) * anchor_y + y0
(4)                 np.testing.assert_allclose(
(12)                        [cy1, cy0],
(12)                        [y1 * shrink + (1 - shrink) * p0, p0 * (1 - shrink) + y0 *
shrink])
(4)                 fig, ax = plt.subplots()
(4)                 cs = ax.contourf(data, levels=levels)
(4)                 cbar = plt.colorbar(
(12)                        cs, ax=ax, use_gridspec=True,
(12)                        location='left', anchor=(1, anchor_y), shrink=shrink)
(4)                 x0, y0, x1, y1 = ax.get_position().extents
(4)                 cx0, cy0, cx1, cy1 = cbar.ax.get_position().extents
(4)                 p0 = (y1 - y0) * anchor_y + y0
(4)                 np.testing.assert_allclose(
(12)                        [cy1, cy0],
(12)                        [y1 * shrink + (1 - shrink) * p0, p0 * (1 - shrink) + y0 *
shrink])
(4)                 shrink = 0.5
(4)                 anchor_x = 0.3
(4)                 fig, ax = plt.subplots()
(4)                 cs = ax.contourf(data, levels=levels)
(4)                 cbar = plt.colorbar(
(12)                        cs, ax=ax, use_gridspec=True,
(12)                        location='top', anchor=(anchor_x, 1), shrink=shrink)
```

```
(4)                 x0, y0, x1, y1 = ax.get_position().extents
(4)                 cx0, cy0, cx1, cy1 = cbar.ax.get_position().extents
(4)                 p0 = (x1 - x0) * anchor_x + x0
(4)                 np.testing.assert_allclose(
(12)                        [cx1, cx0],
(12)                        [x1 * shrink + (1 - shrink) * p0, p0 * (1 - shrink) + x0 *
shrink])
(4)                 shrink = 0.5
(4)                 anchor_x = 0.3
(4)                 fig, ax = plt.subplots()
(4)                 cs = ax.contourf(data, levels=levels)
(4)                 cbar = plt.colorbar(
(12)                        cs, ax=ax, use_gridspec=True,
(12)                        location='bottom', anchor=(anchor_x, 1), shrink=shrink)
(4)                 x0, y0, x1, y1 = ax.get_position().extents
(4)                 cx0, cy0, cx1, cy1 = cbar.ax.get_position().extents
(4)                 p0 = (x1 - x0) * anchor_x + x0
(4)                 np.testing.assert_allclose(
(12)                        [cx1, cx0],
(12)                        [x1 * shrink + (1 - shrink) * p0, p0 * (1 - shrink) + x0 *
shrink])
(0)             @image_comparison(['colorbar_change_lim_scale.png'], remove_text=True,
(18)                            style='mpl20')
(0)             def test_colorbar_change_lim_scale():
(4)                 fig, ax = plt.subplots(1, 2, constrained_layout=True)
(4)                 pc = ax[0].pcolormesh(np.arange(100).reshape(10, 10)+1)
(4)                 cb = fig.colorbar(pc, ax=ax[0], extend='both')
(4)                 cb.ax.set_yscale('log')
(4)                 pc = ax[1].pcolormesh(np.arange(100).reshape(10, 10)+1)
(4)                 cb = fig.colorbar(pc, ax=ax[1], extend='both')
(4)                 cb.ax.set_ylim([20, 90])
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_axes_handles_same_functions(fig_ref, fig_test):
(4)                 for nn, fig in enumerate([fig_ref, fig_test]):
(8)                     ax = fig.add_subplot()
(8)                     pc = ax.pcolormesh(np.ones(300).reshape(10, 30))
(8)                     cax = fig.add_axes([0.9, 0.1, 0.03, 0.8])
(8)                     cb = fig.colorbar(pc, cax=cax)
(8)                     if nn == 0:
(12)                        caxx = cax
(8)                     else:
(12)                        caxx = cb.ax
(8)                     caxx.set_yticks(np.arange(0, 20))
(8)                     caxx.set_yscale('log')
(8)                     caxx.set_position([0.92, 0.1, 0.02, 0.7])
(0)             def test_inset_colorbar_layout():
(4)                 fig, ax = plt.subplots(constrained_layout=True, figsize=(3, 6))
(4)                 pc = ax.imshow(np.arange(100).reshape(10, 10))
(4)                 cax = ax.inset_axes([1.02, 0.1, 0.03, 0.8])
(4)                 cb = fig.colorbar(pc, cax=cax)
(4)                 fig.draw_without_rendering()
(4)                 np.testing.assert_allclose(cb.ax.get_position().bounds,
(31)                                           [0.87, 0.342, 0.0237, 0.315], atol=0.01)
(4)                 assert cb.ax in ax.child_axes
(0)             @image_comparison(['colorbar_twoslope.png'], remove_text=True,
(18)                            style='mpl20')
(0)             def test_twoslope_colorbar():
(4)                 fig, ax = plt.subplots()
(4)                 norm = mcolors.TwoSlopeNorm(20, 5, 95)
(4)                 pc = ax.pcolormesh(np.arange(1, 11), np.arange(1, 11),
(23)                                   np.arange(100).reshape(10, 10),
(23)                                   norm=norm, cmap='RdBu_r')
(4)                 fig.colorbar(pc)
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_remove_cb_whose_mappable_has_no_figure(fig_ref, fig_test):
(4)                 ax = fig_test.add_subplot()
(4)                 cb = fig_test.colorbar(cm.ScalarMappable(), cax=ax)
(4)                 cb.remove()
(0)             def test_aspects():
```

```
(4)                 fig, ax = plt.subplots(3, 2, figsize=(8, 8))
(4)                 aspects = [20, 20, 10]
(4)                 extends = ['neither', 'both', 'both']
(4)                 cb = [[None, None, None], [None, None, None]]
(4)                 for nn, orient in enumerate(['vertical', 'horizontal']):
(8)                     for mm, (aspect, extend) in enumerate(zip(aspects, extends)):
(12)                        pc = ax[mm, nn].pcolormesh(np.arange(100).reshape(10, 10))
(12)                        cb[nn][mm] = fig.colorbar(pc, ax=ax[mm, nn], orientation=orient,
(38)                                                  aspect=aspect, extend=extend)
(4)                 fig.draw_without_rendering()
(4)                 np.testing.assert_almost_equal(cb[0][1].ax.get_position().height,
(35)                                               cb[0][0].ax.get_position().height * 0.9,
(35)                                               decimal=2)
(4)                 np.testing.assert_almost_equal(cb[1][1].ax.get_position().width,
(35)                                               cb[1][0].ax.get_position().width * 0.9,
(35)                                               decimal=2)
(4)                 pos = cb[0][0].ax.get_position(original=False)
(4)                 np.testing.assert_almost_equal(pos.height, pos.width * 20, decimal=2)
(4)                 pos = cb[1][0].ax.get_position(original=False)
(4)                 np.testing.assert_almost_equal(pos.height * 20, pos.width, decimal=2)
(4)                 np.testing.assert_almost_equal(
(8)                     cb[0][0].ax.get_position(original=False).width * 2,
(8)                     cb[0][2].ax.get_position(original=False).width, decimal=2)
(4)                 np.testing.assert_almost_equal(
(8)                     cb[1][0].ax.get_position(original=False).height * 2,
(8)                     cb[1][2].ax.get_position(original=False).height, decimal=2)
(0)             @image_comparison(['proportional_colorbars.png'], remove_text=True,
(18)                              style='mpl20')
(0)             def test_proportional_colorbars():
(4)                 x = y = np.arange(-3.0, 3.01, 0.025)
(4)                 X, Y = np.meshgrid(x, y)
(4)                 Z1 = np.exp(-X**2 - Y**2)
(4)                 Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
(4)                 Z = (Z1 - Z2) * 2
(4)                 levels = [-1.25, -0.5, -0.125, 0.125, 0.5, 1.25]
(4)                 cmap = mcolors.ListedColormap(
(8)                     ['0.3', '0.5', 'white', 'lightblue', 'steelblue'])
(4)                 cmap.set_under('darkred')
(4)                 cmap.set_over('crimson')
(4)                 norm = mcolors.BoundaryNorm(levels, cmap.N)
(4)                 extends = ['neither', 'both']
(4)                 spacings = ['uniform', 'proportional']
(4)                 fig, axs = plt.subplots(2, 2)
(4)                 for i in range(2):
(8)                     for j in range(2):
(12)                        CS3 = axs[i, j].contourf(X, Y, Z, levels, cmap=cmap, norm=norm,
(37)                                                 extend=extends[i])
(12)                        fig.colorbar(CS3, spacing=spacings[j], ax=axs[i, j])
(0)             @image_comparison(['extend_drawedges.png'], remove_text=True, style='mpl20')
(0)             def test_colorbar_extend_drawedges():
(4)                 params = [
(8)                     ('both', 1, [[[1.1, 0], [1.1, 1]],
(21)                                 [[2, 0], [2, 1]],
(21)                                 [[2.9, 0], [2.9, 1]]]),
(8)                     ('min', 0, [[[1.1, 0], [1.1, 1]],
(20)                                [[2, 0], [2, 1]]]),
(8)                     ('max', 0, [[[2, 0], [2, 1]],
(20)                                [[2.9, 0], [2.9, 1]]]),
(8)                     ('neither', -1, [[[2, 0], [2, 1]]]),
(4)                 ]
(4)                 plt.rcParams['axes.linewidth'] = 2
(4)                 fig = plt.figure(figsize=(10, 4))
(4)                 subfigs = fig.subfigures(1, 2)
(4)                 for orientation, subfig in zip(['horizontal', 'vertical'], subfigs):
(8)                     if orientation == 'horizontal':
(12)                        axs = subfig.subplots(4, 1)
(8)                     else:
(12)                        axs = subfig.subplots(1, 4)
(8)                     fig.subplots_adjust(left=0.05, bottom=0.05, right=0.95, top=0.95)
```

```
(8)                     for ax, (extend, coloroffset, res) in zip(axs, params):
(12)                        cmap = mpl.colormaps["viridis"]
(12)                        bounds = np.arange(5)
(12)                        nb_colors = len(bounds) + coloroffset
(12)                        colors = cmap(np.linspace(100, 255, nb_colors).astype(int))
(12)                        cmap, norm = mcolors.from_levels_and_colors(bounds, colors,
(56)                                                                    extend=extend)
(12)                        cbar = Colorbar(ax, cmap=cmap, norm=norm, orientation=orientation,
(28)                                        drawedges=True)
(12)                        if orientation == 'horizontal':
(16)                            ax.set_xlim(1.1, 2.9)
(12)                        else:
(16)                            ax.set_ylim(1.1, 2.9)
(16)                            res = np.array(res)[:, :, [1, 0]]
(12)                        np.testing.assert_array_equal(cbar.dividers.get_segments(), res)
(0)             @image_comparison(['contourf_extend_patches.png'], remove_text=True,
(18)                              style='mpl20')
(0)             def test_colorbar_contourf_extend_patches():
(4)                 params = [
(8)                     ('both', 5, ['\\', '//']),
(8)                     ('min', 7, ['+']),
(8)                     ('max', 2, ['|', '-', '/', '\\', '//']),
(8)                     ('neither', 10, ['//', '\\', '||']),
(4)                 ]
(4)                 plt.rcParams['axes.linewidth'] = 2
(4)                 fig = plt.figure(figsize=(10, 4))
(4)                 subfigs = fig.subfigures(1, 2)
(4)                 fig.subplots_adjust(left=0.05, bottom=0.05, right=0.95, top=0.95)
(4)                 x = np.linspace(-2, 3, 50)
(4)                 y = np.linspace(-2, 3, 30)
(4)                 z = np.cos(x[np.newaxis, :]) + np.sin(y[:, np.newaxis])
(4)                 cmap = mpl.colormaps["viridis"]
(4)                 for orientation, subfig in zip(['horizontal', 'vertical'], subfigs):
(8)                     axs = subfig.subplots(2, 2).ravel()
(8)                     for ax, (extend, levels, hatches) in zip(axs, params):
(12)                        cs = ax.contourf(x, y, z, levels, hatches=hatches,
(29)                                         cmap=cmap, extend=extend)
(12)                        subfig.colorbar(cs, ax=ax, orientation=orientation, fraction=0.4,
(28)                                        extendfrac=0.2, aspect=5)
(0)             def test_negative_boundarynorm():
(4)                 fig, ax = plt.subplots(figsize=(1, 3))
(4)                 cmap = mpl.colormaps["viridis"]
(4)                 clevs = np.arange(-94, -85)
(4)                 norm = BoundaryNorm(clevs, cmap.N)
(4)                 cb = fig.colorbar(cm.ScalarMappable(cmap=cmap, norm=norm), cax=ax)
(4)                 np.testing.assert_allclose(cb.ax.get_ylim(), [clevs[0], clevs[-1]])
(4)                 np.testing.assert_allclose(cb.ax.get_yticks(), clevs)
(4)                 clevs = np.arange(85, 94)
(4)                 norm = BoundaryNorm(clevs, cmap.N)
(4)                 cb = fig.colorbar(cm.ScalarMappable(cmap=cmap, norm=norm), cax=ax)
(4)                 np.testing.assert_allclose(cb.ax.get_ylim(), [clevs[0], clevs[-1]])
(4)                 np.testing.assert_allclose(cb.ax.get_yticks(), clevs)
(4)                 clevs = np.arange(-3, 3)
(4)                 norm = BoundaryNorm(clevs, cmap.N)
(4)                 cb = fig.colorbar(cm.ScalarMappable(cmap=cmap, norm=norm), cax=ax)
(4)                 np.testing.assert_allclose(cb.ax.get_ylim(), [clevs[0], clevs[-1]])
(4)                 np.testing.assert_allclose(cb.ax.get_yticks(), clevs)
(4)                 clevs = np.arange(-8, 1)
(4)                 norm = BoundaryNorm(clevs, cmap.N)
(4)                 cb = fig.colorbar(cm.ScalarMappable(cmap=cmap, norm=norm), cax=ax)
(4)                 np.testing.assert_allclose(cb.ax.get_ylim(), [clevs[0], clevs[-1]])
(4)                 np.testing.assert_allclose(cb.ax.get_yticks(), clevs)
(0)             def test_centerednorm():
(4)                 fig, ax = plt.subplots(figsize=(1, 3))
(4)                 norm = mcolors.CenteredNorm()
(4)                 mappable = ax.pcolormesh(np.zeros((3, 3)), norm=norm)
(4)                 fig.colorbar(mappable)
(4)                 assert (norm.vmin, norm.vmax) == (-0.1, 0.1)
(0)             @image_comparison(['nonorm_colorbars.svg'], style='mpl20')
```

```
(0)  def test_nonorm():
(4)      plt.rcParams['svg.fonttype'] = 'none'
(4)      data = [1, 2, 3, 4, 5]
(4)      fig, ax = plt.subplots(figsize=(6, 1))
(4)      fig.subplots_adjust(bottom=0.5)
(4)      norm = NoNorm(vmin=min(data), vmax=max(data))
(4)      cmap = mpl.colormaps["viridis"].resampled(len(data))
(4)      mappable = cm.ScalarMappable(norm=norm, cmap=cmap)
(4)      cbar = fig.colorbar(mappable, cax=ax, orientation="horizontal")
(0)  @image_comparison(['test_boundaries.png'], remove_text=True,
(18)                   style='mpl20')
(0)  def test_boundaries():
(4)      np.random.seed(seed=19680808)
(4)      fig, ax = plt.subplots(figsize=(2, 2))
(4)      pc = ax.pcolormesh(np.random.randn(10, 10), cmap='RdBu_r')
(4)      cb = fig.colorbar(pc, ax=ax, boundaries=np.linspace(-3, 3, 7))
(0)  def test_colorbar_no_warning_rcparams_grid_true():
(4)      plt.rcParams['axes.grid'] = True
(4)      fig, ax = plt.subplots()
(4)      ax.grid(False)
(4)      im = ax.pcolormesh([0, 1], [0, 1], [[1]])
(4)      fig.colorbar(im)
(0)  def test_colorbar_set_formatter_locator():
(4)      fig, ax = plt.subplots()
(4)      pc = ax.pcolormesh(np.random.randn(10, 10))
(4)      cb = fig.colorbar(pc)
(4)      cb.ax.yaxis.set_major_locator(FixedLocator(np.arange(10)))
(4)      cb.ax.yaxis.set_minor_locator(FixedLocator(np.arange(0, 10, 0.2)))
(4)      assert cb.locator is cb.ax.yaxis.get_major_locator()
(4)      assert cb.minorlocator is cb.ax.yaxis.get_minor_locator()
(4)      cb.ax.yaxis.set_major_formatter(LogFormatter())
(4)      cb.ax.yaxis.set_minor_formatter(LogFormatter())
(4)      assert cb.formatter is cb.ax.yaxis.get_major_formatter()
(4)      assert cb.minorformatter is cb.ax.yaxis.get_minor_formatter()
(4)      loc = FixedLocator(np.arange(7))
(4)      cb.locator = loc
(4)      assert cb.ax.yaxis.get_major_locator() is loc
(4)      loc = FixedLocator(np.arange(0, 7, 0.1))
(4)      cb.minorlocator = loc
(4)      assert cb.ax.yaxis.get_minor_locator() is loc
(4)      fmt = LogFormatter()
(4)      cb.formatter = fmt
(4)      assert cb.ax.yaxis.get_major_formatter() is fmt
(4)      fmt = LogFormatter()
(4)      cb.minorformatter = fmt
(4)      assert cb.ax.yaxis.get_minor_formatter() is fmt
(0)  @image_comparison(['colorbar_extend_alpha.png'], remove_text=True,
(18)                   savefig_kwarg={'dpi': 40})
(0)  def test_colorbar_extend_alpha():
(4)      fig, ax = plt.subplots()
(4)      im = ax.imshow([[0, 1], [2, 3]], alpha=0.3, interpolation="none")
(4)      fig.colorbar(im, extend='both', boundaries=[0.5, 1.5, 2.5])
(0)  def test_offset_text_loc():
(4)      plt.style.use('mpl20')
(4)      fig, ax = plt.subplots()
(4)      np.random.seed(seed=19680808)
(4)      pc = ax.pcolormesh(np.random.randn(10, 10)*1e6)
(4)      cb = fig.colorbar(pc, location='right', extend='max')
(4)      fig.draw_without_rendering()
(4)      assert cb.ax.yaxis.offsetText.get_position()[1] > ax.bbox.y1
(0)  def test_title_text_loc():
(4)      plt.style.use('mpl20')
(4)      fig, ax = plt.subplots()
(4)      np.random.seed(seed=19680808)
(4)      pc = ax.pcolormesh(np.random.randn(10, 10))
(4)      cb = fig.colorbar(pc, location='right', extend='max')
(4)      cb.ax.set_title('Aardvark')
(4)      fig.draw_without_rendering()
(4)      assert (cb.ax.title.get_window_extent(fig.canvas.get_renderer()).ymax >
```

```
842: (12)                        cb.ax.spines['outline'].get_window_extent().ymax)
843: (0)            @check_figures_equal(extensions=["png"])
844: (0)            def test_passing_location(fig_ref, fig_test):
845: (4)                ax_ref = fig_ref.add_subplot()
846: (4)                im = ax_ref.imshow([[0, 1], [2, 3]])
847: (4)                ax_ref.figure.colorbar(im, cax=ax_ref.inset_axes([0, 1.05, 1, 0.05]),
848: (27)                                       orientation="horizontal", ticklocation="top")
849: (4)                ax_test = fig_test.add_subplot()
850: (4)                im = ax_test.imshow([[0, 1], [2, 3]])
851: (4)                ax_test.figure.colorbar(im, cax=ax_test.inset_axes([0, 1.05, 1, 0.05]),
852: (28)                                        location="top")
853: (0)            @pytest.mark.parametrize("kwargs,error,message", [
854: (4)                ({'location': 'top', 'orientation': 'vertical'}, TypeError,
855: (5)                 "location and orientation are mutually exclusive"),
856: (4)                ({'location': 'top', 'orientation': 'vertical', 'cax': True}, TypeError,
857: (5)                 "location and orientation are mutually exclusive"),  # Different to above
858: (4)                ({'ticklocation': 'top', 'orientation': 'vertical', 'cax': True},
859: (5)                 ValueError, "'top' is not a valid value for position"),
860: (4)                ({'location': 'top', 'extendfrac': (0, None)}, ValueError,
861: (5)                 "invalid value for extendfrac"),
862: (4)                ])
863: (0)            def test_colorbar_errors(kwargs, error, message):
864: (4)                fig, ax = plt.subplots()
865: (4)                im = ax.imshow([[0, 1], [2, 3]])
866: (4)                if kwargs.get('cax', None) is True:
867: (8)                    kwargs['cax'] = ax.inset_axes([0, 1.05, 1, 0.05])
868: (4)                with pytest.raises(error, match=message):
869: (8)                    fig.colorbar(im, **kwargs)
870: (0)            def test_colorbar_axes_parmeters():
871: (4)                fig, ax = plt.subplots(2)
872: (4)                im = ax[0].imshow([[0, 1], [2, 3]])
873: (4)                fig.colorbar(im, ax=ax)
874: (4)                fig.colorbar(im, ax=ax[0])
875: (4)                fig.colorbar(im, ax=[_ax for _ax in ax])
876: (4)                fig.colorbar(im, ax=(ax[0], ax[1]))
877: (4)                fig.colorbar(im, ax={i: _ax for i, _ax in enumerate(ax)}.values())
878: (4)                fig.draw_without_rendering()
879: (0)            def test_colorbar_wrong_figure():
880: (4)                fig_tl = plt.figure(layout="tight")
881: (4)                fig_cl = plt.figure(layout="constrained")
882: (4)                im = fig_cl.add_subplot().imshow([[0, 1]])
883: (4)                with pytest.warns(UserWarning, match="different Figure"):
884: (8)                    fig_tl.colorbar(im)
885: (4)                fig_tl.draw_without_rendering()
886: (4)                fig_cl.draw_without_rendering()
887: (0)            def test_colorbar_format_string_and_old():
888: (4)                plt.imshow([[0, 1]])
889: (4)                cb = plt.colorbar(format="{x}%")
890: (4)                assert isinstance(cb._formatter, StrMethodFormatter)
```

File 173 - test_colors.py:

```
1: (0)              import copy
2: (0)              import itertools
3: (0)              import unittest.mock
4: (0)              from io import BytesIO
5: (0)              import numpy as np
6: (0)              from PIL import Image
7: (0)              import pytest
8: (0)              import base64
9: (0)              from numpy.testing import assert_array_equal, assert_array_almost_equal
10: (0)             from matplotlib import cbook, cm
11: (0)             import matplotlib
12: (0)             import matplotlib as mpl
13: (0)             import matplotlib.colors as mcolors
14: (0)             import matplotlib.colorbar as mcolorbar
15: (0)             import matplotlib.pyplot as plt
```

```
 16: (0)             import matplotlib.scale as mscale
 17: (0)             from matplotlib.rcsetup import cycler
 18: (0)             from matplotlib.testing.decorators import image_comparison,
check_figures_equal
 19: (0)             from matplotlib.colors import is_color_like, to_rgba_array
 20: (0)             @pytest.mark.parametrize('N, result', [
 21: (4)                 (5, [1, .6, .2, .1, 0]),
 22: (4)                 (2, [1, 0]),
 23: (4)                 (1, [0]),
 24: (0)             ])
 25: (0)             def test_create_lookup_table(N, result):
 26: (4)                 data = [(0.0, 1.0, 1.0), (0.5, 0.2, 0.2), (1.0, 0.0, 0.0)]
 27: (4)                 assert_array_almost_equal(mcolors._create_lookup_table(N, data), result)
 28: (0)             @pytest.mark.parametrize("dtype", [np.uint8, int, np.float16, float])
 29: (0)             def test_index_dtype(dtype):
 30: (4)                 cm = mpl.colormaps["viridis"]
 31: (4)                 assert_array_equal(cm(dtype(0)), cm(0))
 32: (0)             def test_resampled():
 33: (4)                 """
 34: (4)                 GitHub issue #6025 pointed to incorrect ListedColormap.resampled;
 35: (4)                 here we test the method for LinearSegmentedColormap as well.
 36: (4)                 """
 37: (4)                 n = 101
 38: (4)                 colorlist = np.empty((n, 4), float)
 39: (4)                 colorlist[:, 0] = np.linspace(0, 1, n)
 40: (4)                 colorlist[:, 1] = 0.2
 41: (4)                 colorlist[:, 2] = np.linspace(1, 0, n)
 42: (4)                 colorlist[:, 3] = 0.7
 43: (4)                 lsc = mcolors.LinearSegmentedColormap.from_list('lsc', colorlist)
 44: (4)                 lc = mcolors.ListedColormap(colorlist)
 45: (4)                 for cmap in [lsc, lc]:
 46: (8)                     cmap.set_under('r')
 47: (8)                     cmap.set_over('g')
 48: (8)                     cmap.set_bad('b')
 49: (4)                 lsc3 = lsc.resampled(3)
 50: (4)                 lc3 = lc.resampled(3)
 51: (4)                 expected = np.array([[0.0, 0.2, 1.0, 0.7],
 52: (25)                                     [0.5, 0.2, 0.5, 0.7],
 53: (25)                                     [1.0, 0.2, 0.0, 0.7]], float)
 54: (4)                 assert_array_almost_equal(lsc3([0, 0.5, 1]), expected)
 55: (4)                 assert_array_almost_equal(lc3([0, 0.5, 1]), expected)
 56: (4)                 assert_array_almost_equal(lsc(np.inf), lsc3(np.inf))
 57: (4)                 assert_array_almost_equal(lsc(-np.inf), lsc3(-np.inf))
 58: (4)                 assert_array_almost_equal(lsc(np.nan), lsc3(np.nan))
 59: (4)                 assert_array_almost_equal(lc(np.inf), lc3(np.inf))
 60: (4)                 assert_array_almost_equal(lc(-np.inf), lc3(-np.inf))
 61: (4)                 assert_array_almost_equal(lc(np.nan), lc3(np.nan))
 62: (0)             def test_colormaps_get_cmap():
 63: (4)                 cr = mpl.colormaps
 64: (4)                 assert cr.get_cmap('plasma') == cr["plasma"]
 65: (4)                 assert cr.get_cmap(cr["magma"]) == cr["magma"]
 66: (4)                 assert cr.get_cmap(None) == cr[mpl.rcParams['image.cmap']]
 67: (4)                 bad_cmap = 'AardvarksAreAwkward'
 68: (4)                 with pytest.raises(ValueError, match=bad_cmap):
 69: (8)                     cr.get_cmap(bad_cmap)
 70: (4)                 with pytest.raises(TypeError, match='object'):
 71: (8)                     cr.get_cmap(object())
 72: (0)             def test_double_register_builtin_cmap():
 73: (4)                 name = "viridis"
 74: (4)                 match = f"Re-registering the builtin cmap {name!r}."
 75: (4)                 with pytest.raises(ValueError, match=match):
 76: (8)                     matplotlib.colormaps.register(mpl.colormaps[name], name=name,
force=True)
 77: (0)             def test_colormap_copy():
 78: (4)                 cmap = plt.cm.Reds
 79: (4)                 copied_cmap = copy.copy(cmap)
 80: (4)                 with np.errstate(invalid='ignore'):
 81: (8)                     ret1 = copied_cmap([-1, 0, .5, 1, np.nan, np.inf])
 82: (4)                 cmap2 = copy.copy(copied_cmap)
```

```
(4)                 cmap2.set_bad('g')
(4)                 with np.errstate(invalid='ignore'):
(8)                     ret2 = copied_cmap([-1, 0, .5, 1, np.nan, np.inf])
(4)                 assert_array_equal(ret1, ret2)
(4)                 cmap = plt.cm.Reds
(4)                 copied_cmap = cmap.copy()
(4)                 with np.errstate(invalid='ignore'):
(8)                     ret1 = copied_cmap([-1, 0, .5, 1, np.nan, np.inf])
(4)                 cmap2 = copy.copy(copied_cmap)
(4)                 cmap2.set_bad('g')
(4)                 with np.errstate(invalid='ignore'):
(8)                     ret2 = copied_cmap([-1, 0, .5, 1, np.nan, np.inf])
(4)                 assert_array_equal(ret1, ret2)
(0)             def test_colormap_equals():
(4)                 cmap = mpl.colormaps["plasma"]
(4)                 cm_copy = cmap.copy()
(4)                 assert cm_copy is not cmap
(4)                 assert cm_copy == cmap
(4)                 cm_copy.set_bad('y')
(4)                 assert cm_copy != cmap
(4)                 cm_copy._lut = cm_copy._lut[:10, :]
(4)                 assert cm_copy != cmap
(4)                 cm_copy = cmap.copy()
(4)                 cm_copy.name = "Test"
(4)                 assert cm_copy == cmap
(4)                 cm_copy = cmap.copy()
(4)                 cm_copy.colorbar_extend = not cmap.colorbar_extend
(4)                 assert cm_copy != cmap
(0)             def test_colormap_endian():
(4)                 """
(4)                 GitHub issue #1005: a bug in putmask caused erroneous
(4)                 mapping of 1.0 when input from a non-native-byteorder
(4)                 array.
(4)                 """
(4)                 cmap = mpl.colormaps["jet"]
(4)                 a = [-0.5, 0, 0.5, 1, 1.5, np.nan]
(4)                 for dt in ["f2", "f4", "f8"]:
(8)                     anative = np.ma.masked_invalid(np.array(a, dtype=dt))
(8)                     aforeign = anative.byteswap().view(anative.dtype.newbyteorder())
(8)                     assert_array_equal(cmap(anative), cmap(aforeign))
(0)             def test_colormap_invalid():
(4)                 """
(4)                 GitHub issue #9892: Handling of nan's were getting mapped to under
(4)                 rather than bad. This tests to make sure all invalid values
(4)                 (-inf, nan, inf) are mapped respectively to (under, bad, over).
(4)                 """
(4)                 cmap = mpl.colormaps["plasma"]
(4)                 x = np.array([-np.inf, -1, 0, np.nan, .7, 2, np.inf])
(4)                 expected = np.array([[0.050383, 0.029803, 0.527975, 1.],
(25)                                     [0.050383, 0.029803, 0.527975, 1.],
(25)                                     [0.050383, 0.029803, 0.527975, 1.],
(25)                                     [0.,       0.,       0.,       0.],
(25)                                     [0.949217, 0.517763, 0.295662, 1.],
(25)                                     [0.940015, 0.975158, 0.131326, 1.],
(25)                                     [0.940015, 0.975158, 0.131326, 1.]])
(4)                 assert_array_equal(cmap(x), expected)
(4)                 expected = np.array([[0.,       0.,       0.,       0.],
(25)                                     [0.050383, 0.029803, 0.527975, 1.],
(25)                                     [0.050383, 0.029803, 0.527975, 1.],
(25)                                     [0.,       0.,       0.,       0.],
(25)                                     [0.949217, 0.517763, 0.295662, 1.],
(25)                                     [0.940015, 0.975158, 0.131326, 1.],
(25)                                     [0.,       0.,       0.,       0.]])
(4)                 assert_array_equal(cmap(np.ma.masked_invalid(x)), expected)
(4)                 assert_array_equal(cmap(-np.inf), cmap(0))
(4)                 assert_array_equal(cmap(np.inf), cmap(1.0))
(4)                 assert_array_equal(cmap(np.nan), [0., 0., 0., 0.])
(0)             def test_colormap_return_types():
(4)                 """
```

```
(4)                 Make sure that tuples are returned for scalar input and
(4)                 that the proper shapes are returned for ndarrays.
(4)                 """
(4)                 cmap = mpl.colormaps["plasma"]
(4)                 assert isinstance(cmap(0.5), tuple)
(4)                 assert len(cmap(0.5)) == 4
(4)                 x = np.ones(4)
(4)                 assert cmap(x).shape == x.shape + (4,)
(4)                 x2d = np.zeros((2, 2))
(4)                 assert cmap(x2d).shape == x2d.shape + (4,)
(0)             def test_BoundaryNorm():
(4)                 """
(4)                 GitHub issue #1258: interpolation was failing with numpy
(4)                 1.7 pre-release.
(4)                 """
(4)                 boundaries = [0, 1.1, 2.2]
(4)                 vals = [-1, 0, 1, 2, 2.2, 4]
(4)                 expected = [-1, 0, 0, 1, 2, 2]
(4)                 ncolors = len(boundaries) - 1
(4)                 bn = mcolors.BoundaryNorm(boundaries, ncolors)
(4)                 assert_array_equal(bn(vals), expected)
(4)                 expected = [-1, 0, 0, 2, 3, 3]
(4)                 ncolors = len(boundaries)
(4)                 bn = mcolors.BoundaryNorm(boundaries, ncolors)
(4)                 assert_array_equal(bn(vals), expected)
(4)                 expected = [-1, 1, 1, 1, 3, 3]
(4)                 bn = mcolors.BoundaryNorm([0, 2.2], ncolors)
(4)                 assert_array_equal(bn(vals), expected)
(4)                 boundaries = [0, 1, 2, 3]
(4)                 vals = [-1, 0.1, 1.1, 2.2, 4]
(4)                 ncolors = 5
(4)                 expected = [-1, 0, 2, 4, 5]
(4)                 bn = mcolors.BoundaryNorm(boundaries, ncolors)
(4)                 assert_array_equal(bn(vals), expected)
(4)                 boundaries = [0, 1, 2]
(4)                 vals = [-1, 0.1, 1.1, 2.2]
(4)                 bn = mcolors.BoundaryNorm(boundaries, 2)
(4)                 expected = [-1, 0, 1, 2]
(4)                 for v, ex in zip(vals, expected):
(8)                     ret = bn(v)
(8)                     assert isinstance(ret, int)
(8)                     assert_array_equal(ret, ex)
(8)                     assert_array_equal(bn([v]), ex)
(4)                 bn = mcolors.BoundaryNorm(boundaries, 3)
(4)                 expected = [-1, 0, 2, 3]
(4)                 for v, ex in zip(vals, expected):
(8)                     ret = bn(v)
(8)                     assert isinstance(ret, int)
(8)                     assert_array_equal(ret, ex)
(8)                     assert_array_equal(bn([v]), ex)
(4)                 bn = mcolors.BoundaryNorm(boundaries, 3, clip=True)
(4)                 expected = [0, 0, 2, 2]
(4)                 for v, ex in zip(vals, expected):
(8)                     ret = bn(v)
(8)                     assert isinstance(ret, int)
(8)                     assert_array_equal(ret, ex)
(8)                     assert_array_equal(bn([v]), ex)
(4)                 boundaries = [0, 1.1, 2.2]
(4)                 vals = np.ma.masked_invalid([-1., np.nan, 0, 1.4, 9])
(4)                 ncolors = len(boundaries) - 1
(4)                 bn = mcolors.BoundaryNorm(boundaries, ncolors)
(4)                 expected = np.ma.masked_array([-1, -99, 0, 1, 2], mask=[0, 1, 0, 0, 0])
(4)                 assert_array_equal(bn(vals), expected)
(4)                 bn = mcolors.BoundaryNorm(boundaries, len(boundaries))
(4)                 expected = np.ma.masked_array([-1, -99, 0, 2, 3], mask=[0, 1, 0, 0, 0])
(4)                 assert_array_equal(bn(vals), expected)
(4)                 vals = np.ma.masked_invalid([np.inf, np.nan])
(4)                 assert np.all(bn(vals).mask)
(4)                 vals = np.ma.masked_invalid([np.inf])
```

```
(4)                 assert np.all(bn(vals).mask)
(4)                 with pytest.raises(ValueError, match="not compatible"):
(8)                     mcolors.BoundaryNorm(np.arange(4), 5, extend='both', clip=True)
(4)                 with pytest.raises(ValueError, match="ncolors must equal or exceed"):
(8)                     mcolors.BoundaryNorm(np.arange(4), 2)
(4)                 with pytest.raises(ValueError, match="ncolors must equal or exceed"):
(8)                     mcolors.BoundaryNorm(np.arange(4), 3, extend='min')
(4)                 with pytest.raises(ValueError, match="ncolors must equal or exceed"):
(8)                     mcolors.BoundaryNorm(np.arange(4), 4, extend='both')
(4)                 bounds = [1, 2, 3]
(4)                 cmap = mpl.colormaps['viridis']
(4)                 mynorm = mcolors.BoundaryNorm(bounds, cmap.N, extend='both')
(4)                 refnorm = mcolors.BoundaryNorm([0] + bounds + [4], cmap.N)
(4)                 x = np.random.randn(100) * 10 + 2
(4)                 ref = refnorm(x)
(4)                 ref[ref == 0] = -1
(4)                 ref[ref == cmap.N - 1] = cmap.N
(4)                 assert_array_equal(mynorm(x), ref)
(4)                 cmref = mcolors.ListedColormap(['blue', 'red'])
(4)                 cmref.set_over('black')
(4)                 cmref.set_under('white')
(4)                 cmshould = mcolors.ListedColormap(['white', 'blue', 'red', 'black'])
(4)                 assert mcolors.same_color(cmref.get_over(), 'black')
(4)                 assert mcolors.same_color(cmref.get_under(), 'white')
(4)                 refnorm = mcolors.BoundaryNorm(bounds, cmref.N)
(4)                 mynorm = mcolors.BoundaryNorm(bounds, cmshould.N, extend='both')
(4)                 assert mynorm.vmin == refnorm.vmin
(4)                 assert mynorm.vmax == refnorm.vmax
(4)                 assert mynorm(bounds[0] - 0.1) == -1  # under
(4)                 assert mynorm(bounds[0] + 0.1) == 1   # first bin -> second color
(4)                 assert mynorm(bounds[-1] - 0.1) == cmshould.N - 2  # next-to-last color
(4)                 assert mynorm(bounds[-1] + 0.1) == cmshould.N  # over
(4)                 x = [-1, 1.2, 2.3, 9.6]
(4)                 assert_array_equal(cmshould(mynorm(x)), cmshould([0, 1, 2, 3]))
(4)                 x = np.random.randn(100) * 10 + 2
(4)                 assert_array_equal(cmshould(mynorm(x)), cmref(refnorm(x)))
(4)                 cmref = mcolors.ListedColormap(['blue', 'red'])
(4)                 cmref.set_under('white')
(4)                 cmshould = mcolors.ListedColormap(['white', 'blue', 'red'])
(4)                 assert mcolors.same_color(cmref.get_under(), 'white')
(4)                 assert cmref.N == 2
(4)                 assert cmshould.N == 3
(4)                 refnorm = mcolors.BoundaryNorm(bounds, cmref.N)
(4)                 mynorm = mcolors.BoundaryNorm(bounds, cmshould.N, extend='min')
(4)                 assert mynorm.vmin == refnorm.vmin
(4)                 assert mynorm.vmax == refnorm.vmax
(4)                 x = [-1, 1.2, 2.3]
(4)                 assert_array_equal(cmshould(mynorm(x)), cmshould([0, 1, 2]))
(4)                 x = np.random.randn(100) * 10 + 2
(4)                 assert_array_equal(cmshould(mynorm(x)), cmref(refnorm(x)))
(4)                 cmref = mcolors.ListedColormap(['blue', 'red'])
(4)                 cmref.set_over('black')
(4)                 cmshould = mcolors.ListedColormap(['blue', 'red', 'black'])
(4)                 assert mcolors.same_color(cmref.get_over(), 'black')
(4)                 assert cmref.N == 2
(4)                 assert cmshould.N == 3
(4)                 refnorm = mcolors.BoundaryNorm(bounds, cmref.N)
(4)                 mynorm = mcolors.BoundaryNorm(bounds, cmshould.N, extend='max')
(4)                 assert mynorm.vmin == refnorm.vmin
(4)                 assert mynorm.vmax == refnorm.vmax
(4)                 x = [1.2, 2.3, 4]
(4)                 assert_array_equal(cmshould(mynorm(x)), cmshould([0, 1, 2]))
(4)                 x = np.random.randn(100) * 10 + 2
(4)                 assert_array_equal(cmshould(mynorm(x)), cmref(refnorm(x)))
(0)             def test_CenteredNorm():
(4)                 np.random.seed(0)
(4)                 x = np.random.normal(size=100)
(4)                 x_maxabs = np.max(np.abs(x))
(4)                 norm_ref = mcolors.Normalize(vmin=-x_maxabs, vmax=x_maxabs)
```

```
290: (4)                norm = mcolors.CenteredNorm()
291: (4)                assert_array_almost_equal(norm_ref(x), norm(x))
292: (4)                vcenter = int(np.random.normal(scale=50))
293: (4)                norm = mcolors.CenteredNorm(vcenter=vcenter)
294: (4)                norm.autoscale_None([1, 2])
295: (4)                assert norm.vmax + norm.vmin == 2 * vcenter
296: (4)                norm = mcolors.CenteredNorm(halfrange=1.0)
297: (4)                norm.autoscale_None([1, 3000])
298: (4)                assert norm.halfrange == 1.0
299: (4)                x = np.random.normal(size=10)
300: (4)                norm = mcolors.CenteredNorm(vcenter=0.5, halfrange=0.5)
301: (4)                assert_array_almost_equal(x, norm(x))
302: (4)                norm = mcolors.CenteredNorm(vcenter=1, halfrange=1)
303: (4)                assert_array_almost_equal(x, 2 * norm(x))
304: (4)                norm = mcolors.CenteredNorm()
305: (4)                norm.vcenter = 2
306: (4)                norm.halfrange = 2
307: (4)                assert_array_almost_equal(x, 4 * norm(x))
308: (4)                norm = mcolors.CenteredNorm()
309: (4)                norm.halfrange = 2
310: (4)                norm.vcenter = 2
311: (4)                assert_array_almost_equal(x, 4 * norm(x))
312: (4)                norm = mcolors.CenteredNorm()
313: (4)                assert norm.vcenter == 0
314: (4)                norm(np.linspace(-1.0, 0.0, 10))
315: (4)                assert norm.vmax == 1.0
316: (4)                assert norm.halfrange == 1.0
317: (4)                norm.vcenter = 1
318: (4)                assert norm.vmin == 0
319: (4)                assert norm.vmax == 2
320: (4)                assert norm.halfrange == 1
321: (4)                norm.vmin = -1
322: (4)                assert norm.halfrange == 2
323: (4)                assert norm.vmax == 3
324: (4)                assert norm.vcenter == 1
325: (4)                norm.vmax = 2
326: (4)                assert norm.halfrange == 1
327: (4)                assert norm.vmin == 0
328: (4)                assert norm.vcenter == 1
329: (0)            @pytest.mark.parametrize("vmin,vmax", [[-1, 2], [3, 1]])
330: (0)            def test_lognorm_invalid(vmin, vmax):
331: (4)                norm = mcolors.LogNorm(vmin=vmin, vmax=vmax)
332: (4)                with pytest.raises(ValueError):
333: (8)                    norm(1)
334: (4)                with pytest.raises(ValueError):
335: (8)                    norm.inverse(1)
336: (0)            def test_LogNorm():
337: (4)                """
338: (4)                LogNorm ignored clip, now it has the same
339: (4)                behavior as Normalize, e.g., values > vmax are bigger than 1
340: (4)                without clip, with clip they are 1.
341: (4)                """
342: (4)                ln = mcolors.LogNorm(clip=True, vmax=5)
343: (4)                assert_array_equal(ln([1, 6]), [0, 1.0])
344: (0)            def test_LogNorm_inverse():
345: (4)                """
346: (4)                Test that lists work, and that the inverse works
347: (4)                """
348: (4)                norm = mcolors.LogNorm(vmin=0.1, vmax=10)
349: (4)                assert_array_almost_equal(norm([0.5, 0.4]), [0.349485, 0.30103])
350: (4)                assert_array_almost_equal([0.5, 0.4], norm.inverse([0.349485, 0.30103]))
351: (4)                assert_array_almost_equal(norm(0.4), [0.30103])
352: (4)                assert_array_almost_equal([0.4], norm.inverse([0.30103]))
353: (0)            def test_PowerNorm():
354: (4)                a = np.array([0, 0.5, 1, 1.5], dtype=float)
355: (4)                pnorm = mcolors.PowerNorm(1)
356: (4)                norm = mcolors.Normalize()
357: (4)                assert_array_almost_equal(norm(a), pnorm(a))
358: (4)                a = np.array([-0.5, 0, 2, 4, 8], dtype=float)
```

```
(4)                 expected = [-1/16, 0, 1/16, 1/4, 1]
(4)                 pnorm = mcolors.PowerNorm(2, vmin=0, vmax=8)
(4)                 assert_array_almost_equal(pnorm(a), expected)
(4)                 assert pnorm(a[0]) == expected[0]
(4)                 assert pnorm(a[2]) == expected[2]
(4)                 a_roundtrip = pnorm.inverse(pnorm(a))
(4)                 assert_array_almost_equal(a, a_roundtrip)
(4)                 assert_array_equal(a_roundtrip.mask, np.zeros(a.shape, dtype=bool))
(4)                 a = np.array([-0.5, 0, 1, 8, 16], dtype=float)
(4)                 expected = [0, 0, 0, 1, 1]
(4)                 pnorm = mcolors.PowerNorm(2, vmin=2, vmax=8, clip=True)
(4)                 assert_array_almost_equal(pnorm(a), expected)
(4)                 assert pnorm(a[0]) == expected[0]
(4)                 assert pnorm(a[-1]) == expected[-1]
(4)                 pnorm = mcolors.PowerNorm(2, vmin=2, vmax=8, clip=False)
(4)                 assert_array_almost_equal(pnorm(a, clip=True), expected)
(4)                 assert pnorm(a[0], clip=True) == expected[0]
(4)                 assert pnorm(a[-1], clip=True) == expected[-1]
(4)                 a = np.ma.array([5, 2], mask=[True, False])
(4)                 out = pnorm(a, clip=True)
(4)                 assert_array_equal(out.mask, [True, False])
(0)             def test_PowerNorm_translation_invariance():
(4)                 a = np.array([0, 1/2, 1], dtype=float)
(4)                 expected = [0, 1/8, 1]
(4)                 pnorm = mcolors.PowerNorm(vmin=0, vmax=1, gamma=3)
(4)                 assert_array_almost_equal(pnorm(a), expected)
(4)                 pnorm = mcolors.PowerNorm(vmin=-2, vmax=-1, gamma=3)
(4)                 assert_array_almost_equal(pnorm(a - 2), expected)
(0)             def test_powernorm_cbar_limits():
(4)                 fig, ax = plt.subplots()
(4)                 vmin, vmax = 300, 1000
(4)                 data = np.arange(10*10).reshape(10, 10) + vmin
(4)                 im = ax.imshow(data, norm=mcolors.PowerNorm(gamma=0.2, vmin=vmin,
vmax=vmax))
(4)                 cbar = fig.colorbar(im)
(4)                 assert cbar.ax.get_ylim() == (vmin, vmax)
(0)             def test_Normalize():
(4)                 norm = mcolors.Normalize()
(4)                 vals = np.arange(-10, 10, 1, dtype=float)
(4)                 _inverse_tester(norm, vals)
(4)                 _scalar_tester(norm, vals)
(4)                 _mask_tester(norm, vals)
(4)                 vals = np.array([-128, 127], dtype=np.int8)
(4)                 norm = mcolors.Normalize(vals.min(), vals.max())
(4)                 assert_array_equal(norm(vals), [0, 1])
(4)                 vals = np.array([1.2345678901, 9.8765432109], dtype=np.longdouble)
(4)                 norm = mcolors.Normalize(vals[0], vals[1])
(4)                 assert norm(vals).dtype == np.longdouble
(4)                 assert_array_equal(norm(vals), [0, 1])
(4)                 eps = np.finfo(np.longdouble).resolution
(4)                 norm = plt.Normalize(1, 1 + 100 * eps)
(4)                 assert_array_almost_equal(norm(1 + 50 * eps), 0.5, decimal=3)
(0)             def test_FuncNorm():
(4)                 def forward(x):
(8)                     return (x**2)
(4)                 def inverse(x):
(8)                     return np.sqrt(x)
(4)                 norm = mcolors.FuncNorm((forward, inverse), vmin=0, vmax=10)
(4)                 expected = np.array([0, 0.25, 1])
(4)                 input = np.array([0, 5, 10])
(4)                 assert_array_almost_equal(norm(input), expected)
(4)                 assert_array_almost_equal(norm.inverse(expected), input)
(4)                 def forward(x):
(8)                     return np.log10(x)
(4)                 def inverse(x):
(8)                     return 10**x
(4)                 norm = mcolors.FuncNorm((forward, inverse), vmin=0.1, vmax=10)
(4)                 lognorm = mcolors.LogNorm(vmin=0.1, vmax=10)
(4)                 assert_array_almost_equal(norm([0.2, 5, 10]), lognorm([0.2, 5, 10]))
```

```
(4)                 assert_array_almost_equal(norm.inverse([0.2, 5, 10]),
(30)                                          lognorm.inverse([0.2, 5, 10]))
(0)             def test_TwoSlopeNorm_autoscale():
(4)                 norm = mcolors.TwoSlopeNorm(vcenter=20)
(4)                 norm.autoscale([10, 20, 30, 40])
(4)                 assert norm.vmin == 10.
(4)                 assert norm.vmax == 40.
(0)             def test_TwoSlopeNorm_autoscale_None_vmin():
(4)                 norm = mcolors.TwoSlopeNorm(2, vmin=0, vmax=None)
(4)                 norm.autoscale_None([1, 2, 3, 4, 5])
(4)                 assert norm(5) == 1
(4)                 assert norm.vmax == 5
(0)             def test_TwoSlopeNorm_autoscale_None_vmax():
(4)                 norm = mcolors.TwoSlopeNorm(2, vmin=None, vmax=10)
(4)                 norm.autoscale_None([1, 2, 3, 4, 5])
(4)                 assert norm(1) == 0
(4)                 assert norm.vmin == 1
(0)             def test_TwoSlopeNorm_scale():
(4)                 norm = mcolors.TwoSlopeNorm(2)
(4)                 assert norm.scaled() is False
(4)                 norm([1, 2, 3, 4])
(4)                 assert norm.scaled() is True
(0)             def test_TwoSlopeNorm_scaleout_center():
(4)                 norm = mcolors.TwoSlopeNorm(vcenter=0)
(4)                 norm([0, 1, 2, 3, 5])
(4)                 assert norm.vmin == -5
(4)                 assert norm.vmax == 5
(0)             def test_TwoSlopeNorm_scaleout_center_max():
(4)                 norm = mcolors.TwoSlopeNorm(vcenter=0)
(4)                 norm([0, -1, -2, -3, -5])
(4)                 assert norm.vmax == 5
(4)                 assert norm.vmin == -5
(0)             def test_TwoSlopeNorm_Even():
(4)                 norm = mcolors.TwoSlopeNorm(vmin=-1, vcenter=0, vmax=4)
(4)                 vals = np.array([-1.0, -0.5, 0.0, 1.0, 2.0, 3.0, 4.0])
(4)                 expected = np.array([0.0, 0.25, 0.5, 0.625, 0.75, 0.875, 1.0])
(4)                 assert_array_equal(norm(vals), expected)
(0)             def test_TwoSlopeNorm_Odd():
(4)                 norm = mcolors.TwoSlopeNorm(vmin=-2, vcenter=0, vmax=5)
(4)                 vals = np.array([-2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
(4)                 expected = np.array([0.0, 0.25, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
(4)                 assert_array_equal(norm(vals), expected)
(0)             def test_TwoSlopeNorm_VminEqualsVcenter():
(4)                 with pytest.raises(ValueError):
(8)                     mcolors.TwoSlopeNorm(vmin=-2, vcenter=-2, vmax=2)
(0)             def test_TwoSlopeNorm_VmaxEqualsVcenter():
(4)                 with pytest.raises(ValueError):
(8)                     mcolors.TwoSlopeNorm(vmin=-2, vcenter=2, vmax=2)
(0)             def test_TwoSlopeNorm_VminGTVcenter():
(4)                 with pytest.raises(ValueError):
(8)                     mcolors.TwoSlopeNorm(vmin=10, vcenter=0, vmax=20)
(0)             def test_TwoSlopeNorm_TwoSlopeNorm_VminGTVmax():
(4)                 with pytest.raises(ValueError):
(8)                     mcolors.TwoSlopeNorm(vmin=10, vcenter=0, vmax=5)
(0)             def test_TwoSlopeNorm_VcenterGTVmax():
(4)                 with pytest.raises(ValueError):
(8)                     mcolors.TwoSlopeNorm(vmin=10, vcenter=25, vmax=20)
(0)             def test_TwoSlopeNorm_premature_scaling():
(4)                 norm = mcolors.TwoSlopeNorm(vcenter=2)
(4)                 with pytest.raises(ValueError):
(8)                     norm.inverse(np.array([0.1, 0.5, 0.9]))
(0)             def test_SymLogNorm():
(4)                 """
(4)                 Test SymLogNorm behavior
(4)                 """
(4)                 norm = mcolors.SymLogNorm(3, vmax=5, linscale=1.2, base=np.e)
(4)                 vals = np.array([-30, -1, 2, 6], dtype=float)
(4)                 normed_vals = norm(vals)
(4)                 expected = [0., 0.53980074, 0.826991, 1.02758204]
```

```
(4)                 assert_array_almost_equal(normed_vals, expected)
(4)                 _inverse_tester(norm, vals)
(4)                 _scalar_tester(norm, vals)
(4)                 _mask_tester(norm, vals)
(4)                 norm = mcolors.SymLogNorm(3, vmin=-30, vmax=5, linscale=1.2, base=np.e)
(4)                 normed_vals = norm(vals)
(4)                 assert_array_almost_equal(normed_vals, expected)
(4)                 norm = mcolors.SymLogNorm(1, vmin=-np.e**3, vmax=np.e**3, base=np.e)
(4)                 nn = norm([-np.e**3, -np.e**2, -np.e**1, -1,
(14)                          0, 1, np.e**1, np.e**2, np.e**3])
(4)                 xx = np.array([0., 0.109123, 0.218246, 0.32737, 0.5, 0.67263,
(19)                               0.781754, 0.890877, 1.])
(4)                 assert_array_almost_equal(nn, xx)
(4)                 norm = mcolors.SymLogNorm(1, vmin=-10**3, vmax=10**3, base=10)
(4)                 nn = norm([-10**3, -10**2, -10**1, -1,
(14)                          0, 1, 10**1, 10**2, 10**3])
(4)                 xx = np.array([0., 0.121622, 0.243243, 0.364865, 0.5, 0.635135,
(19)                               0.756757, 0.878378, 1.])
(4)                 assert_array_almost_equal(nn, xx)
(0)             def test_SymLogNorm_colorbar():
(4)                 """
(4)                 Test un-called SymLogNorm in a colorbar.
(4)                 """
(4)                 norm = mcolors.SymLogNorm(0.1, vmin=-1, vmax=1, linscale=1, base=np.e)
(4)                 fig = plt.figure()
(4)                 mcolorbar.ColorbarBase(fig.add_subplot(), norm=norm)
(4)                 plt.close(fig)
(0)             def test_SymLogNorm_single_zero():
(4)                 """
(4)                 Test SymLogNorm to ensure it is not adding sub-ticks to zero label
(4)                 """
(4)                 fig = plt.figure()
(4)                 norm = mcolors.SymLogNorm(1e-5, vmin=-1, vmax=1, base=np.e)
(4)                 cbar = mcolorbar.ColorbarBase(fig.add_subplot(), norm=norm)
(4)                 ticks = cbar.get_ticks()
(4)                 assert np.count_nonzero(ticks == 0) <= 1
(4)                 plt.close(fig)
(0)             class TestAsinhNorm:
(4)                 """
(4)                 Tests for `~.colors.AsinhNorm`
(4)                 """
(4)                 def test_init(self):
(8)                     norm0 = mcolors.AsinhNorm()
(8)                     assert norm0.linear_width == 1
(8)                     norm5 = mcolors.AsinhNorm(linear_width=5)
(8)                     assert norm5.linear_width == 5
(4)                 def test_norm(self):
(8)                     norm = mcolors.AsinhNorm(2, vmin=-4, vmax=4)
(8)                     vals = np.arange(-3.5, 3.5, 10)
(8)                     normed_vals = norm(vals)
(8)                     asinh2 = np.arcsinh(2)
(8)                     expected = (2 * np.arcsinh(vals / 2) + 2 * asinh2) / (4 * asinh2)
(8)                     assert_array_almost_equal(normed_vals, expected)
(0)             def _inverse_tester(norm_instance, vals):
(4)                 """
(4)                 Checks if the inverse of the given normalization is working.
(4)                 """
(4)                 assert_array_almost_equal(norm_instance.inverse(norm_instance(vals)),
vals)
(0)             def _scalar_tester(norm_instance, vals):
(4)                 """
(4)                 Checks if scalars and arrays are handled the same way.
(4)                 Tests only for float.
(4)                 """
(4)                 scalar_result = [norm_instance(float(v)) for v in vals]
(4)                 assert_array_almost_equal(scalar_result, norm_instance(vals))
(0)             def _mask_tester(norm_instance, vals):
(4)                 """
(4)                 Checks mask handling
```

```
564: (4)                """
565: (4)                masked_array = np.ma.array(vals)
566: (4)                masked_array[0] = np.ma.masked
567: (4)                assert_array_equal(masked_array.mask, norm_instance(masked_array).mask)
568: (0)            @image_comparison(['levels_and_colors.png'])
569: (0)            def test_cmap_and_norm_from_levels_and_colors():
570: (4)                plt.rcParams['pcolormesh.snap'] = False
571: (4)                data = np.linspace(-2, 4, 49).reshape(7, 7)
572: (4)                levels = [-1, 2, 2.5, 3]
573: (4)                colors = ['red', 'green', 'blue', 'yellow', 'black']
574: (4)                extend = 'both'
575: (4)                cmap, norm = mcolors.from_levels_and_colors(levels, colors, extend=extend)
576: (4)                ax = plt.axes()
577: (4)                m = plt.pcolormesh(data, cmap=cmap, norm=norm)
578: (4)                plt.colorbar(m)
579: (4)                ax.tick_params(labelleft=False, labelbottom=False)
580: (0)            @image_comparison(baseline_images=['boundarynorm_and_colorbar'],
581: (18)                             extensions=['png'], tol=1.0)
582: (0)            def test_boundarynorm_and_colorbarbase():
583: (4)                plt.rcParams['pcolormesh.snap'] = False
584: (4)                fig = plt.figure()
585: (4)                ax1 = fig.add_axes([0.05, 0.80, 0.9, 0.15])
586: (4)                ax2 = fig.add_axes([0.05, 0.475, 0.9, 0.15])
587: (4)                ax3 = fig.add_axes([0.05, 0.15, 0.9, 0.15])
588: (4)                bounds = [-1, 2, 5, 7, 12, 15]
589: (4)                cmap = mpl.colormaps['viridis']
590: (4)                norm = mcolors.BoundaryNorm(bounds, cmap.N)
591: (4)                cb1 = mcolorbar.ColorbarBase(ax1, cmap=cmap, norm=norm, extend='both',
592: (33)                                            orientation='horizontal', spacing='uniform')
593: (4)                norm = mcolors.BoundaryNorm(bounds, cmap.N, extend='both')
594: (4)                cb2 = mcolorbar.ColorbarBase(ax2, cmap=cmap, norm=norm,
595: (33)                                            orientation='horizontal')
596: (4)                norm = mcolors.BoundaryNorm(bounds, cmap.N, extend='both')
597: (4)                cb3 = mcolorbar.ColorbarBase(ax3, cmap=cmap, norm=norm,
598: (33)                                            extend='neither', orientation='horizontal')
599: (0)            def test_cmap_and_norm_from_levels_and_colors2():
600: (4)                levels = [-1, 2, 2.5, 3]
601: (4)                colors = ['red', (0, 1, 0), 'blue', (0.5, 0.5, 0.5), (0.0, 0.0, 0.0, 1.0)]
602: (4)                clr = mcolors.to_rgba_array(colors)
603: (4)                bad = (0.1, 0.1, 0.1, 0.1)
604: (4)                no_color = (0.0, 0.0, 0.0, 0.0)
605: (4)                masked_value = 'masked_value'
606: (4)                tests = [('both', None, {-2: clr[0],
607: (29)                                        -1: clr[1],
608: (29)                                        2: clr[2],
609: (29)                                        2.25: clr[2],
610: (29)                                        3: clr[4],
611: (29)                                        3.5: clr[4],
612: (29)                                        masked_value: bad}),
613: (13)                         ('min', -1, {-2: clr[0],
614: (26)                                      -1: clr[1],
615: (26)                                      2: clr[2],
616: (26)                                      2.25: clr[2],
617: (26)                                      3: no_color,
618: (26)                                      3.5: no_color,
619: (26)                                      masked_value: bad}),
620: (13)                         ('max', -1, {-2: no_color,
621: (26)                                      -1: clr[0],
622: (26)                                      2: clr[1],
623: (26)                                      2.25: clr[1],
624: (26)                                      3: clr[3],
625: (26)                                      3.5: clr[3],
626: (26)                                      masked_value: bad}),
627: (13)                         ('neither', -2, {-2: no_color,
628: (30)                                          -1: clr[0],
629: (30)                                          2: clr[1],
630: (30)                                          2.25: clr[1],
631: (30)                                          3: no_color,
632: (30)                                          3.5: no_color,
```

```
(30)                                    masked_value: bad}),
(13)                        ]
(4)                 for extend, i1, cases in tests:
(8)                     cmap, norm = mcolors.from_levels_and_colors(levels, colors[0:i1],
(52)                                                            extend=extend)
(8)                     cmap.set_bad(bad)
(8)                     for d_val, expected_color in cases.items():
(12)                        if d_val == masked_value:
(16)                            d_val = np.ma.array([1], mask=True)
(12)                        else:
(16)                            d_val = [d_val]
(12)                        assert_array_equal(expected_color, cmap(norm(d_val))[0],
(31)                                           f'With extend={extend!r} and data '
(31)                                           f'value={d_val!r}')
(4)                 with pytest.raises(ValueError):
(8)                     mcolors.from_levels_and_colors(levels, colors)
(0)             def test_rgb_hsv_round_trip():
(4)                 for a_shape in [(500, 500, 3), (500, 3), (1, 3), (3,)]:
(8)                     np.random.seed(0)
(8)                     tt = np.random.random(a_shape)
(8)                     assert_array_almost_equal(
(12)                        tt, mcolors.hsv_to_rgb(mcolors.rgb_to_hsv(tt)))
(8)                     assert_array_almost_equal(
(12)                        tt, mcolors.rgb_to_hsv(mcolors.hsv_to_rgb(tt)))
(0)             def test_autoscale_masked():
(4)                 data = np.ma.masked_all((12, 20))
(4)                 plt.pcolor(data)
(4)                 plt.draw()
(0)             @image_comparison(['light_source_shading_topo.png'])
(0)             def test_light_source_topo_surface():
(4)                 """Shades a DEM using different v.e.'s and blend modes."""
(4)                 dem = cbook.get_sample_data('jacksboro_fault_dem.npz')
(4)                 elev = dem['elevation']
(4)                 dx, dy = dem['dx'], dem['dy']
(4)                 dx = 111320.0 * dx * np.cos(dem['ymin'])
(4)                 dy = 111320.0 * dy
(4)                 ls = mcolors.LightSource(315, 45)
(4)                 cmap = cm.gist_earth
(4)                 fig, axs = plt.subplots(nrows=3, ncols=3)
(4)                 for row, mode in zip(axs, ['hsv', 'overlay', 'soft']):
(8)                     for ax, ve in zip(row, [0.1, 1, 10]):
(12)                        rgb = ls.shade(elev, cmap, vert_exag=ve, dx=dx, dy=dy,
(27)                                       blend_mode=mode)
(12)                        ax.imshow(rgb)
(12)                        ax.set(xticks=[], yticks=[])
(0)             def test_light_source_shading_default():
(4)                 """
(4)                 Array comparison test for the default "hsv" blend mode. Ensure the
(4)                 default result doesn't change without warning.
(4)                 """
(4)                 y, x = np.mgrid[-1.2:1.2:8j, -1.2:1.2:8j]
(4)                 z = 10 * np.cos(x**2 + y**2)
(4)                 cmap = plt.cm.copper
(4)                 ls = mcolors.LightSource(315, 45)
(4)                 rgb = ls.shade(z, cmap)
(4)                 expect = np.array(
(8)                     [[[0.00, 0.45, 0.90, 0.90, 0.82, 0.62, 0.28, 0.00],
(10)                      [0.45, 0.94, 0.99, 1.00, 1.00, 0.96, 0.65, 0.17],
(10)                      [0.90, 0.99, 1.00, 1.00, 1.00, 1.00, 0.94, 0.35],
(10)                      [0.90, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 0.49],
(10)                      [0.82, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 0.41],
(10)                      [0.62, 0.96, 1.00, 1.00, 1.00, 1.00, 0.90, 0.07],
(10)                      [0.28, 0.65, 0.94, 1.00, 1.00, 0.90, 0.35, 0.01],
(10)                      [0.00, 0.17, 0.35, 0.49, 0.41, 0.07, 0.01, 0.00]],
(9)                      [[0.00, 0.28, 0.59, 0.72, 0.62, 0.40, 0.18, 0.00],
(10)                      [0.28, 0.78, 0.93, 0.92, 0.83, 0.66, 0.39, 0.11],
(10)                      [0.59, 0.93, 0.99, 1.00, 0.92, 0.75, 0.50, 0.21],
(10)                      [0.72, 0.92, 1.00, 0.99, 0.93, 0.76, 0.51, 0.18],
(10)                      [0.62, 0.83, 0.92, 0.93, 0.87, 0.68, 0.42, 0.08],
```

```
(10)                     [0.40, 0.66, 0.75, 0.76, 0.68, 0.52, 0.23, 0.02],
(10)                     [0.18, 0.39, 0.50, 0.51, 0.42, 0.23, 0.00, 0.00],
(10)                     [0.00, 0.11, 0.21, 0.18, 0.08, 0.02, 0.00, 0.00]],
(9)                    [[0.00, 0.18, 0.38, 0.46, 0.39, 0.26, 0.11, 0.00],
(10)                     [0.18, 0.50, 0.70, 0.75, 0.64, 0.44, 0.25, 0.07],
(10)                     [0.38, 0.70, 0.91, 0.98, 0.81, 0.51, 0.29, 0.13],
(10)                     [0.46, 0.75, 0.98, 0.96, 0.84, 0.48, 0.22, 0.12],
(10)                     [0.39, 0.64, 0.81, 0.84, 0.71, 0.31, 0.11, 0.05],
(10)                     [0.26, 0.44, 0.51, 0.48, 0.31, 0.10, 0.03, 0.01],
(10)                     [0.11, 0.25, 0.29, 0.22, 0.11, 0.03, 0.00, 0.00],
(10)                     [0.00, 0.07, 0.13, 0.12, 0.05, 0.01, 0.00, 0.00]],
(9)                    [[1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00],
(10)                     [1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00],
(10)                     [1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00],
(10)                     [1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00],
(10)                     [1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00],
(10)                     [1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00],
(10)                     [1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00],
(10)                     [1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00]]
(9)                    ]).T
(4)                assert_array_almost_equal(rgb, expect, decimal=2)
(0)            def test_light_source_shading_empty_mask():
(4)                y, x = np.mgrid[-1.2:1.2:8j, -1.2:1.2:8j]
(4)                z0 = 10 * np.cos(x**2 + y**2)
(4)                z1 = np.ma.array(z0)
(4)                cmap = plt.cm.copper
(4)                ls = mcolors.LightSource(315, 45)
(4)                rgb0 = ls.shade(z0, cmap)
(4)                rgb1 = ls.shade(z1, cmap)
(4)                assert_array_almost_equal(rgb0, rgb1)
(0)            def test_light_source_masked_shading():
(4)                """
(4)                Array comparison test for a surface with a masked portion. Ensures that
(4)                we don't wind up with "fringes" of odd colors around masked regions.
(4)                """
(4)                y, x = np.mgrid[-1.2:1.2:8j, -1.2:1.2:8j]
(4)                z = 10 * np.cos(x**2 + y**2)
(4)                z = np.ma.masked_greater(z, 9.9)
(4)                cmap = plt.cm.copper
(4)                ls = mcolors.LightSource(315, 45)
(4)                rgb = ls.shade(z, cmap)
(4)                expect = np.array(
(8)                    [[[0.00, 0.46, 0.91, 0.91, 0.84, 0.64, 0.29, 0.00],
(10)                     [0.46, 0.96, 1.00, 1.00, 1.00, 0.97, 0.67, 0.18],
(10)                     [0.91, 1.00, 1.00, 1.00, 1.00, 1.00, 0.96, 0.36],
(10)                     [0.91, 1.00, 1.00, 0.00, 0.00, 1.00, 1.00, 0.51],
(10)                     [0.84, 1.00, 1.00, 0.00, 0.00, 1.00, 1.00, 0.44],
(10)                     [0.64, 0.97, 1.00, 1.00, 1.00, 1.00, 0.94, 0.09],
(10)                     [0.29, 0.67, 0.96, 1.00, 1.00, 0.94, 0.38, 0.01],
(10)                     [0.00, 0.18, 0.36, 0.51, 0.44, 0.09, 0.01, 0.00]],
(9)                    [[0.00, 0.29, 0.61, 0.75, 0.64, 0.41, 0.18, 0.00],
(10)                     [0.29, 0.81, 0.95, 0.93, 0.85, 0.68, 0.40, 0.11],
(10)                     [0.61, 0.95, 1.00, 0.78, 0.78, 0.77, 0.52, 0.22],
(10)                     [0.75, 0.93, 0.78, 0.00, 0.00, 0.78, 0.54, 0.19],
(10)                     [0.64, 0.85, 0.78, 0.00, 0.00, 0.78, 0.45, 0.08],
(10)                     [0.41, 0.68, 0.77, 0.78, 0.78, 0.55, 0.25, 0.02],
(10)                     [0.18, 0.40, 0.52, 0.54, 0.45, 0.25, 0.00, 0.00],
(10)                     [0.00, 0.11, 0.22, 0.19, 0.08, 0.02, 0.00, 0.00]],
(9)                    [[0.00, 0.19, 0.39, 0.48, 0.41, 0.26, 0.12, 0.00],
(10)                     [0.19, 0.52, 0.73, 0.78, 0.66, 0.46, 0.26, 0.07],
(10)                     [0.39, 0.73, 0.95, 0.50, 0.50, 0.53, 0.30, 0.14],
(10)                     [0.48, 0.78, 0.50, 0.00, 0.00, 0.50, 0.23, 0.12],
(10)                     [0.41, 0.66, 0.50, 0.00, 0.00, 0.50, 0.11, 0.05],
(10)                     [0.26, 0.46, 0.53, 0.50, 0.50, 0.11, 0.03, 0.01],
(10)                     [0.12, 0.26, 0.30, 0.23, 0.11, 0.03, 0.00, 0.00],
(10)                     [0.00, 0.07, 0.14, 0.12, 0.05, 0.01, 0.00, 0.00]],
(9)                    [[1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00],
(10)                     [1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00],
(10)                     [1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00],
```

```
(10)                    [1.00, 1.00, 1.00, 0.00, 0.00, 1.00, 1.00, 1.00],
(10)                    [1.00, 1.00, 1.00, 0.00, 0.00, 1.00, 1.00, 1.00],
(10)                    [1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00],
(10)                    [1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00],
(10)                    [1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00]],
(9)                   ]).T
(4)                assert_array_almost_equal(rgb, expect, decimal=2)
(0)            def test_light_source_hillshading():
(4)                """
(4)                Compare the current hillshading method against one that should be
(4)                mathematically equivalent. Illuminates a cone from a range of angles.
(4)                """
(4)                def alternative_hillshade(azimuth, elev, z):
(8)                    illum = _sph2cart(*_azimuth2math(azimuth, elev))
(8)                    illum = np.array(illum)
(8)                    dy, dx = np.gradient(-z)
(8)                    dy = -dy
(8)                    dz = np.ones_like(dy)
(8)                    normals = np.dstack([dx, dy, dz])
(8)                    normals /= np.linalg.norm(normals, axis=2)[..., None]
(8)                    intensity = np.tensordot(normals, illum, axes=(2, 0))
(8)                    intensity -= intensity.min()
(8)                    intensity /= np.ptp(intensity)
(8)                    return intensity
(4)                y, x = np.mgrid[5:0:-1, :5]
(4)                z = -np.hypot(x - x.mean(), y - y.mean())
(4)                for az, elev in itertools.product(range(0, 390, 30), range(0, 105, 15)):
(8)                    ls = mcolors.LightSource(az, elev)
(8)                    h1 = ls.hillshade(z)
(8)                    h2 = alternative_hillshade(az, elev, z)
(8)                    assert_array_almost_equal(h1, h2)
(0)            def test_light_source_planar_hillshading():
(4)                """
(4)                Ensure that the illumination intensity is correct for planar surfaces.
(4)                """
(4)                def plane(azimuth, elevation, x, y):
(8)                    """
(8)                    Create a plane whose normal vector is at the given azimuth and
(8)                    elevation.
(8)                    """
(8)                    theta, phi = _azimuth2math(azimuth, elevation)
(8)                    a, b, c = _sph2cart(theta, phi)
(8)                    z = -(a*x + b*y) / c
(8)                    return z
(4)                def angled_plane(azimuth, elevation, angle, x, y):
(8)                    """
(8)                    Create a plane whose normal vector is at an angle from the given
(8)                    azimuth and elevation.
(8)                    """
(8)                    elevation = elevation + angle
(8)                    if elevation > 90:
(12)                       azimuth = (azimuth + 180) % 360
(12)                       elevation = (90 - elevation) % 90
(8)                    return plane(azimuth, elevation, x, y)
(4)                y, x = np.mgrid[5:0:-1, :5]
(4)                for az, elev in itertools.product(range(0, 390, 30), range(0, 105, 15)):
(8)                    ls = mcolors.LightSource(az, elev)
(8)                    for angle in range(0, 105, 15):
(12)                       z = angled_plane(az, elev, angle, x, y)
(12)                       h = ls.hillshade(z)
(12)                       assert_array_almost_equal(h, np.cos(np.radians(angle)))
(0)            def test_color_names():
(4)                assert mcolors.to_hex("blue") == "#0000ff"
(4)                assert mcolors.to_hex("xkcd:blue") == "#0343df"
(4)                assert mcolors.to_hex("tab:blue") == "#1f77b4"
(0)            def _sph2cart(theta, phi):
(4)                x = np.cos(theta) * np.sin(phi)
(4)                y = np.sin(theta) * np.sin(phi)
(4)                z = np.cos(phi)
```

```
(4)                 return x, y, z
(0)             def _azimuth2math(azimuth, elevation):
(4)                 """
(4)                 Convert from clockwise-from-north and up-from-horizontal to mathematical
(4)                 conventions.
(4)                 """
(4)                 theta = np.radians((90 - azimuth) % 360)
(4)                 phi = np.radians(90 - elevation)
(4)                 return theta, phi
(0)             def test_pandas_iterable(pd):
(4)                 lst = ['red', 'blue', 'green']
(4)                 s = pd.Series(lst)
(4)                 cm1 = mcolors.ListedColormap(lst, N=5)
(4)                 cm2 = mcolors.ListedColormap(s, N=5)
(4)                 assert_array_equal(cm1.colors, cm2.colors)
(0)             @pytest.mark.parametrize('name', sorted(mpl.colormaps()))
(0)             def test_colormap_reversing(name):
(4)                 """
(4)                 Check the generated _lut data of a colormap and corresponding reversed
(4)                 colormap if they are almost the same.
(4)                 """
(4)                 cmap = mpl.colormaps[name]
(4)                 cmap_r = cmap.reversed()
(4)                 if not cmap_r._isinit:
(8)                     cmap._init()
(8)                     cmap_r._init()
(4)                 assert_array_almost_equal(cmap._lut[:-3], cmap_r._lut[-4::-1])
(4)                 assert_array_almost_equal(cmap(-np.inf), cmap_r(np.inf))
(4)                 assert_array_almost_equal(cmap(np.inf), cmap_r(-np.inf))
(4)                 assert_array_almost_equal(cmap(np.nan), cmap_r(np.nan))
(0)             def test_has_alpha_channel():
(4)                 assert mcolors._has_alpha_channel((0, 0, 0, 0))
(4)                 assert mcolors._has_alpha_channel([1, 1, 1, 1])
(4)                 assert not mcolors._has_alpha_channel('blue')  # 4-char string!
(4)                 assert not mcolors._has_alpha_channel('0.25')
(4)                 assert not mcolors._has_alpha_channel('r')
(4)                 assert not mcolors._has_alpha_channel((1, 0, 0))
(0)             def test_cn():
(4)                 matplotlib.rcParams['axes.prop_cycle'] = cycler('color',
(52)                                                            ['blue', 'r'])
(4)                 assert mcolors.to_hex("C0") == '#0000ff'
(4)                 assert mcolors.to_hex("C1") == '#ff0000'
(4)                 matplotlib.rcParams['axes.prop_cycle'] = cycler('color',
(52)                                                            ['xkcd:blue', 'r'])
(4)                 assert mcolors.to_hex("C0") == '#0343df'
(4)                 assert mcolors.to_hex("C1") == '#ff0000'
(4)                 assert mcolors.to_hex("C10") == '#0343df'
(4)                 assert mcolors.to_hex("C11") == '#ff0000'
(4)                 matplotlib.rcParams['axes.prop_cycle'] = cycler('color', ['8e4585', 'r'])
(4)                 assert mcolors.to_hex("C0") == '#8e4585'
(4)                 assert mcolors.to_rgb("C0")[0] != np.inf
(0)             def test_conversions():
(4)                 assert_array_equal(mcolors.to_rgba_array("none"), np.zeros((0, 4)))
(4)                 assert_array_equal(mcolors.to_rgba_array([]), np.zeros((0, 4)))
(4)                 assert_array_equal(
(8)                     mcolors.to_rgba_array([".2", ".5", ".8"]),
(8)                     np.vstack([mcolors.to_rgba(c) for c in [".2", ".5", ".8"]]))
(4)                 assert mcolors.to_rgba((1, 1, 1), .5) == (1, 1, 1, .5)
(4)                 assert mcolors.to_rgba(".1", .5) == (.1, .1, .1, .5)
(4)                 assert mcolors.to_hex((.7, .7, .7)) == "#b2b2b2"
(4)                 hex_color = "#1234abcd"
(4)                 assert mcolors.to_hex(mcolors.to_rgba(hex_color), keep_alpha=True) == \
(8)                     hex_color
(0)             def test_conversions_masked():
(4)                 x1 = np.ma.array(['k', 'b'], mask=[True, False])
(4)                 x2 = np.ma.array([[0, 0, 0, 1], [0, 0, 1, 1]])
(4)                 x2[0] = np.ma.masked
(4)                 assert mcolors.to_rgba(x1[0]) == (0, 0, 0, 0)
(4)                 assert_array_equal(mcolors.to_rgba_array(x1),
```

```
 909: (23)                                [[0, 0, 0, 0], [0, 0, 1, 1]])
 910: (4)                assert_array_equal(mcolors.to_rgba_array(x2), mcolors.to_rgba_array(x1))
 911: (0)            def test_to_rgba_array_single_str():
 912: (4)                assert_array_equal(mcolors.to_rgba_array("red"), [(1, 0, 0, 1)])
 913: (4)                with pytest.raises(ValueError,
 914: (23)                                  match="'rgb' is not a valid color value."):
 915: (8)                    array = mcolors.to_rgba_array("rgb")
 916: (0)            def test_to_rgba_array_2tuple_str():
 917: (4)                expected = np.array([[0, 0, 0, 1], [1, 1, 1, 1]])
 918: (4)                assert_array_equal(mcolors.to_rgba_array(("k", "w")), expected)
 919: (0)            def test_to_rgba_array_alpha_array():
 920: (4)                with pytest.raises(ValueError, match="The number of colors must match"):
 921: (8)                    mcolors.to_rgba_array(np.ones((5, 3), float), alpha=np.ones((2,)))
 922: (4)                alpha = [0.5, 0.6]
 923: (4)                c = mcolors.to_rgba_array(np.ones((2, 3), float), alpha=alpha)
 924: (4)                assert_array_equal(c[:, 3], alpha)
 925: (4)                c = mcolors.to_rgba_array(['r', 'g'], alpha=alpha)
 926: (4)                assert_array_equal(c[:, 3], alpha)
 927: (0)            def test_to_rgba_array_accepts_color_alpha_tuple():
 928: (4)                assert_array_equal(
 929: (8)                    mcolors.to_rgba_array(('black', 0.9)),
 930: (8)                    [[0, 0, 0, 0.9]])
 931: (0)            def test_to_rgba_array_explicit_alpha_overrides_tuple_alpha():
 932: (4)                assert_array_equal(
 933: (8)                    mcolors.to_rgba_array(('black', 0.9), alpha=0.5),
 934: (8)                    [[0, 0, 0, 0.5]])
 935: (0)            def test_to_rgba_array_accepts_color_alpha_tuple_with_multiple_colors():
 936: (4)                color_array = np.array([[1., 1., 1., 1.], [0., 0., 1., 0.]])
 937: (4)                assert_array_equal(
 938: (8)                    mcolors.to_rgba_array((color_array, 0.2)),
 939: (8)                    [[1., 1., 1., 0.2], [0., 0., 1., 0.2]])
 940: (4)                color_sequence = [[1., 1., 1., 1.], [0., 0., 1., 0.]]
 941: (4)                assert_array_equal(
 942: (8)                    mcolors.to_rgba_array((color_sequence, 0.4)),
 943: (8)                    [[1., 1., 1., 0.4], [0., 0., 1., 0.4]])
 944: (0)            def test_to_rgba_array_error_with_color_invalid_alpha_tuple():
 945: (4)                with pytest.raises(ValueError, match="'alpha' must be between 0 and 1,"):
 946: (8)                    mcolors.to_rgba_array(('black', 2.0))
 947: (0)            @pytest.mark.parametrize('rgba_alpha',
 948: (25)                                   [('white', 0.5), ('#ffffff', 0.5), ('#ffffff00',
 0.5),
 949: (26)                                    ((1.0, 1.0, 1.0, 1.0), 0.5)])
 950: (0)            def test_to_rgba_accepts_color_alpha_tuple(rgba_alpha):
 951: (4)                assert mcolors.to_rgba(rgba_alpha) == (1, 1, 1, 0.5)
 952: (0)            def test_to_rgba_explicit_alpha_overrides_tuple_alpha():
 953: (4)                assert mcolors.to_rgba(('red', 0.1), alpha=0.9) == (1, 0, 0, 0.9)
 954: (0)            def test_to_rgba_error_with_color_invalid_alpha_tuple():
 955: (4)                with pytest.raises(ValueError, match="'alpha' must be between 0 and 1"):
 956: (8)                    mcolors.to_rgba(('blue', 2.0))
 957: (0)            @pytest.mark.parametrize("bytes", (True, False))
 958: (0)            def test_scalarmappable_to_rgba(bytes):
 959: (4)                sm = cm.ScalarMappable()
 960: (4)                alpha_1 = 255 if bytes else 1
 961: (4)                x = np.ones((2, 3, 4), dtype=np.uint8)
 962: (4)                expected = x.copy() if bytes else x.astype(np.float32)/255
 963: (4)                np.testing.assert_almost_equal(sm.to_rgba(x, bytes=bytes), expected)
 964: (4)                expected[..., 3] = alpha_1
 965: (4)                np.testing.assert_almost_equal(sm.to_rgba(x[..., :3], bytes=bytes),
 expected)
 966: (4)                xm = np.ma.masked_array(x, mask=np.zeros_like(x))
 967: (4)                xm.mask[0, 0, 0] = True
 968: (4)                expected = x.copy() if bytes else x.astype(np.float32)/255
 969: (4)                expected[0, 0, 3] = 0
 970: (4)                np.testing.assert_almost_equal(sm.to_rgba(xm, bytes=bytes), expected)
 971: (4)                expected[..., 3] = alpha_1
 972: (4)                expected[0, 0, 3] = 0
 973: (4)                np.testing.assert_almost_equal(sm.to_rgba(xm[..., :3], bytes=bytes),
 expected)
 974: (4)                x = np.ones((2, 3, 4), dtype=float) * 0.5
```

```
975: (4)                expected = (x * 255).astype(np.uint8) if bytes else x.copy()
976: (4)                np.testing.assert_almost_equal(sm.to_rgba(x, bytes=bytes), expected)
977: (4)                expected[..., 3] = alpha_1
978: (4)                np.testing.assert_almost_equal(sm.to_rgba(x[..., :3], bytes=bytes),
expected)
979: (4)                xm = np.ma.masked_array(x, mask=np.zeros_like(x))
980: (4)                xm.mask[0, 0, 0] = True
981: (4)                expected = (x * 255).astype(np.uint8) if bytes else x.copy()
982: (4)                expected[0, 0, 3] = 0
983: (4)                np.testing.assert_almost_equal(sm.to_rgba(xm, bytes=bytes), expected)
984: (4)                expected[..., 3] = alpha_1
985: (4)                expected[0, 0, 3] = 0
986: (4)                np.testing.assert_almost_equal(sm.to_rgba(xm[..., :3], bytes=bytes),
expected)
987: (0)            @pytest.mark.parametrize("bytes", (True, False))
988: (0)            def test_scalarmappable_nan_to_rgba(bytes):
989: (4)                sm = cm.ScalarMappable()
990: (4)                x = np.ones((2, 3, 4), dtype=float) * 0.5
991: (4)                x[0, 0, 0] = np.nan
992: (4)                expected = x.copy()
993: (4)                expected[0, 0, :] = 0
994: (4)                if bytes:
995: (8)                    expected = (expected * 255).astype(np.uint8)
996: (4)                np.testing.assert_almost_equal(sm.to_rgba(x, bytes=bytes), expected)
997: (4)                assert np.any(np.isnan(x))  # Input array should not be changed
998: (4)                expected[..., 3] = 255 if bytes else 1
999: (4)                expected[0, 0, 3] = 0
1000: (4)               np.testing.assert_almost_equal(sm.to_rgba(x[..., :3], bytes=bytes),
expected)
1001: (4)               assert np.any(np.isnan(x))  # Input array should not be changed
1002: (4)               x[1, 0, 0] = 42
1003: (4)               with pytest.raises(ValueError, match='0..1 range'):
1004: (8)                   sm.to_rgba(x[..., :3], bytes=bytes)
1005: (0)           def test_failed_conversions():
1006: (4)               with pytest.raises(ValueError):
1007: (8)                   mcolors.to_rgba('5')
1008: (4)               with pytest.raises(ValueError):
1009: (8)                   mcolors.to_rgba('-1')
1010: (4)               with pytest.raises(ValueError):
1011: (8)                   mcolors.to_rgba('nan')
1012: (4)               with pytest.raises(ValueError):
1013: (8)                   mcolors.to_rgba('unknown_color')
1014: (4)               with pytest.raises(ValueError):
1015: (8)                   mcolors.to_rgba(0.4)
1016: (0)           def test_grey_gray():
1017: (4)               color_mapping = mcolors._colors_full_map
1018: (4)               for k in color_mapping.keys():
1019: (8)                   if 'grey' in k:
1020: (12)                      assert color_mapping[k] == color_mapping[k.replace('grey',
'gray')]
1021: (8)                   if 'gray' in k:
1022: (12)                      assert color_mapping[k] == color_mapping[k.replace('gray',
'grey')]
1023: (0)           def test_tableau_order():
1024: (4)               dflt_cycle = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728',
1025: (18)                            '#9467bd', '#8c564b', '#e377c2', '#7f7f7f',
1026: (18)                            '#bcbd22', '#17becf']
1027: (4)               assert list(mcolors.TABLEAU_COLORS.values()) == dflt_cycle
1028: (0)           def test_ndarray_subclass_norm():
1029: (4)               class MyArray(np.ndarray):
1030: (8)                   def __isub__(self, other):  # type: ignore
1031: (12)                      raise RuntimeError
1032: (8)                   def __add__(self, other):
1033: (12)                      raise RuntimeError
1034: (4)               data = np.arange(-10, 10, 1, dtype=float).reshape((10, 2))
1035: (4)               mydata = data.view(MyArray)
1036: (4)               for norm in [mcolors.Normalize(), mcolors.LogNorm(),
1037: (17)                           mcolors.SymLogNorm(3, vmax=5, linscale=1, base=np.e),
1038: (17)                           mcolors.Normalize(vmin=mydata.min(), vmax=mydata.max()),
```

```
(17)                            mcolors.SymLogNorm(3, vmin=mydata.min(), vmax=mydata.max(),
(36)                                               base=np.e),
(17)                            mcolors.PowerNorm(1)]:
(8)                     assert_array_equal(norm(mydata), norm(data))
(8)                     fig, ax = plt.subplots()
(8)                     ax.imshow(mydata, norm=norm)
(8)                     fig.canvas.draw()  # Check that no warning is emitted.
(0)             def test_same_color():
(4)                 assert mcolors.same_color('k', (0, 0, 0))
(4)                 assert not mcolors.same_color('w', (1, 1, 0))
(4)                 assert mcolors.same_color(['red', 'blue'], ['r', 'b'])
(4)                 assert mcolors.same_color('none', 'none')
(4)                 assert not mcolors.same_color('none', 'red')
(4)                 with pytest.raises(ValueError):
(8)                     mcolors.same_color(['r', 'g', 'b'], ['r'])
(4)                 with pytest.raises(ValueError):
(8)                     mcolors.same_color(['red', 'green'], 'none')
(0)             def test_hex_shorthand_notation():
(4)                 assert mcolors.same_color("#123", "#112233")
(4)                 assert mcolors.same_color("#123a", "#112233aa")
(0)             def test_repr_png():
(4)                 cmap = mpl.colormaps['viridis']
(4)                 png = cmap._repr_png_()
(4)                 assert len(png) > 0
(4)                 img = Image.open(BytesIO(png))
(4)                 assert img.width > 0
(4)                 assert img.height > 0
(4)                 assert 'Title' in img.text
(4)                 assert 'Description' in img.text
(4)                 assert 'Author' in img.text
(4)                 assert 'Software' in img.text
(0)             def test_repr_html():
(4)                 cmap = mpl.colormaps['viridis']
(4)                 html = cmap._repr_html_()
(4)                 assert len(html) > 0
(4)                 png = cmap._repr_png_()
(4)                 assert base64.b64encode(png).decode('ascii') in html
(4)                 assert cmap.name in html
(4)                 assert html.startswith('<div')
(4)                 assert html.endswith('</div>')
(0)             def test_get_under_over_bad():
(4)                 cmap = mpl.colormaps['viridis']
(4)                 assert_array_equal(cmap.get_under(), cmap(-np.inf))
(4)                 assert_array_equal(cmap.get_over(), cmap(np.inf))
(4)                 assert_array_equal(cmap.get_bad(), cmap(np.nan))
(0)             @pytest.mark.parametrize('kind', ('over', 'under', 'bad'))
(0)             def test_non_mutable_get_values(kind):
(4)                 cmap = copy.copy(mpl.colormaps['viridis'])
(4)                 init_value = getattr(cmap, f'get_{kind}')()
(4)                 getattr(cmap, f'set_{kind}')('k')
(4)                 black_value = getattr(cmap, f'get_{kind}')()
(4)                 assert np.all(black_value == [0, 0, 0, 1])
(4)                 assert not np.all(init_value == black_value)
(0)             def test_colormap_alpha_array():
(4)                 cmap = mpl.colormaps['viridis']
(4)                 vals = [-1, 0.5, 2]  # under, valid, over
(4)                 with pytest.raises(ValueError, match="alpha is array-like but"):
(8)                     cmap(vals, alpha=[1, 1, 1, 1])
(4)                 alpha = np.array([0.1, 0.2, 0.3])
(4)                 c = cmap(vals, alpha=alpha)
(4)                 assert_array_equal(c[:, -1], alpha)
(4)                 c = cmap(vals, alpha=alpha, bytes=True)
(4)                 assert_array_equal(c[:, -1], (alpha * 255).astype(np.uint8))
(0)             def test_colormap_bad_data_with_alpha():
(4)                 cmap = mpl.colormaps['viridis']
(4)                 c = cmap(np.nan, alpha=0.5)
(4)                 assert c == (0, 0, 0, 0)
(4)                 c = cmap([0.5, np.nan], alpha=0.5)
(4)                 assert_array_equal(c[1], (0, 0, 0, 0))
```

```
(4)             c = cmap([0.5, np.nan], alpha=[0.1, 0.2])
(4)             assert_array_equal(c[1], (0, 0, 0, 0))
(4)             c = cmap([[np.nan, 0.5], [0, 0]], alpha=0.5)
(4)             assert_array_equal(c[0, 0], (0, 0, 0, 0))
(4)             c = cmap([[np.nan, 0.5], [0, 0]], alpha=np.full((2, 2), 0.5))
(4)             assert_array_equal(c[0, 0], (0, 0, 0, 0))
(0)         def test_2d_to_rgba():
(4)             color = np.array([0.1, 0.2, 0.3])
(4)             rgba_1d = mcolors.to_rgba(color.reshape(-1))
(4)             rgba_2d = mcolors.to_rgba(color.reshape((1, -1)))
(4)             assert rgba_1d == rgba_2d
(0)         def test_set_dict_to_rgba():
(4)             colors = {(0, .5, 1), (1, .2, .5), (.4, 1, .2)}
(4)             res = mcolors.to_rgba_array(colors)
(4)             palette = {"red": (1, 0, 0), "green": (0, 1, 0), "blue": (0, 0, 1)}
(4)             res = mcolors.to_rgba_array(palette.values())
(4)             exp = np.eye(3)
(4)             np.testing.assert_array_almost_equal(res[:, :-1], exp)
(0)         def test_norm_deepcopy():
(4)             norm = mcolors.LogNorm()
(4)             norm.vmin = 0.0002
(4)             norm2 = copy.deepcopy(norm)
(4)             assert norm2.vmin == norm.vmin
(4)             assert isinstance(norm2._scale, mscale.LogScale)
(4)             norm = mcolors.Normalize()
(4)             norm.vmin = 0.0002
(4)             norm2 = copy.deepcopy(norm)
(4)             assert norm2._scale is None
(4)             assert norm2.vmin == norm.vmin
(0)         def test_norm_callback():
(4)             increment = unittest.mock.Mock(return_value=None)
(4)             norm = mcolors.Normalize()
(4)             norm.callbacks.connect('changed', increment)
(4)             assert increment.call_count == 0
(4)             norm.vmin = 1
(4)             assert increment.call_count == 1
(4)             norm.vmax = 5
(4)             assert increment.call_count == 2
(4)             norm.vmin = 1
(4)             assert increment.call_count == 2
(4)             norm.vmax = 5
(4)             assert increment.call_count == 2
(4)             increment.call_count = 0
(4)             norm.autoscale([0, 1, 2])
(4)             assert increment.call_count == 1
(0)         def test_scalarmappable_norm_update():
(4)             norm = mcolors.Normalize()
(4)             sm = matplotlib.cm.ScalarMappable(norm=norm, cmap='plasma')
(4)             sm.stale = False
(4)             norm.vmin = 5
(4)             assert sm.stale
(4)             sm.stale = False
(4)             norm.vmax = 5
(4)             assert sm.stale
(4)             sm.stale = False
(4)             norm.clip = True
(4)             assert sm.stale
(4)             norm = mcolors.CenteredNorm()
(4)             sm.norm = norm
(4)             sm.stale = False
(4)             norm.vcenter = 1
(4)             assert sm.stale
(4)             norm = mcolors.TwoSlopeNorm(vcenter=0, vmin=-1, vmax=1)
(4)             sm.set_norm(norm)
(4)             sm.stale = False
(4)             norm.vcenter = 1
(4)             assert sm.stale
(0)         @check_figures_equal()
(0)         def test_norm_update_figs(fig_test, fig_ref):
```

```
(4)             ax_ref = fig_ref.add_subplot()
(4)             ax_test = fig_test.add_subplot()
(4)             z = np.arange(100).reshape((10, 10))
(4)             ax_ref.imshow(z, norm=mcolors.Normalize(10, 90))
(4)             norm = mcolors.Normalize(0, 1)
(4)             ax_test.imshow(z, norm=norm)
(4)             fig_test.canvas.draw()
(4)             norm.vmin, norm.vmax = 10, 90
(0)         def test_make_norm_from_scale_name():
(4)             logitnorm = mcolors.make_norm_from_scale(
(8)                 mscale.LogitScale, mcolors.Normalize)
(4)             assert logitnorm.__name__ == logitnorm.__qualname__ == "LogitScaleNorm"
(0)         def test_color_sequences():
(4)             assert plt.color_sequences is matplotlib.color_sequences  # same registry
(4)             assert list(plt.color_sequences) == [
(8)                 'tab10', 'tab20', 'tab20b', 'tab20c', 'Pastel1', 'Pastel2', 'Paired',
(8)                 'Accent', 'Dark2', 'Set1', 'Set2', 'Set3']
(4)             assert len(plt.color_sequences['tab10']) == 10
(4)             assert len(plt.color_sequences['tab20']) == 20
(4)             tab_colors = [
(8)                 'tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple',
(8)                 'tab:brown', 'tab:pink', 'tab:gray', 'tab:olive', 'tab:cyan']
(4)             for seq_color, tab_color in zip(plt.color_sequences['tab10'], tab_colors):
(8)                 assert mcolors.same_color(seq_color, tab_color)
(4)             with pytest.raises(ValueError, match="reserved name"):
(8)                 plt.color_sequences.register('tab10', ['r', 'g', 'b'])
(4)             with pytest.raises(ValueError, match="not a valid color specification"):
(8)                 plt.color_sequences.register('invalid', ['not a color'])
(4)             rgb_colors = ['r', 'g', 'b']
(4)             plt.color_sequences.register('rgb', rgb_colors)
(4)             assert plt.color_sequences['rgb'] == ['r', 'g', 'b']
(4)             rgb_colors.append('c')
(4)             assert plt.color_sequences['rgb'] == ['r', 'g', 'b']
(4)             plt.color_sequences['rgb'].append('c')
(4)             assert plt.color_sequences['rgb'] == ['r', 'g', 'b']
(4)             plt.color_sequences.unregister('rgb')
(4)             with pytest.raises(KeyError):
(8)                 plt.color_sequences['rgb']  # rgb is gone
(4)             plt.color_sequences.unregister('rgb')  # multiple unregisters are ok
(4)             with pytest.raises(ValueError, match="Cannot unregister builtin"):
(8)                 plt.color_sequences.unregister('tab10')
(0)         def test_cm_set_cmap_error():
(4)             sm = cm.ScalarMappable()
(4)             bad_cmap = 'AardvarksAreAwkward'
(4)             with pytest.raises(ValueError, match=bad_cmap):
(8)                 sm.set_cmap(bad_cmap)
(0)         def test_set_cmap_mismatched_name():
(4)             cmap = matplotlib.colormaps["viridis"].with_extremes(over='r')
(4)             cmap.name = "test-cmap"
(4)             matplotlib.colormaps.register(name='wrong-cmap', cmap=cmap)
(4)             plt.set_cmap("wrong-cmap")
(4)             cmap_returned = plt.get_cmap("wrong-cmap")
(4)             assert cmap_returned == cmap
(4)             assert cmap_returned.name == "wrong-cmap"
(0)         def test_to_rgba_array_none_color_with_alpha_param():
(4)             c = ["blue", "none"]
(4)             alpha = [1, 1]
(4)             assert_array_equal(
(8)                 to_rgba_array(c, alpha), [[0., 0., 1., 1.], [0., 0., 0., 0.]]
(4)             )
(0)         @pytest.mark.parametrize('input, expected',
(25)                                 [('red', True),
(26)                                  (('red', 0.5), True),
(26)                                  (('red', 2), False),
(26)                                  (['red', 0.5], False),
(26)                                  (('red', 'blue'), False),
(26)                                  (['red', 'blue'], False),
(26)                                  ('C3', True),
(26)                                  (('C3', 0.5), True)])
```

```
1246: (0)           def test_is_color_like(input, expected):
1247: (4)               assert is_color_like(input) is expected

----------------------------------------

File 174 - test_compare_images.py:

1: (0)              from pathlib import Path
2: (0)              import shutil
3: (0)              import pytest
4: (0)              from pytest import approx
5: (0)              from matplotlib.testing.compare import compare_images
6: (0)              from matplotlib.testing.decorators import _image_directories
7: (0)              @pytest.mark.parametrize(
8: (4)                  'im1, im2, tol, expect_rms',
9: (4)                  [
10: (8)                     ('basn3p02.png', 'basn3p02-minorchange.png', 10, None),
11: (8)                     ('basn3p02.png', 'basn3p02-minorchange.png', 0, 6.50646),
12: (8)                     ('basn3p02.png', 'basn3p02-1px-offset.png', 0, 90.15611),
13: (8)                     ('basn3p02.png', 'basn3p02-half-1px-offset.png', 0, 63.75),
14: (8)                     ('basn3p02.png', 'basn3p02-scrambled.png', 0, 172.63582),
15: (8)                     ('all127.png', 'all128.png', 0, 1),
16: (8)                     ('all128.png', 'all127.png', 0, 1),
17: (4)                 ])
18: (0)             def test_image_comparison_expect_rms(im1, im2, tol, expect_rms, tmp_path,
19: (37)                                                 monkeypatch):
20: (4)                 """
21: (4)                 Compare two images, expecting a particular RMS error.
22: (4)                 im1 and im2 are filenames relative to the baseline_dir directory.
23: (4)                 tol is the tolerance to pass to compare_images.
24: (4)                 expect_rms is the expected RMS value, or None. If None, the test will
25: (4)                 succeed if compare_images succeeds. Otherwise, the test will succeed if
26: (4)                 compare_images fails and returns an RMS error almost equal to this value.
27: (4)                 """
28: (4)                 monkeypatch.chdir(tmp_path)
29: (4)                 baseline_dir, result_dir = map(Path, _image_directories(lambda: "dummy"))
30: (4)                 result_im2 = result_dir / im1
31: (4)                 shutil.copyfile(baseline_dir / im2, result_im2)
32: (4)                 results = compare_images(
33: (8)                     baseline_dir / im1, result_im2, tol=tol, in_decorator=True)
34: (4)                 if expect_rms is None:
35: (8)                     assert results is None
36: (4)                 else:
37: (8)                     assert results is not None
38: (8)                     assert results['rms'] == approx(expect_rms, abs=1e-4)

----------------------------------------

File 175 - test_constrainedlayout.py:

1: (0)              import gc
2: (0)              import platform
3: (0)              import numpy as np
4: (0)              import pytest
5: (0)              import matplotlib as mpl
6: (0)              from matplotlib.testing.decorators import image_comparison
7: (0)              import matplotlib.pyplot as plt
8: (0)              import matplotlib.transforms as mtransforms
9: (0)              from matplotlib import gridspec, ticker
10: (0)             def example_plot(ax, fontsize=12, nodec=False):
11: (4)                 ax.plot([1, 2])
12: (4)                 ax.locator_params(nbins=3)
13: (4)                 if not nodec:
14: (8)                     ax.set_xlabel('x-label', fontsize=fontsize)
15: (8)                     ax.set_ylabel('y-label', fontsize=fontsize)
16: (8)                     ax.set_title('Title', fontsize=fontsize)
17: (4)                 else:
18: (8)                     ax.set_xticklabels([])
19: (8)                     ax.set_yticklabels([])
```

```
(0)  def example_pcolor(ax, fontsize=12):
(4)      dx, dy = 0.6, 0.6
(4)      y, x = np.mgrid[slice(-3, 3 + dy, dy),
(20)                     slice(-3, 3 + dx, dx)]
(4)      z = (1 - x / 2. + x ** 5 + y ** 3) * np.exp(-x ** 2 - y ** 2)
(4)      pcm = ax.pcolormesh(x, y, z[:-1, :-1], cmap='RdBu_r', vmin=-1., vmax=1.,
(24)                         rasterized=True)
(4)      ax.set_xlabel('x-label', fontsize=fontsize)
(4)      ax.set_ylabel('y-label', fontsize=fontsize)
(4)      ax.set_title('Title', fontsize=fontsize)
(4)      return pcm
(0)  @image_comparison(['constrained_layout1.png'])
(0)  def test_constrained_layout1():
(4)      """Test constrained_layout for a single subplot"""
(4)      fig = plt.figure(layout="constrained")
(4)      ax = fig.add_subplot()
(4)      example_plot(ax, fontsize=24)
(0)  @image_comparison(['constrained_layout2.png'])
(0)  def test_constrained_layout2():
(4)      """Test constrained_layout for 2x2 subplots"""
(4)      fig, axs = plt.subplots(2, 2, layout="constrained")
(4)      for ax in axs.flat:
(8)          example_plot(ax, fontsize=24)
(0)  @image_comparison(['constrained_layout3.png'])
(0)  def test_constrained_layout3():
(4)      """Test constrained_layout for colorbars with subplots"""
(4)      fig, axs = plt.subplots(2, 2, layout="constrained")
(4)      for nn, ax in enumerate(axs.flat):
(8)          pcm = example_pcolor(ax, fontsize=24)
(8)          if nn == 3:
(12)             pad = 0.08
(8)          else:
(12)             pad = 0.02  # default
(8)          fig.colorbar(pcm, ax=ax, pad=pad)
(0)  @image_comparison(['constrained_layout4.png'])
(0)  def test_constrained_layout4():
(4)      """Test constrained_layout for a single colorbar with subplots"""
(4)      fig, axs = plt.subplots(2, 2, layout="constrained")
(4)      for ax in axs.flat:
(8)          pcm = example_pcolor(ax, fontsize=24)
(4)      fig.colorbar(pcm, ax=axs, pad=0.01, shrink=0.6)
(0)  @image_comparison(['constrained_layout5.png'], tol=0.002)
(0)  def test_constrained_layout5():
(4)      """
(4)      Test constrained_layout for a single colorbar with subplots,
(4)      colorbar bottom
(4)      """
(4)      fig, axs = plt.subplots(2, 2, layout="constrained")
(4)      for ax in axs.flat:
(8)          pcm = example_pcolor(ax, fontsize=24)
(4)      fig.colorbar(pcm, ax=axs,
(17)                  use_gridspec=False, pad=0.01, shrink=0.6,
(17)                  location='bottom')
(0)  @image_comparison(['constrained_layout6.png'], tol=0.002)
(0)  def test_constrained_layout6():
(4)      """Test constrained_layout for nested gridspecs"""
(4)      plt.rcParams['pcolormesh.snap'] = False
(4)      fig = plt.figure(layout="constrained")
(4)      gs = fig.add_gridspec(1, 2, figure=fig)
(4)      gsl = gs[0].subgridspec(2, 2)
(4)      gsr = gs[1].subgridspec(1, 2)
(4)      axsl = []
(4)      for gs in gsl:
(8)          ax = fig.add_subplot(gs)
(8)          axsl += [ax]
(8)          example_plot(ax, fontsize=12)
(4)      ax.set_xlabel('x-label\nMultiLine')
(4)      axsr = []
(4)      for gs in gsr:
```

```
 89: (8)                    ax = fig.add_subplot(gs)
 90: (8)                    axsr += [ax]
 91: (8)                    pcm = example_pcolor(ax, fontsize=12)
 92: (4)                fig.colorbar(pcm, ax=axsr,
 93: (17)                            pad=0.01, shrink=0.99, location='bottom',
 94: (17)                            ticks=ticker.MaxNLocator(nbins=5))
 95: (0)            def test_identical_subgridspec():
 96: (4)                fig = plt.figure(constrained_layout=True)
 97: (4)                GS = fig.add_gridspec(2, 1)
 98: (4)                GSA = GS[0].subgridspec(1, 3)
 99: (4)                GSB = GS[1].subgridspec(1, 3)
 100: (4)               axa = []
 101: (4)               axb = []
 102: (4)               for i in range(3):
 103: (8)                   axa += [fig.add_subplot(GSA[i])]
 104: (8)                   axb += [fig.add_subplot(GSB[i])]
 105: (4)               fig.draw_without_rendering()
 106: (4)               assert axa[0].get_position().y0 > axb[0].get_position().y1
 107: (0)           def test_constrained_layout7():
 108: (4)               """Test for proper warning if fig not set in GridSpec"""
 109: (4)               with pytest.warns(
 110: (8)                   UserWarning, match=('There are no gridspecs with layoutgrids. '
 111: (28)                                      'Possibly did not call parent GridSpec with '
 112: (28)                                      'the "figure" keyword')):
 113: (8)                   fig = plt.figure(layout="constrained")
 114: (8)                   gs = gridspec.GridSpec(1, 2)
 115: (8)                   gsl = gridspec.GridSpecFromSubplotSpec(2, 2, gs[0])
 116: (8)                   gsr = gridspec.GridSpecFromSubplotSpec(1, 2, gs[1])
 117: (8)                   for gs in gsl:
 118: (12)                      fig.add_subplot(gs)
 119: (8)                   fig.draw_without_rendering()
 120: (0)           @image_comparison(['constrained_layout8.png'])
 121: (0)           def test_constrained_layout8():
 122: (4)               """Test for gridspecs that are not completely full"""
 123: (4)               fig = plt.figure(figsize=(10, 5), layout="constrained")
 124: (4)               gs = gridspec.GridSpec(3, 5, figure=fig)
 125: (4)               axs = []
 126: (4)               for j in [0, 1]:
 127: (8)                   if j == 0:
 128: (12)                      ilist = [1]
 129: (8)                   else:
 130: (12)                      ilist = [0, 4]
 131: (8)                   for i in ilist:
 132: (12)                      ax = fig.add_subplot(gs[j, i])
 133: (12)                      axs += [ax]
 134: (12)                      example_pcolor(ax, fontsize=9)
 135: (12)                      if i > 0:
 136: (16)                          ax.set_ylabel('')
 137: (12)                      if j < 1:
 138: (16)                          ax.set_xlabel('')
 139: (12)                      ax.set_title('')
 140: (4)               ax = fig.add_subplot(gs[2, :])
 141: (4)               axs += [ax]
 142: (4)               pcm = example_pcolor(ax, fontsize=9)
 143: (4)               fig.colorbar(pcm, ax=axs, pad=0.01, shrink=0.6)
 144: (0)           @image_comparison(['constrained_layout9.png'])
 145: (0)           def test_constrained_layout9():
 146: (4)               """Test for handling suptitle and for sharex and sharey"""
 147: (4)               fig, axs = plt.subplots(2, 2, layout="constrained",
 148: (28)                                      sharex=False, sharey=False)
 149: (4)               for ax in axs.flat:
 150: (8)                   pcm = example_pcolor(ax, fontsize=24)
 151: (8)                   ax.set_xlabel('')
 152: (8)                   ax.set_ylabel('')
 153: (4)               ax.set_aspect(2.)
 154: (4)               fig.colorbar(pcm, ax=axs, pad=0.01, shrink=0.6)
 155: (4)               fig.suptitle('Test Suptitle', fontsize=28)
 156: (0)           @image_comparison(['constrained_layout10.png'],
 157: (18)                           tol=0.032 if platform.machine() == 'arm64' else 0)
```

```
158: (0)            def test_constrained_layout10():
159: (4)                """Test for handling legend outside axis"""
160: (4)                fig, axs = plt.subplots(2, 2, layout="constrained")
161: (4)                for ax in axs.flat:
162: (8)                    ax.plot(np.arange(12), label='This is a label')
163: (4)                ax.legend(loc='center left', bbox_to_anchor=(0.8, 0.5))
164: (0)            @image_comparison(['constrained_layout11.png'])
165: (0)            def test_constrained_layout11():
166: (4)                """Test for multiple nested gridspecs"""
167: (4)                fig = plt.figure(layout="constrained", figsize=(13, 3))
168: (4)                gs0 = gridspec.GridSpec(1, 2, figure=fig)
169: (4)                gsl = gridspec.GridSpecFromSubplotSpec(1, 2, gs0[0])
170: (4)                gsl0 = gridspec.GridSpecFromSubplotSpec(2, 2, gsl[1])
171: (4)                ax = fig.add_subplot(gs0[1])
172: (4)                example_plot(ax, fontsize=9)
173: (4)                axs = []
174: (4)                for gs in gsl0:
175: (8)                    ax = fig.add_subplot(gs)
176: (8)                    axs += [ax]
177: (8)                    pcm = example_pcolor(ax, fontsize=9)
178: (4)                fig.colorbar(pcm, ax=axs, shrink=0.6, aspect=70.)
179: (4)                ax = fig.add_subplot(gsl[0])
180: (4)                example_plot(ax, fontsize=9)
181: (0)            @image_comparison(['constrained_layout11rat.png'])
182: (0)            def test_constrained_layout11rat():
183: (4)                """Test for multiple nested gridspecs with width_ratios"""
184: (4)                fig = plt.figure(layout="constrained", figsize=(10, 3))
185: (4)                gs0 = gridspec.GridSpec(1, 2, figure=fig, width_ratios=[6, 1])
186: (4)                gsl = gridspec.GridSpecFromSubplotSpec(1, 2, gs0[0])
187: (4)                gsl0 = gridspec.GridSpecFromSubplotSpec(2, 2, gsl[1], height_ratios=[2,
1])
188: (4)                ax = fig.add_subplot(gs0[1])
189: (4)                example_plot(ax, fontsize=9)
190: (4)                axs = []
191: (4)                for gs in gsl0:
192: (8)                    ax = fig.add_subplot(gs)
193: (8)                    axs += [ax]
194: (8)                    pcm = example_pcolor(ax, fontsize=9)
195: (4)                fig.colorbar(pcm, ax=axs, shrink=0.6, aspect=70.)
196: (4)                ax = fig.add_subplot(gsl[0])
197: (4)                example_plot(ax, fontsize=9)
198: (0)            @image_comparison(['constrained_layout12.png'])
199: (0)            def test_constrained_layout12():
200: (4)                """Test that very unbalanced labeling still works."""
201: (4)                fig = plt.figure(layout="constrained", figsize=(6, 8))
202: (4)                gs0 = gridspec.GridSpec(6, 2, figure=fig)
203: (4)                ax1 = fig.add_subplot(gs0[:3, 1])
204: (4)                ax2 = fig.add_subplot(gs0[3:, 1])
205: (4)                example_plot(ax1, fontsize=18)
206: (4)                example_plot(ax2, fontsize=18)
207: (4)                ax = fig.add_subplot(gs0[0:2, 0])
208: (4)                example_plot(ax, nodec=True)
209: (4)                ax = fig.add_subplot(gs0[2:4, 0])
210: (4)                example_plot(ax, nodec=True)
211: (4)                ax = fig.add_subplot(gs0[4:, 0])
212: (4)                example_plot(ax, nodec=True)
213: (4)                ax.set_xlabel('x-label')
214: (0)            @image_comparison(['constrained_layout13.png'], tol=2.e-2)
215: (0)            def test_constrained_layout13():
216: (4)                """Test that padding works."""
217: (4)                fig, axs = plt.subplots(2, 2, layout="constrained")
218: (4)                for ax in axs.flat:
219: (8)                    pcm = example_pcolor(ax, fontsize=12)
220: (8)                    fig.colorbar(pcm, ax=ax, shrink=0.6, aspect=20., pad=0.02)
221: (4)                with pytest.raises(TypeError):
222: (8)                    fig.get_layout_engine().set(wpad=1, hpad=2)
223: (4)                fig.get_layout_engine().set(w_pad=24./72., h_pad=24./72.)
224: (0)            @image_comparison(['constrained_layout14.png'])
225: (0)            def test_constrained_layout14():
```

```
(4)                 """Test that padding works."""
(4)                 fig, axs = plt.subplots(2, 2, layout="constrained")
(4)                 for ax in axs.flat:
(8)                     pcm = example_pcolor(ax, fontsize=12)
(8)                     fig.colorbar(pcm, ax=ax, shrink=0.6, aspect=20., pad=0.02)
(4)                 fig.get_layout_engine().set(
(12)                        w_pad=3./72., h_pad=3./72.,
(12)                        hspace=0.2, wspace=0.2)
(0)             @image_comparison(['constrained_layout15.png'])
(0)             def test_constrained_layout15():
(4)                 """Test that rcparams work."""
(4)                 mpl.rcParams['figure.constrained_layout.use'] = True
(4)                 fig, axs = plt.subplots(2, 2)
(4)                 for ax in axs.flat:
(8)                     example_plot(ax, fontsize=12)
(0)             @image_comparison(['constrained_layout16.png'])
(0)             def test_constrained_layout16():
(4)                 """Test ax.set_position."""
(4)                 fig, ax = plt.subplots(layout="constrained")
(4)                 example_plot(ax, fontsize=12)
(4)                 ax2 = fig.add_axes([0.2, 0.2, 0.4, 0.4])
(0)             @image_comparison(['constrained_layout17.png'])
(0)             def test_constrained_layout17():
(4)                 """Test uneven gridspecs"""
(4)                 fig = plt.figure(layout="constrained")
(4)                 gs = gridspec.GridSpec(3, 3, figure=fig)
(4)                 ax1 = fig.add_subplot(gs[0, 0])
(4)                 ax2 = fig.add_subplot(gs[0, 1:])
(4)                 ax3 = fig.add_subplot(gs[1:, 0:2])
(4)                 ax4 = fig.add_subplot(gs[1:, -1])
(4)                 example_plot(ax1)
(4)                 example_plot(ax2)
(4)                 example_plot(ax3)
(4)                 example_plot(ax4)
(0)             def test_constrained_layout18():
(4)                 """Test twinx"""
(4)                 fig, ax = plt.subplots(layout="constrained")
(4)                 ax2 = ax.twinx()
(4)                 example_plot(ax)
(4)                 example_plot(ax2, fontsize=24)
(4)                 fig.draw_without_rendering()
(4)                 assert all(ax.get_position().extents == ax2.get_position().extents)
(0)             def test_constrained_layout19():
(4)                 """Test twiny"""
(4)                 fig, ax = plt.subplots(layout="constrained")
(4)                 ax2 = ax.twiny()
(4)                 example_plot(ax)
(4)                 example_plot(ax2, fontsize=24)
(4)                 ax2.set_title('')
(4)                 ax.set_title('')
(4)                 fig.draw_without_rendering()
(4)                 assert all(ax.get_position().extents == ax2.get_position().extents)
(0)             def test_constrained_layout20():
(4)                 """Smoke test cl does not mess up added Axes"""
(4)                 gx = np.linspace(-5, 5, 4)
(4)                 img = np.hypot(gx, gx[:, None])
(4)                 fig = plt.figure()
(4)                 ax = fig.add_axes([0, 0, 1, 1])
(4)                 mesh = ax.pcolormesh(gx, gx, img[:-1, :-1])
(4)                 fig.colorbar(mesh)
(0)             def test_constrained_layout21():
(4)                 """#11035: repeated calls to suptitle should not alter the layout"""
(4)                 fig, ax = plt.subplots(layout="constrained")
(4)                 fig.suptitle("Suptitle0")
(4)                 fig.draw_without_rendering()
(4)                 extents0 = np.copy(ax.get_position().extents)
(4)                 fig.suptitle("Suptitle1")
(4)                 fig.draw_without_rendering()
(4)                 extents1 = np.copy(ax.get_position().extents)
```

```
(4)                 np.testing.assert_allclose(extents0, extents1)
(0)             def test_constrained_layout22():
(4)                 """#11035: suptitle should not be include in CL if manually positioned"""
(4)                 fig, ax = plt.subplots(layout="constrained")
(4)                 fig.draw_without_rendering()
(4)                 extents0 = np.copy(ax.get_position().extents)
(4)                 fig.suptitle("Suptitle", y=0.5)
(4)                 fig.draw_without_rendering()
(4)                 extents1 = np.copy(ax.get_position().extents)
(4)                 np.testing.assert_allclose(extents0, extents1)
(0)             def test_constrained_layout23():
(4)                 """
(4)                 Comment in #11035: suptitle used to cause an exception when
(4)                 reusing a figure w/ CL with ``clear=True``.
(4)                 """
(4)                 for i in range(2):
(8)                     fig = plt.figure(layout="constrained", clear=True, num="123")
(8)                     gs = fig.add_gridspec(1, 2)
(8)                     sub = gs[0].subgridspec(2, 2)
(8)                     fig.suptitle(f"Suptitle{i}")
(0)             @image_comparison(['test_colorbar_location.png'],
(18)                              remove_text=True, style='mpl20')
(0)             def test_colorbar_location():
(4)                 """
(4)                 Test that colorbar handling is as expected for various complicated
(4)                 cases...
(4)                 """
(4)                 plt.rcParams['pcolormesh.snap'] = False
(4)                 fig, axs = plt.subplots(4, 5, layout="constrained")
(4)                 for ax in axs.flat:
(8)                     pcm = example_pcolor(ax)
(8)                     ax.set_xlabel('')
(8)                     ax.set_ylabel('')
(4)                 fig.colorbar(pcm, ax=axs[:, 1], shrink=0.4)
(4)                 fig.colorbar(pcm, ax=axs[-1, :2], shrink=0.5, location='bottom')
(4)                 fig.colorbar(pcm, ax=axs[0, 2:], shrink=0.5, location='bottom', pad=0.05)
(4)                 fig.colorbar(pcm, ax=axs[-2, 3:], shrink=0.5, location='top')
(4)                 fig.colorbar(pcm, ax=axs[0, 0], shrink=0.5, location='left')
(4)                 fig.colorbar(pcm, ax=axs[1:3, 2], shrink=0.5, location='right')
(0)             def test_hidden_axes():
(4)                 fig, axs = plt.subplots(2, 2, layout="constrained")
(4)                 axs[0, 1].set_visible(False)
(4)                 fig.draw_without_rendering()
(4)                 extents1 = np.copy(axs[0, 0].get_position().extents)
(4)                 np.testing.assert_allclose(
(8)                     extents1, [0.045552, 0.543288, 0.47819, 0.982638], rtol=1e-5)
(0)             def test_colorbar_align():
(4)                 for location in ['right', 'left', 'top', 'bottom']:
(8)                     fig, axs = plt.subplots(2, 2, layout="constrained")
(8)                     cbs = []
(8)                     for nn, ax in enumerate(axs.flat):
(12)                        ax.tick_params(direction='in')
(12)                        pc = example_pcolor(ax)
(12)                        cb = fig.colorbar(pc, ax=ax, location=location, shrink=0.6,
(30)                                          pad=0.04)
(12)                        cbs += [cb]
(12)                        cb.ax.tick_params(direction='in')
(12)                        if nn != 1:
(16)                            cb.ax.xaxis.set_ticks([])
(16)                            cb.ax.yaxis.set_ticks([])
(16)                            ax.set_xticklabels([])
(16)                            ax.set_yticklabels([])
(8)                     fig.get_layout_engine().set(w_pad=4 / 72, h_pad=4 / 72,
(36)                                                hspace=0.1, wspace=0.1)
(8)                     fig.draw_without_rendering()
(8)                     if location in ['left', 'right']:
(12)                        np.testing.assert_allclose(cbs[0].ax.get_position().x0,
(39)                                                   cbs[2].ax.get_position().x0)
(12)                        np.testing.assert_allclose(cbs[1].ax.get_position().x0,
```

```
(39)                                                cbs[3].ax.get_position().x0)
(8)                     else:
(12)                        np.testing.assert_allclose(cbs[0].ax.get_position().y0,
(39)                                                cbs[1].ax.get_position().y0)
(12)                        np.testing.assert_allclose(cbs[2].ax.get_position().y0,
(39)                                                cbs[3].ax.get_position().y0)
(0)             @image_comparison(['test_colorbars_no_overlapV.png'], style='mpl20')
(0)             def test_colorbars_no_overlapV():
(4)                 fig = plt.figure(figsize=(2, 4), layout="constrained")
(4)                 axs = fig.subplots(2, 1, sharex=True, sharey=True)
(4)                 for ax in axs:
(8)                     ax.yaxis.set_major_formatter(ticker.NullFormatter())
(8)                     ax.tick_params(axis='both', direction='in')
(8)                     im = ax.imshow([[1, 2], [3, 4]])
(8)                     fig.colorbar(im, ax=ax, orientation="vertical")
(4)                 fig.suptitle("foo")
(0)             @image_comparison(['test_colorbars_no_overlapH.png'], style='mpl20')
(0)             def test_colorbars_no_overlapH():
(4)                 fig = plt.figure(figsize=(4, 2), layout="constrained")
(4)                 fig.suptitle("foo")
(4)                 axs = fig.subplots(1, 2, sharex=True, sharey=True)
(4)                 for ax in axs:
(8)                     ax.yaxis.set_major_formatter(ticker.NullFormatter())
(8)                     ax.tick_params(axis='both', direction='in')
(8)                     im = ax.imshow([[1, 2], [3, 4]])
(8)                     fig.colorbar(im, ax=ax, orientation="horizontal")
(0)             def test_manually_set_position():
(4)                 fig, axs = plt.subplots(1, 2, layout="constrained")
(4)                 axs[0].set_position([0.2, 0.2, 0.3, 0.3])
(4)                 fig.draw_without_rendering()
(4)                 pp = axs[0].get_position()
(4)                 np.testing.assert_allclose(pp, [[0.2, 0.2], [0.5, 0.5]])
(4)                 fig, axs = plt.subplots(1, 2, layout="constrained")
(4)                 axs[0].set_position([0.2, 0.2, 0.3, 0.3])
(4)                 pc = axs[0].pcolormesh(np.random.rand(20, 20))
(4)                 fig.colorbar(pc, ax=axs[0])
(4)                 fig.draw_without_rendering()
(4)                 pp = axs[0].get_position()
(4)                 np.testing.assert_allclose(pp, [[0.2, 0.2], [0.44, 0.5]])
(0)             @image_comparison(['test_bboxtight.png'],
(18)                            remove_text=True, style='mpl20',
(18)                            savefig_kwarg={'bbox_inches': 'tight'})
(0)             def test_bboxtight():
(4)                 fig, ax = plt.subplots(layout="constrained")
(4)                 ax.set_aspect(1.)
(0)             @image_comparison(['test_bbox.png'],
(18)                            remove_text=True, style='mpl20',
(18)                            savefig_kwarg={'bbox_inches':
(33)                                           mtransforms.Bbox([[0.5, 0], [2.5, 2]])})
(0)             def test_bbox():
(4)                 fig, ax = plt.subplots(layout="constrained")
(4)                 ax.set_aspect(1.)
(0)             def test_align_labels():
(4)                 """
(4)                 Tests for a bug in which constrained layout and align_ylabels on
(4)                 three unevenly sized subplots, one of whose y tick labels include
(4)                 negative numbers, drives the non-negative subplots' y labels off
(4)                 the edge of the plot
(4)                 """
(4)                 fig, (ax3, ax1, ax2) = plt.subplots(3, 1, layout="constrained",
(40)                                                    figsize=(6.4, 8),
(40)                                                    gridspec_kw={"height_ratios": (1, 1,
(71)                                                                                      0.7)})
(4)                 ax1.set_ylim(0, 1)
(4)                 ax1.set_ylabel("Label")
(4)                 ax2.set_ylim(-1.5, 1.5)
(4)                 ax2.set_ylabel("Label")
(4)                 ax3.set_ylim(0, 1)
(4)                 ax3.set_ylabel("Label")
```

```
433: (4)                fig.align_ylabels(axs=(ax3, ax1, ax2))
434: (4)                fig.draw_without_rendering()
435: (4)                after_align = [ax1.yaxis.label.get_window_extent(),
436: (19)                              ax2.yaxis.label.get_window_extent(),
437: (19)                              ax3.yaxis.label.get_window_extent()]
438: (4)                np.testing.assert_allclose([after_align[0].x0, after_align[2].x0],
439: (31)                                          after_align[1].x0, rtol=0, atol=1e-05)
440: (4)                assert after_align[0].x0 >= 1
441: (0)            def test_suplabels():
442: (4)                fig, ax = plt.subplots(layout="constrained")
443: (4)                fig.draw_without_rendering()
444: (4)                pos0 = ax.get_tightbbox(fig.canvas.get_renderer())
445: (4)                fig.supxlabel('Boo')
446: (4)                fig.supylabel('Booy')
447: (4)                fig.draw_without_rendering()
448: (4)                pos = ax.get_tightbbox(fig.canvas.get_renderer())
449: (4)                assert pos.y0 > pos0.y0 + 10.0
450: (4)                assert pos.x0 > pos0.x0 + 10.0
451: (4)                fig, ax = plt.subplots(layout="constrained")
452: (4)                fig.draw_without_rendering()
453: (4)                pos0 = ax.get_tightbbox(fig.canvas.get_renderer())
454: (4)                fig.supxlabel('Boo', x=0.5)
455: (4)                fig.supylabel('Boo', y=0.5)
456: (4)                fig.draw_without_rendering()
457: (4)                pos = ax.get_tightbbox(fig.canvas.get_renderer())
458: (4)                assert pos.y0 > pos0.y0 + 10.0
459: (4)                assert pos.x0 > pos0.x0 + 10.0
460: (0)            def test_gridspec_addressing():
461: (4)                fig = plt.figure()
462: (4)                gs = fig.add_gridspec(3, 3)
463: (4)                sp = fig.add_subplot(gs[0:, 1:])
464: (4)                fig.draw_without_rendering()
465: (0)            def test_discouraged_api():
466: (4)                fig, ax = plt.subplots(constrained_layout=True)
467: (4)                fig.draw_without_rendering()
468: (4)                with pytest.warns(PendingDeprecationWarning,
469: (22)                                  match="will be deprecated"):
470: (8)                    fig, ax = plt.subplots()
471: (8)                    fig.set_constrained_layout(True)
472: (8)                    fig.draw_without_rendering()
473: (4)                with pytest.warns(PendingDeprecationWarning,
474: (22)                                  match="will be deprecated"):
475: (8)                    fig, ax = plt.subplots()
476: (8)                    fig.set_constrained_layout({'w_pad': 0.02, 'h_pad': 0.02})
477: (8)                    fig.draw_without_rendering()
478: (0)            def test_kwargs():
479: (4)                fig, ax = plt.subplots(constrained_layout={'h_pad': 0.02})
480: (4)                fig.draw_without_rendering()
481: (0)            def test_rect():
482: (4)                fig, ax = plt.subplots(layout='constrained')
483: (4)                fig.get_layout_engine().set(rect=[0, 0, 0.5, 0.5])
484: (4)                fig.draw_without_rendering()
485: (4)                ppos = ax.get_position()
486: (4)                assert ppos.x1 < 0.5
487: (4)                assert ppos.y1 < 0.5
488: (4)                fig, ax = plt.subplots(layout='constrained')
489: (4)                fig.get_layout_engine().set(rect=[0.2, 0.2, 0.3, 0.3])
490: (4)                fig.draw_without_rendering()
491: (4)                ppos = ax.get_position()
492: (4)                assert ppos.x1 < 0.5
493: (4)                assert ppos.y1 < 0.5
494: (4)                assert ppos.x0 > 0.2
495: (4)                assert ppos.y0 > 0.2
496: (0)            def test_compressed1():
497: (4)                fig, axs = plt.subplots(3, 2, layout='compressed',
498: (28)                                       sharex=True, sharey=True)
499: (4)                for ax in axs.flat:
500: (8)                    pc = ax.imshow(np.random.randn(20, 20))
501: (4)                fig.colorbar(pc, ax=axs)
```

```
502: (4)                fig.draw_without_rendering()
503: (4)                pos = axs[0, 0].get_position()
504: (4)                np.testing.assert_allclose(pos.x0, 0.2344, atol=1e-3)
505: (4)                pos = axs[0, 1].get_position()
506: (4)                np.testing.assert_allclose(pos.x1, 0.7024, atol=1e-3)
507: (4)                fig, axs = plt.subplots(2, 3, layout='compressed',
508: (28)                                        sharex=True, sharey=True, figsize=(5, 4))
509: (4)                for ax in axs.flat:
510: (8)                    pc = ax.imshow(np.random.randn(20, 20))
511: (4)                fig.colorbar(pc, ax=axs)
512: (4)                fig.draw_without_rendering()
513: (4)                pos = axs[0, 0].get_position()
514: (4)                np.testing.assert_allclose(pos.x0, 0.06195, atol=1e-3)
515: (4)                np.testing.assert_allclose(pos.y1, 0.8537, atol=1e-3)
516: (4)                pos = axs[1, 2].get_position()
517: (4)                np.testing.assert_allclose(pos.x1, 0.8618, atol=1e-3)
518: (4)                np.testing.assert_allclose(pos.y0, 0.1934, atol=1e-3)
519: (0)            @pytest.mark.parametrize('arg, state', [
520: (4)                (True, True),
521: (4)                (False, False),
522: (4)                ({}, True),
523: (4)                ({'rect': None}, True)
524: (0)            ])
525: (0)            def test_set_constrained_layout(arg, state):
526: (4)                fig, ax = plt.subplots(constrained_layout=arg)
527: (4)                assert fig.get_constrained_layout() is state
528: (0)            def test_constrained_toggle():
529: (4)                fig, ax = plt.subplots()
530: (4)                with pytest.warns(PendingDeprecationWarning):
531: (8)                    fig.set_constrained_layout(True)
532: (8)                    assert fig.get_constrained_layout()
533: (8)                    fig.set_constrained_layout(False)
534: (8)                    assert not fig.get_constrained_layout()
535: (8)                    fig.set_constrained_layout(True)
536: (8)                    assert fig.get_constrained_layout()
537: (0)            def test_layout_leak():
538: (4)                fig = plt.figure(constrained_layout=True, figsize=(10, 10))
539: (4)                fig.add_subplot()
540: (4)                fig.draw_without_rendering()
541: (4)                plt.close("all")
542: (4)                del fig
543: (4)                gc.collect()
544: (4)                assert not any(isinstance(obj, mpl._layoutgrid.LayoutGrid)
545: (19)                               for obj in gc.get_objects())

----------------------------------------

File 176 - test_container.py:

1: (0)              import numpy as np
2: (0)              import matplotlib.pyplot as plt
3: (0)              def test_stem_remove():
4: (4)                  ax = plt.gca()
5: (4)                  st = ax.stem([1, 2], [1, 2])
6: (4)                  st.remove()
7: (0)              def test_errorbar_remove():
8: (4)                  ax = plt.gca()
9: (4)                  eb = ax.errorbar([1], [1])
10: (4)                 eb.remove()
11: (4)                 eb = ax.errorbar([1], [1], xerr=1)
12: (4)                 eb.remove()
13: (4)                 eb = ax.errorbar([1], [1], yerr=2)
14: (4)                 eb.remove()
15: (4)                 eb = ax.errorbar([1], [1], xerr=[2], yerr=2)
16: (4)                 eb.remove()
17: (4)                 eb = ax.errorbar([1], [1], fmt='none')
18: (4)                 eb.remove()
19: (0)             def test_nonstring_label():
20: (4)                 plt.bar(np.arange(10), np.random.rand(10), label=1)
```

```
21: (4)                 plt.legend()

----------------------------------------

File 177 - test_contour.py:

1: (0)              import datetime
2: (0)              import platform
3: (0)              import re
4: (0)              from unittest import mock
5: (0)              import contourpy
6: (0)              import numpy as np
7: (0)              from numpy.testing import (
8: (4)                  assert_array_almost_equal, assert_array_almost_equal_nulp,
assert_array_equal)
9: (0)              import matplotlib as mpl
10: (0)             from matplotlib import pyplot as plt, rc_context, ticker
11: (0)             from matplotlib.colors import LogNorm, same_color
12: (0)             import matplotlib.patches as mpatches
13: (0)             from matplotlib.testing.decorators import check_figures_equal,
image_comparison
14: (0)             import pytest
15: (0)             def _maybe_split_collections(do_split):
16: (4)                 if not do_split:
17: (8)                     return
18: (4)                 for fig in map(plt.figure, plt.get_fignums()):
19: (8)                     for ax in fig.axes:
20: (12)                        for coll in ax.collections:
21: (16)                            if isinstance(coll, mpl.contour.ContourSet):
22: (20)                                with pytest.warns(mpl._api.MatplotlibDeprecationWarning):
23: (24)                                    coll.collections
24: (0)             def test_contour_shape_1d_valid():
25: (4)                 x = np.arange(10)
26: (4)                 y = np.arange(9)
27: (4)                 z = np.random.random((9, 10))
28: (4)                 fig, ax = plt.subplots()
29: (4)                 ax.contour(x, y, z)
30: (0)             def test_contour_shape_2d_valid():
31: (4)                 x = np.arange(10)
32: (4)                 y = np.arange(9)
33: (4)                 xg, yg = np.meshgrid(x, y)
34: (4)                 z = np.random.random((9, 10))
35: (4)                 fig, ax = plt.subplots()
36: (4)                 ax.contour(xg, yg, z)
37: (0)             @pytest.mark.parametrize("args, message", [
38: (4)                 ((np.arange(9), np.arange(9), np.empty((9, 10))),
39: (5)                  'Length of x (9) must match number of columns in z (10)'),
40: (4)                 ((np.arange(10), np.arange(10), np.empty((9, 10))),
41: (5)                  'Length of y (10) must match number of rows in z (9)'),
42: (4)                 ((np.empty((10, 10)), np.arange(10), np.empty((9, 10))),
43: (5)                  'Number of dimensions of x (2) and y (1) do not match'),
44: (4)                 ((np.arange(10), np.empty((10, 10)), np.empty((9, 10))),
45: (5)                  'Number of dimensions of x (1) and y (2) do not match'),
46: (4)                 ((np.empty((9, 9)), np.empty((9, 10)), np.empty((9, 10))),
47: (5)                  'Shapes of x (9, 9) and z (9, 10) do not match'),
48: (4)                 ((np.empty((9, 10)), np.empty((9, 9)), np.empty((9, 10))),
49: (5)                  'Shapes of y (9, 9) and z (9, 10) do not match'),
50: (4)                 ((np.empty((3, 3, 3)), np.empty((3, 3, 3)), np.empty((9, 10))),
51: (5)                  'Inputs x and y must be 1D or 2D, not 3D'),
52: (4)                 ((np.empty((3, 3, 3)), np.empty((3, 3, 3)), np.empty((3, 3, 3))),
53: (5)                  'Input z must be 2D, not 3D'),
54: (4)                 (([[0]],),  # github issue 8197
55: (5)                  'Input z must be at least a (2, 2) shaped array, but has shape (1, 1)'),
56: (4)                 (([0], [0], [[0]]),
57: (5)                  'Input z must be at least a (2, 2) shaped array, but has shape (1, 1)'),
58: (0)             ])
59: (0)             def test_contour_shape_error(args, message):
60: (4)                 fig, ax = plt.subplots()
61: (4)                 with pytest.raises(TypeError, match=re.escape(message)):
```

```
 62: (8)                    ax.contour(*args)
 63: (0)             def test_contour_no_valid_levels():
 64: (4)                 fig, ax = plt.subplots()
 65: (4)                 ax.contour(np.random.rand(9, 9), levels=[])
 66: (4)                 cs = ax.contour(np.arange(81).reshape((9, 9)), levels=[100])
 67: (4)                 ax.clabel(cs, fmt={100: '%1.2f'})
 68: (4)                 ax.contour(np.ones((9, 9)))
 69: (0)             def test_contour_Nlevels():
 70: (4)                 z = np.arange(12).reshape((3, 4))
 71: (4)                 fig, ax = plt.subplots()
 72: (4)                 cs1 = ax.contour(z, 5)
 73: (4)                 assert len(cs1.levels) > 1
 74: (4)                 cs2 = ax.contour(z, levels=5)
 75: (4)                 assert (cs1.levels == cs2.levels).all()
 76: (0)             @check_figures_equal(extensions=['png'])
 77: (0)             def test_contour_set_paths(fig_test, fig_ref):
 78: (4)                 cs_test = fig_test.subplots().contour([[0, 1], [1, 2]])
 79: (4)                 cs_ref = fig_ref.subplots().contour([[1, 0], [2, 1]])
 80: (4)                 cs_test.set_paths(cs_ref.get_paths())
 81: (0)             @pytest.mark.parametrize("split_collections", [False, True])
 82: (0)             @image_comparison(['contour_manual_labels'], remove_text=True, style='mpl20',
tol=0.26)
 83: (0)             def test_contour_manual_labels(split_collections):
 84: (4)                 x, y = np.meshgrid(np.arange(0, 10), np.arange(0, 10))
 85: (4)                 z = np.max(np.dstack([abs(x), abs(y)]), 2)
 86: (4)                 plt.figure(figsize=(6, 2), dpi=200)
 87: (4)                 cs = plt.contour(x, y, z)
 88: (4)                 _maybe_split_collections(split_collections)
 89: (4)                 pts = np.array([(1.0, 3.0), (1.0, 4.4), (1.0, 6.0)])
 90: (4)                 plt.clabel(cs, manual=pts)
 91: (4)                 pts = np.array([(2.0, 3.0), (2.0, 4.4), (2.0, 6.0)])
 92: (4)                 plt.clabel(cs, manual=pts, fontsize='small', colors=('r', 'g'))
 93: (0)             def test_contour_manual_moveto():
 94: (4)                 x = np.linspace(-10, 10)
 95: (4)                 y = np.linspace(-10, 10)
 96: (4)                 X, Y = np.meshgrid(x, y)
 97: (4)                 Z = X**2 * 1 / Y**2 - 1
 98: (4)                 contours = plt.contour(X, Y, Z, levels=[0, 100])
 99: (4)                 point = (1.3, 1)
100: (4)                 clabels = plt.clabel(contours, manual=[point])
101: (4)                 assert clabels[0].get_text() == "0"
102: (0)             @pytest.mark.parametrize("split_collections", [False, True])
103: (0)             @image_comparison(['contour_disconnected_segments'],
104: (18)                              remove_text=True, style='mpl20', extensions=['png'])
105: (0)             def test_contour_label_with_disconnected_segments(split_collections):
106: (4)                 x, y = np.mgrid[-1:1:21j, -1:1:21j]
107: (4)                 z = 1 / np.sqrt(0.01 + (x + 0.3) ** 2 + y ** 2)
108: (4)                 z += 1 / np.sqrt(0.01 + (x - 0.3) ** 2 + y ** 2)
109: (4)                 plt.figure()
110: (4)                 cs = plt.contour(x, y, z, levels=[7])
111: (4)                 _maybe_split_collections(split_collections)
112: (4)                 cs.clabel(manual=[(0.2, 0.1)])
113: (4)                 _maybe_split_collections(split_collections)
114: (0)             @pytest.mark.parametrize("split_collections", [False, True])
115: (0)             @image_comparison(['contour_manual_colors_and_levels.png'], remove_text=True,
116: (18)                              tol=0.018 if platform.machine() == 'arm64' else 0)
117: (0)             def test_given_colors_levels_and_extends(split_collections):
118: (4)                 plt.rcParams['pcolormesh.snap'] = False
119: (4)                 _, axs = plt.subplots(2, 4)
120: (4)                 data = np.arange(12).reshape(3, 4)
121: (4)                 colors = ['red', 'yellow', 'pink', 'blue', 'black']
122: (4)                 levels = [2, 4, 8, 10]
123: (4)                 for i, ax in enumerate(axs.flat):
124: (8)                     filled = i % 2 == 0.
125: (8)                     extend = ['neither', 'min', 'max', 'both'][i // 2]
126: (8)                     if filled:
127: (12)                        first_color = 1 if extend in ['max', 'neither'] else None
128: (12)                        last_color = -1 if extend in ['min', 'neither'] else None
129: (12)                        c = ax.contourf(data, colors=colors[first_color:last_color],
```

```
(28)                                        levels=levels, extend=extend)
(8)                     else:
(12)                        c = ax.contour(data, colors=colors[:-1],
(27)                                       levels=levels, extend=extend)
(8)                     plt.colorbar(c, ax=ax)
(4)                 _maybe_split_collections(split_collections)
(0)             @pytest.mark.parametrize("split_collections", [False, True])
(0)             @image_comparison(['contour_log_locator.svg'], style='mpl20',
remove_text=False)
(0)             def test_log_locator_levels(split_collections):
(4)                 fig, ax = plt.subplots()
(4)                 N = 100
(4)                 x = np.linspace(-3.0, 3.0, N)
(4)                 y = np.linspace(-2.0, 2.0, N)
(4)                 X, Y = np.meshgrid(x, y)
(4)                 Z1 = np.exp(-X**2 - Y**2)
(4)                 Z2 = np.exp(-(X * 10)**2 - (Y * 10)**2)
(4)                 data = Z1 + 50 * Z2
(4)                 c = ax.contourf(data, locator=ticker.LogLocator())
(4)                 assert_array_almost_equal(c.levels, np.power(10.0, np.arange(-6, 3)))
(4)                 cb = fig.colorbar(c, ax=ax)
(4)                 assert_array_almost_equal(cb.ax.get_yticks(), c.levels)
(4)                 _maybe_split_collections(split_collections)
(0)             @pytest.mark.parametrize("split_collections", [False, True])
(0)             @image_comparison(['contour_datetime_axis.png'], style='mpl20')
(0)             def test_contour_datetime_axis(split_collections):
(4)                 fig = plt.figure()
(4)                 fig.subplots_adjust(hspace=0.4, top=0.98, bottom=.15)
(4)                 base = datetime.datetime(2013, 1, 1)
(4)                 x = np.array([base + datetime.timedelta(days=d) for d in range(20)])
(4)                 y = np.arange(20)
(4)                 z1, z2 = np.meshgrid(np.arange(20), np.arange(20))
(4)                 z = z1 * z2
(4)                 plt.subplot(221)
(4)                 plt.contour(x, y, z)
(4)                 plt.subplot(222)
(4)                 plt.contourf(x, y, z)
(4)                 x = np.repeat(x[np.newaxis], 20, axis=0)
(4)                 y = np.repeat(y[:, np.newaxis], 20, axis=1)
(4)                 plt.subplot(223)
(4)                 plt.contour(x, y, z)
(4)                 plt.subplot(224)
(4)                 plt.contourf(x, y, z)
(4)                 for ax in fig.get_axes():
(8)                     for label in ax.get_xticklabels():
(12)                        label.set_ha('right')
(12)                        label.set_rotation(30)
(4)                 _maybe_split_collections(split_collections)
(0)             @pytest.mark.parametrize("split_collections", [False, True])
(0)             @image_comparison(['contour_test_label_transforms.png'],
(18)                              remove_text=True, style='mpl20', tol=1.1)
(0)             def test_labels(split_collections):
(4)                 delta = 0.025
(4)                 x = np.arange(-3.0, 3.0, delta)
(4)                 y = np.arange(-2.0, 2.0, delta)
(4)                 X, Y = np.meshgrid(x, y)
(4)                 Z1 = np.exp(-(X**2 + Y**2) / 2) / (2 * np.pi)
(4)                 Z2 = (np.exp(-(((X - 1) / 1.5)**2 + ((Y - 1) / 0.5)**2) / 2) /
(10)                      (2 * np.pi * 0.5 * 1.5))
(4)                 Z = 10.0 * (Z2 - Z1)
(4)                 fig, ax = plt.subplots(1, 1)
(4)                 CS = ax.contour(X, Y, Z)
(4)                 disp_units = [(216, 177), (359, 290), (521, 406)]
(4)                 data_units = [(-2, .5), (0, -1.5), (2.8, 1)]
(4)                 _maybe_split_collections(split_collections)
(4)                 CS.clabel()
(4)                 for x, y in data_units:
(8)                     CS.add_label_near(x, y, inline=True, transform=None)
(4)                 for x, y in disp_units:
```

```
198: (8)                    CS.add_label_near(x, y, inline=True, transform=False)
199: (4)                _maybe_split_collections(split_collections)
200: (0)            def test_label_contour_start():
201: (4)                _, ax = plt.subplots(dpi=100)
202: (4)                lats = lons = np.linspace(-np.pi / 2, np.pi / 2, 50)
203: (4)                lons, lats = np.meshgrid(lons, lats)
204: (4)                wave = 0.75 * (np.sin(2 * lats) ** 8) * np.cos(4 * lons)
205: (4)                mean = 0.5 * np.cos(2 * lats) * ((np.sin(2 * lats)) ** 2 + 2)
206: (4)                data = wave + mean
207: (4)                cs = ax.contour(lons, lats, data)
208: (4)                with mock.patch.object(
209: (12)                       cs, '_split_path_and_get_label_rotation',
210: (12)                       wraps=cs._split_path_and_get_label_rotation) as mocked_splitter:
211: (8)                    cs.clabel(fontsize=9)
212: (4)                idxs = [cargs[0][1] for cargs in mocked_splitter.call_args_list]
213: (4)                assert 0 in idxs
214: (0)            @pytest.mark.parametrize("split_collections", [False, True])
215: (0)            @image_comparison(['contour_corner_mask_False.png',
'contour_corner_mask_True.png'],
216: (18)                               remove_text=True, tol=1.88)
217: (0)            def test_corner_mask(split_collections):
218: (4)                n = 60
219: (4)                mask_level = 0.95
220: (4)                noise_amp = 1.0
221: (4)                np.random.seed([1])
222: (4)                x, y = np.meshgrid(np.linspace(0, 2.0, n), np.linspace(0, 2.0, n))
223: (4)                z = np.cos(7*x)*np.sin(8*y) + noise_amp*np.random.rand(n, n)
224: (4)                mask = np.random.rand(n, n) >= mask_level
225: (4)                z = np.ma.array(z, mask=mask)
226: (4)                for corner_mask in [False, True]:
227: (8)                    plt.figure()
228: (8)                    plt.contourf(z, corner_mask=corner_mask)
229: (4)                _maybe_split_collections(split_collections)
230: (0)            def test_contourf_decreasing_levels():
231: (4)                z = [[0.1, 0.3], [0.5, 0.7]]
232: (4)                plt.figure()
233: (4)                with pytest.raises(ValueError):
234: (8)                    plt.contourf(z, [1.0, 0.0])
235: (0)            def test_contourf_symmetric_locator():
236: (4)                z = np.arange(12).reshape((3, 4))
237: (4)                locator = plt.MaxNLocator(nbins=4, symmetric=True)
238: (4)                cs = plt.contourf(z, locator=locator)
239: (4)                assert_array_almost_equal(cs.levels, np.linspace(-12, 12, 5))
240: (0)            def test_circular_contour_warning():
241: (4)                x, y = np.meshgrid(np.linspace(-2, 2, 4), np.linspace(-2, 2, 4))
242: (4)                r = np.hypot(x, y)
243: (4)                plt.figure()
244: (4)                cs = plt.contour(x, y, r)
245: (4)                plt.clabel(cs)
246: (0)            @pytest.mark.parametrize("use_clabeltext, contour_zorder, clabel_zorder",
247: (25)                                   [(True, 123, 1234), (False, 123, 1234),
248: (26)                                    (True, 123, None), (False, 123, None)])
249: (0)            def test_clabel_zorder(use_clabeltext, contour_zorder, clabel_zorder):
250: (4)                x, y = np.meshgrid(np.arange(0, 10), np.arange(0, 10))
251: (4)                z = np.max(np.dstack([abs(x), abs(y)]), 2)
252: (4)                fig, (ax1, ax2) = plt.subplots(ncols=2)
253: (4)                cs = ax1.contour(x, y, z, zorder=contour_zorder)
254: (4)                cs_filled = ax2.contourf(x, y, z, zorder=contour_zorder)
255: (4)                clabels1 = cs.clabel(zorder=clabel_zorder, use_clabeltext=use_clabeltext)
256: (4)                clabels2 = cs_filled.clabel(zorder=clabel_zorder,
257: (32)                                           use_clabeltext=use_clabeltext)
258: (4)                if clabel_zorder is None:
259: (8)                    expected_clabel_zorder = 2+contour_zorder
260: (4)                else:
261: (8)                    expected_clabel_zorder = clabel_zorder
262: (4)                for clabel in clabels1:
263: (8)                    assert clabel.get_zorder() == expected_clabel_zorder
264: (4)                for clabel in clabels2:
265: (8)                    assert clabel.get_zorder() == expected_clabel_zorder
```

```
(0)  def test_clabel_with_large_spacing():
(4)      x = y = np.arange(-3.0, 3.01, 0.05)
(4)      X, Y = np.meshgrid(x, y)
(4)      Z = np.exp(-X**2 - Y**2)
(4)      fig, ax = plt.subplots()
(4)      contourset = ax.contour(X, Y, Z, levels=[0.01, 0.2, .5, .8])
(4)      ax.clabel(contourset, inline_spacing=100)
(0)  @pytest.mark.parametrize("split_collections", [False, True])
(0)  @image_comparison(['contour_log_extension.png'],
(18)                   remove_text=True, style='mpl20',
(18)                   tol=1.444)
(0)  def test_contourf_log_extension(split_collections):
(4)      plt.rcParams['pcolormesh.snap'] = False
(4)      fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(10, 5))
(4)      fig.subplots_adjust(left=0.05, right=0.95)
(4)      data_exp = np.linspace(-7.5, 9.5, 1200)
(4)      data = np.power(10, data_exp).reshape(30, 40)
(4)      levels_exp = np.arange(-4., 7.)
(4)      levels = np.power(10., levels_exp)
(4)      c1 = ax1.contourf(data,
(22)                       norm=LogNorm(vmin=data.min(), vmax=data.max()))
(4)      c2 = ax2.contourf(data, levels=levels,
(22)                       norm=LogNorm(vmin=levels.min(), vmax=levels.max()),
(22)                       extend='neither')
(4)      c3 = ax3.contourf(data, levels=levels,
(22)                       norm=LogNorm(vmin=levels.min(), vmax=levels.max()),
(22)                       extend='both')
(4)      cb = plt.colorbar(c1, ax=ax1)
(4)      assert cb.ax.get_ylim() == (1e-8, 1e10)
(4)      cb = plt.colorbar(c2, ax=ax2)
(4)      assert_array_almost_equal_nulp(cb.ax.get_ylim(), np.array((1e-4, 1e6)))
(4)      cb = plt.colorbar(c3, ax=ax3)
(4)      _maybe_split_collections(split_collections)
(0)  @pytest.mark.parametrize("split_collections", [False, True])
(0)  @image_comparison(
(4)      ['contour_addlines.png'], remove_text=True, style='mpl20',
(4)      tol=0.15 if platform.machine() in ('aarch64', 'ppc64le', 's390x')
(8)          else 0.03)
(0)  def test_contour_addlines(split_collections):
(4)      plt.rcParams['pcolormesh.snap'] = False
(4)      fig, ax = plt.subplots()
(4)      np.random.seed(19680812)
(4)      X = np.random.rand(10, 10)*10000
(4)      pcm = ax.pcolormesh(X)
(4)      cont = ax.contour(X+1000)
(4)      cb = fig.colorbar(pcm)
(4)      cb.add_lines(cont)
(4)      assert_array_almost_equal(cb.ax.get_ylim(), [114.3091, 9972.30735], 3)
(4)      _maybe_split_collections(split_collections)
(0)  @pytest.mark.parametrize("split_collections", [False, True])
(0)  @image_comparison(baseline_images=['contour_uneven'],
(18)                   extensions=['png'], remove_text=True, style='mpl20')
(0)  def test_contour_uneven(split_collections):
(4)      plt.rcParams['pcolormesh.snap'] = False
(4)      z = np.arange(24).reshape(4, 6)
(4)      fig, axs = plt.subplots(1, 2)
(4)      ax = axs[0]
(4)      cs = ax.contourf(z, levels=[2, 4, 6, 10, 20])
(4)      fig.colorbar(cs, ax=ax, spacing='proportional')
(4)      ax = axs[1]
(4)      cs = ax.contourf(z, levels=[2, 4, 6, 10, 20])
(4)      fig.colorbar(cs, ax=ax, spacing='uniform')
(4)      _maybe_split_collections(split_collections)
(0)  @pytest.mark.parametrize(
(4)      "rc_lines_linewidth, rc_contour_linewidth, call_linewidths, expected", [
(8)          (1.23, None, None, 1.23),
(8)          (1.23, 4.24, None, 4.24),
(8)          (1.23, 4.24, 5.02, 5.02)
(8)          ])
```

```
335: (0)            def test_contour_linewidth(
336: (8)                    rc_lines_linewidth, rc_contour_linewidth, call_linewidths, expected):
337: (4)                with rc_context(rc={"lines.linewidth": rc_lines_linewidth,
338: (24)                                   "contour.linewidth": rc_contour_linewidth}):
339: (8)                    fig, ax = plt.subplots()
340: (8)                    X = np.arange(4*3).reshape(4, 3)
341: (8)                    cs = ax.contour(X, linewidths=call_linewidths)
342: (8)                    assert cs.get_linewidths()[0] == expected
343: (8)                    with pytest.warns(mpl.MatplotlibDeprecationWarning,
match="tlinewidths"):
344: (12)                       assert cs.tlinewidths[0][0] == expected
345: (0)            @pytest.mark.backend("pdf")
346: (0)            def test_label_nonagg():
347: (4)                plt.clabel(plt.contour([[1, 2], [3, 4]]))
348: (0)            @pytest.mark.parametrize("split_collections", [False, True])
349: (0)            @image_comparison(baseline_images=['contour_closed_line_loop'],
350: (18)                              extensions=['png'], remove_text=True)
351: (0)            def test_contour_closed_line_loop(split_collections):
352: (4)                z = [[0, 0, 0], [0, 2, 0], [0, 0, 0], [2, 1, 2]]
353: (4)                fig, ax = plt.subplots(figsize=(2, 2))
354: (4)                ax.contour(z, [0.5], linewidths=[20], alpha=0.7)
355: (4)                ax.set_xlim(-0.1, 2.1)
356: (4)                ax.set_ylim(-0.1, 3.1)
357: (4)                _maybe_split_collections(split_collections)
358: (0)            def test_quadcontourset_reuse():
359: (4)                x, y = np.meshgrid([0.0, 1.0], [0.0, 1.0])
360: (4)                z = x + y
361: (4)                fig, ax = plt.subplots()
362: (4)                qcs1 = ax.contourf(x, y, z)
363: (4)                qcs2 = ax.contour(x, y, z)
364: (4)                assert qcs2._contour_generator != qcs1._contour_generator
365: (4)                qcs3 = ax.contour(qcs1, z)
366: (4)                assert qcs3._contour_generator == qcs1._contour_generator
367: (0)            @pytest.mark.parametrize("split_collections", [False, True])
368: (0)            @image_comparison(baseline_images=['contour_manual'],
369: (18)                              extensions=['png'], remove_text=True, tol=0.89)
370: (0)            def test_contour_manual(split_collections):
371: (4)                from matplotlib.contour import ContourSet
372: (4)                fig, ax = plt.subplots(figsize=(4, 4))
373: (4)                cmap = 'viridis'
374: (4)                lines0 = [[[2, 0], [1, 2], [1, 3]]]  # Single line.
375: (4)                lines1 = [[[3, 0], [3, 2]], [[3, 3], [3, 4]]]  # Two lines.
376: (4)                filled01 = [[[0, 0], [0, 4], [1, 3], [1, 2], [2, 0]]]
377: (4)                filled12 = [[[2, 0], [3, 0], [3, 2], [1, 3], [1, 2]],  # Two polygons.
378: (16)                           [[1, 4], [3, 4], [3, 3]]]
379: (4)                ContourSet(ax, [0, 1, 2], [filled01, filled12], filled=True, cmap=cmap)
380: (4)                ContourSet(ax, [1, 2], [lines0, lines1], linewidths=3, colors=['r', 'k'])
381: (4)                segs = [[[4, 0], [7, 0], [7, 3], [4, 3], [4, 0],
382: (13)                        [5, 1], [5, 2], [6, 2], [6, 1], [5, 1]]]
383: (4)                kinds = [[1, 2, 2, 2, 79, 1, 2, 2, 2, 79]]  # Polygon containing hole.
384: (4)                ContourSet(ax, [2, 3], [segs], [kinds], filled=True, cmap=cmap)
385: (4)                ContourSet(ax, [2], [segs], [kinds], colors='k', linewidths=3)
386: (4)                _maybe_split_collections(split_collections)
387: (0)            @pytest.mark.parametrize("split_collections", [False, True])
388: (0)            @image_comparison(baseline_images=['contour_line_start_on_corner_edge'],
389: (18)                              extensions=['png'], remove_text=True)
390: (0)            def test_contour_line_start_on_corner_edge(split_collections):
391: (4)                fig, ax = plt.subplots(figsize=(6, 5))
392: (4)                x, y = np.meshgrid([0, 1, 2, 3, 4], [0, 1, 2])
393: (4)                z = 1.2 - (x - 2)**2 + (y - 1)**2
394: (4)                mask = np.zeros_like(z, dtype=bool)
395: (4)                mask[1, 1] = mask[1, 3] = True
396: (4)                z = np.ma.array(z, mask=mask)
397: (4)                filled = ax.contourf(x, y, z, corner_mask=True)
398: (4)                cbar = fig.colorbar(filled)
399: (4)                lines = ax.contour(x, y, z, corner_mask=True, colors='k')
400: (4)                cbar.add_lines(lines)
401: (4)                _maybe_split_collections(split_collections)
402: (0)            def test_find_nearest_contour():
```

```
403: (4)                xy = np.indices((15, 15))
404: (4)                img = np.exp(-np.pi * (np.sum((xy - 5)**2, 0)/5.**2))
405: (4)                cs = plt.contour(img, 10)
406: (4)                nearest_contour = cs.find_nearest_contour(1, 1, pixel=False)
407: (4)                expected_nearest = (1, 0, 33, 1.965966, 1.965966, 1.866183)
408: (4)                assert_array_almost_equal(nearest_contour, expected_nearest)
409: (4)                nearest_contour = cs.find_nearest_contour(8, 1, pixel=False)
410: (4)                expected_nearest = (1, 0, 5, 7.550173, 1.587542, 0.547550)
411: (4)                assert_array_almost_equal(nearest_contour, expected_nearest)
412: (4)                nearest_contour = cs.find_nearest_contour(2, 5, pixel=False)
413: (4)                expected_nearest = (3, 0, 21, 1.884384, 5.023335, 0.013911)
414: (4)                assert_array_almost_equal(nearest_contour, expected_nearest)
415: (4)                nearest_contour = cs.find_nearest_contour(2, 5, indices=(5, 7),
pixel=False)
416: (4)                expected_nearest = (5, 0, 16, 2.628202, 5.0, 0.394638)
417: (4)                assert_array_almost_equal(nearest_contour, expected_nearest)
418: (0)            def test_find_nearest_contour_no_filled():
419: (4)                xy = np.indices((15, 15))
420: (4)                img = np.exp(-np.pi * (np.sum((xy - 5)**2, 0)/5.**2))
421: (4)                cs = plt.contourf(img, 10)
422: (4)                with pytest.raises(ValueError, match="Method does not support filled
contours"):
423: (8)                    cs.find_nearest_contour(1, 1, pixel=False)
424: (4)                with pytest.raises(ValueError, match="Method does not support filled
contours"):
425: (8)                    cs.find_nearest_contour(1, 10, indices=(5, 7), pixel=False)
426: (4)                with pytest.raises(ValueError, match="Method does not support filled
contours"):
427: (8)                    cs.find_nearest_contour(2, 5, indices=(2, 7), pixel=True)
428: (0)            @mpl.style.context("default")
429: (0)            def test_contour_autolabel_beyond_powerlimits():
430: (4)                ax = plt.figure().add_subplot()
431: (4)                cs = plt.contour(np.geomspace(1e-6, 1e-4, 100).reshape(10, 10),
432: (21)                                 levels=[.25e-5, 1e-5, 4e-5])
433: (4)                ax.clabel(cs)
434: (4)                assert {text.get_text() for text in ax.texts} == {"0.25", "1.00", "4.00"}
435: (0)            def test_contourf_legend_elements():
436: (4)                from matplotlib.patches import Rectangle
437: (4)                x = np.arange(1, 10)
438: (4)                y = x.reshape(-1, 1)
439: (4)                h = x * y
440: (4)                cs = plt.contourf(h, levels=[10, 30, 50],
441: (22)                                  colors=['#FFFF00', '#FF00FF', '#00FFFF'],
442: (22)                                  extend='both')
443: (4)                cs.cmap.set_over('red')
444: (4)                cs.cmap.set_under('blue')
445: (4)                cs.changed()
446: (4)                artists, labels = cs.legend_elements()
447: (4)                assert labels == ['$x \\leq -1e+250s$',
448: (22)                                  '$10.0 < x \\leq 30.0$',
449: (22)                                  '$30.0 < x \\leq 50.0$',
450: (22)                                  '$x > 1e+250s$']
451: (4)                expected_colors = ('blue', '#FFFF00', '#FF00FF', 'red')
452: (4)                assert all(isinstance(a, Rectangle) for a in artists)
453: (4)                assert all(same_color(a.get_facecolor(), c)
454: (15)                           for a, c in zip(artists, expected_colors))
455: (0)            def test_contour_legend_elements():
456: (4)                x = np.arange(1, 10)
457: (4)                y = x.reshape(-1, 1)
458: (4)                h = x * y
459: (4)                colors = ['blue', '#00FF00', 'red']
460: (4)                cs = plt.contour(h, levels=[10, 30, 50],
461: (21)                                 colors=colors,
462: (21)                                 extend='both')
463: (4)                artists, labels = cs.legend_elements()
464: (4)                assert labels == ['$x = 10.0$', '$x = 30.0$', '$x = 50.0$']
465: (4)                assert all(isinstance(a, mpl.lines.Line2D) for a in artists)
466: (4)                assert all(same_color(a.get_color(), c)
467: (15)                           for a, c in zip(artists, colors))
```

```
(0)             @pytest.mark.parametrize(
(4)                 "algorithm, klass",
(4)                 [('mpl2005', contourpy.Mpl2005ContourGenerator),
(5)                  ('mpl2014', contourpy.Mpl2014ContourGenerator),
(5)                  ('serial', contourpy.SerialContourGenerator),
(5)                  ('threaded', contourpy.ThreadedContourGenerator),
(5)                  ('invalid', None)])
(0)             def test_algorithm_name(algorithm, klass):
(4)                 z = np.array([[1.0, 2.0], [3.0, 4.0]])
(4)                 if klass is not None:
(8)                     cs = plt.contourf(z, algorithm=algorithm)
(8)                     assert isinstance(cs._contour_generator, klass)
(4)                 else:
(8)                     with pytest.raises(ValueError):
(12)                        plt.contourf(z, algorithm=algorithm)
(0)             @pytest.mark.parametrize(
(4)                 "algorithm", ['mpl2005', 'mpl2014', 'serial', 'threaded'])
(0)             def test_algorithm_supports_corner_mask(algorithm):
(4)                 z = np.array([[1.0, 2.0], [3.0, 4.0]])
(4)                 plt.contourf(z, algorithm=algorithm, corner_mask=False)
(4)                 if algorithm != 'mpl2005':
(8)                     plt.contourf(z, algorithm=algorithm, corner_mask=True)
(4)                 else:
(8)                     with pytest.raises(ValueError):
(12)                        plt.contourf(z, algorithm=algorithm, corner_mask=True)
(0)             @pytest.mark.parametrize("split_collections", [False, True])
(0)             @image_comparison(baseline_images=['contour_all_algorithms'],
(18)                              extensions=['png'], remove_text=True, tol=0.06)
(0)             def test_all_algorithms(split_collections):
(4)                 algorithms = ['mpl2005', 'mpl2014', 'serial', 'threaded']
(4)                 rng = np.random.default_rng(2981)
(4)                 x, y = np.meshgrid(np.linspace(0.0, 1.0, 10), np.linspace(0.0, 1.0, 6))
(4)                 z = np.sin(15*x)*np.cos(10*y) + rng.normal(scale=0.5, size=(6, 10))
(4)                 mask = np.zeros_like(z, dtype=bool)
(4)                 mask[3, 7] = True
(4)                 z = np.ma.array(z, mask=mask)
(4)                 _, axs = plt.subplots(2, 2)
(4)                 for ax, algorithm in zip(axs.ravel(), algorithms):
(8)                     ax.contourf(x, y, z, algorithm=algorithm)
(8)                     ax.contour(x, y, z, algorithm=algorithm, colors='k')
(8)                     ax.set_title(algorithm)
(4)                 _maybe_split_collections(split_collections)
(0)             def test_subfigure_clabel():
(4)                 delta = 0.025
(4)                 x = np.arange(-3.0, 3.0, delta)
(4)                 y = np.arange(-2.0, 2.0, delta)
(4)                 X, Y = np.meshgrid(x, y)
(4)                 Z1 = np.exp(-(X**2) - Y**2)
(4)                 Z2 = np.exp(-((X - 1) ** 2) - (Y - 1) ** 2)
(4)                 Z = (Z1 - Z2) * 2
(4)                 fig = plt.figure()
(4)                 figs = fig.subfigures(nrows=1, ncols=2)
(4)                 for f in figs:
(8)                     ax = f.subplots()
(8)                     CS = ax.contour(X, Y, Z)
(8)                     ax.clabel(CS, inline=True, fontsize=10)
(8)                     ax.set_title("Simplest default with labels")
(0)             @pytest.mark.parametrize(
(4)                 "style", ['solid', 'dashed', 'dashdot', 'dotted'])
(0)             def test_linestyles(style):
(4)                 delta = 0.025
(4)                 x = np.arange(-3.0, 3.0, delta)
(4)                 y = np.arange(-2.0, 2.0, delta)
(4)                 X, Y = np.meshgrid(x, y)
(4)                 Z1 = np.exp(-X**2 - Y**2)
(4)                 Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
(4)                 Z = (Z1 - Z2) * 2
(4)                 fig1, ax1 = plt.subplots()
(4)                 CS1 = ax1.contour(X, Y, Z, 6, colors='k')
```

```
(4)                 ax1.clabel(CS1, fontsize=9, inline=True)
(4)                 ax1.set_title('Single color - positive contours solid (default)')
(4)                 assert CS1.linestyles is None  # default
(4)                 fig2, ax2 = plt.subplots()
(4)                 CS2 = ax2.contour(X, Y, Z, 6, colors='k', linestyles=style)
(4)                 ax2.clabel(CS2, fontsize=9, inline=True)
(4)                 ax2.set_title(f'Single color - positive contours {style}')
(4)                 assert CS2.linestyles == style
(4)                 fig3, ax3 = plt.subplots()
(4)                 CS3 = ax3.contour(X, Y, Z, 6, colors='k', linestyles=style,
(22)                                  negative_linestyles='dashdot')
(4)                 ax3.clabel(CS3, fontsize=9, inline=True)
(4)                 ax3.set_title(f'Single color - positive contours {style}')
(4)                 assert CS3.linestyles == style
(0)             @pytest.mark.parametrize(
(4)                 "style", ['solid', 'dashed', 'dashdot', 'dotted'])
(0)             def test_negative_linestyles(style):
(4)                 delta = 0.025
(4)                 x = np.arange(-3.0, 3.0, delta)
(4)                 y = np.arange(-2.0, 2.0, delta)
(4)                 X, Y = np.meshgrid(x, y)
(4)                 Z1 = np.exp(-X**2 - Y**2)
(4)                 Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
(4)                 Z = (Z1 - Z2) * 2
(4)                 fig1, ax1 = plt.subplots()
(4)                 CS1 = ax1.contour(X, Y, Z, 6, colors='k')
(4)                 ax1.clabel(CS1, fontsize=9, inline=True)
(4)                 ax1.set_title('Single color - negative contours dashed (default)')
(4)                 assert CS1.negative_linestyles == 'dashed'  # default
(4)                 plt.rcParams['contour.negative_linestyle'] = style
(4)                 fig2, ax2 = plt.subplots()
(4)                 CS2 = ax2.contour(X, Y, Z, 6, colors='k')
(4)                 ax2.clabel(CS2, fontsize=9, inline=True)
(4)                 ax2.set_title(f'Single color - negative contours {style}'
(19)                              '(using rcParams)')
(4)                 assert CS2.negative_linestyles == style
(4)                 fig3, ax3 = plt.subplots()
(4)                 CS3 = ax3.contour(X, Y, Z, 6, colors='k', negative_linestyles=style)
(4)                 ax3.clabel(CS3, fontsize=9, inline=True)
(4)                 ax3.set_title(f'Single color - negative contours {style}')
(4)                 assert CS3.negative_linestyles == style
(4)                 fig4, ax4 = plt.subplots()
(4)                 CS4 = ax4.contour(X, Y, Z, 6, colors='k', linestyles='dashdot',
(22)                                  negative_linestyles=style)
(4)                 ax4.clabel(CS4, fontsize=9, inline=True)
(4)                 ax4.set_title(f'Single color - negative contours {style}')
(4)                 assert CS4.negative_linestyles == style
(0)             def test_contour_remove():
(4)                 ax = plt.figure().add_subplot()
(4)                 orig_children = ax.get_children()
(4)                 cs = ax.contour(np.arange(16).reshape((4, 4)))
(4)                 cs.clabel()
(4)                 assert ax.get_children() != orig_children
(4)                 cs.remove()
(4)                 assert ax.get_children() == orig_children
(0)             def test_contour_no_args():
(4)                 fig, ax = plt.subplots()
(4)                 data = [[0, 1], [1, 0]]
(4)                 with pytest.raises(TypeError, match=r"contour\(\) takes from 1 to 4"):
(8)                     ax.contour(Z=data)
(0)             def test_contour_clip_path():
(4)                 fig, ax = plt.subplots()
(4)                 data = [[0, 1], [1, 0]]
(4)                 circle = mpatches.Circle([0.5, 0.5], 0.5, transform=ax.transAxes)
(4)                 cs = ax.contour(data, clip_path=circle)
(4)                 assert cs.get_clip_path() is not None
(0)             def test_bool_autolevel():
(4)                 x, y = np.random.rand(2, 9)
(4)                 z = (np.arange(9) % 2).reshape((3, 3)).astype(bool)
```

```
606: (4)                m = [[False, False, False], [False, True, False], [False, False, False]]
607: (4)                assert plt.contour(z.tolist()).levels.tolist() == [.5]
608: (4)                assert plt.contour(z).levels.tolist() == [.5]
609: (4)                assert plt.contour(np.ma.array(z, mask=m)).levels.tolist() == [.5]
610: (4)                assert plt.contourf(z.tolist()).levels.tolist() == [0, .5, 1]
611: (4)                assert plt.contourf(z).levels.tolist() == [0, .5, 1]
612: (4)                assert plt.contourf(np.ma.array(z, mask=m)).levels.tolist() == [0, .5, 1]
613: (4)                z = z.ravel()
614: (4)                assert plt.tricontour(x, y, z.tolist()).levels.tolist() == [.5]
615: (4)                assert plt.tricontour(x, y, z).levels.tolist() == [.5]
616: (4)                assert plt.tricontourf(x, y, z.tolist()).levels.tolist() == [0, .5, 1]
617: (4)                assert plt.tricontourf(x, y, z).levels.tolist() == [0, .5, 1]
618: (0)            def test_all_nan():
619: (4)                x = np.array([[np.nan, np.nan], [np.nan, np.nan]])
620: (4)                assert_array_almost_equal(plt.contour(x).levels,
621: (30)                                          [-1e-13, -7.5e-14, -5e-14, -2.4e-14, 0.0,
622: (32)                                            2.4e-14, 5e-14, 7.5e-14, 1e-13])
623: (0)            def test_allsegs_allkinds():
624: (4)                x, y = np.meshgrid(np.arange(0, 10, 2), np.arange(0, 10, 2))
625: (4)                z = np.sin(x) * np.cos(y)
626: (4)                cs = plt.contour(x, y, z, levels=[0, 0.5])
627: (4)                for result in [cs.allsegs, cs.allkinds]:
628: (8)                    assert len(result) == 2
629: (8)                    assert len(result[0]) == 5
630: (8)                    assert len(result[1]) == 4
631: (0)            def test_deprecated_apis():
632: (4)                cs = plt.contour(np.arange(16).reshape((4, 4)))
633: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning, match="collections"):
634: (8)                    colls = cs.collections
635: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning, match="tcolors"):
636: (8)                    assert_array_equal(cs.tcolors, [c.get_edgecolor() for c in colls])
637: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning, match="tlinewidths"):
638: (8)                    assert cs.tlinewidths == [c.get_linewidth() for c in colls]
639: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning, match="antialiased"):
640: (8)                    assert cs.antialiased
641: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning, match="antialiased"):
642: (8)                    cs.antialiased = False
643: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning, match="antialiased"):
644: (8)                    assert not cs.antialiased
```

File 178 - test_cycles.py:

```
1: (0)              import contextlib
2: (0)              from io import StringIO
3: (0)              import matplotlib as mpl
4: (0)              import matplotlib.pyplot as plt
5: (0)              import numpy as np
6: (0)              import pytest
7: (0)              from cycler import cycler
8: (0)              def test_colorcycle_basic():
9: (4)                  fig, ax = plt.subplots()
10: (4)                 ax.set_prop_cycle(cycler('color', ['r', 'g', 'y']))
11: (4)                 for _ in range(4):
12: (8)                     ax.plot(range(10), range(10))
13: (4)                 assert [l.get_color() for l in ax.lines] == ['r', 'g', 'y', 'r']
14: (0)             def test_marker_cycle():
15: (4)                 fig, ax = plt.subplots()
16: (4)                 ax.set_prop_cycle(cycler('c', ['r', 'g', 'y']) +
17: (22)                                  cycler('marker', ['.', '*', 'x']))
18: (4)                 for _ in range(4):
19: (8)                     ax.plot(range(10), range(10))
20: (4)                 assert [l.get_color() for l in ax.lines] == ['r', 'g', 'y', 'r']
21: (4)                 assert [l.get_marker() for l in ax.lines] == ['.', '*', 'x', '.']
22: (0)             def test_valid_marker_cycles():
23: (4)                 fig, ax = plt.subplots()
24: (4)                 ax.set_prop_cycle(cycler(marker=[1, "+", ".", 4]))
25: (0)             def test_marker_cycle_kwargs_arrays_iterators():
```

```
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_prop_cycle(c=np.array(['r', 'g', 'y']),
(22)                                  marker=iter(['.', '*', 'x']))
(4)                 for _ in range(4):
(8)                     ax.plot(range(10), range(10))
(4)                 assert [l.get_color() for l in ax.lines] == ['r', 'g', 'y', 'r']
(4)                 assert [l.get_marker() for l in ax.lines] == ['.', '*', 'x', '.']
(0)             def test_linestylecycle_basic():
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_prop_cycle(cycler('ls', ['-', '--', ':']))
(4)                 for _ in range(4):
(8)                     ax.plot(range(10), range(10))
(4)                 assert [l.get_linestyle() for l in ax.lines] == ['-', '--', ':', '-']
(0)             def test_fillcycle_basic():
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_prop_cycle(cycler('c',  ['r', 'g', 'y']) +
(22)                                  cycler('hatch', ['xx', 'O', '|-']) +
(22)                                  cycler('linestyle', ['-', '--', ':']))
(4)                 for _ in range(4):
(8)                     ax.fill(range(10), range(10))
(4)                 assert ([p.get_facecolor() for p in ax.patches]
(12)                        == [mpl.colors.to_rgba(c) for c in ['r', 'g', 'y', 'r']])
(4)                 assert [p.get_hatch() for p in ax.patches] == ['xx', 'O', '|-', 'xx']
(4)                 assert [p.get_linestyle() for p in ax.patches] == ['-', '--', ':', '-']
(0)             def test_fillcycle_ignore():
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_prop_cycle(cycler('color',  ['r', 'g', 'y']) +
(22)                                  cycler('hatch', ['xx', 'O', '|-']) +
(22)                                  cycler('marker', ['.', '*', 'D']))
(4)                 t = range(10)
(4)                 ax.fill(t, t, 'r', hatch='xx')
(4)                 ax.fill(t, t, hatch='O')
(4)                 ax.fill(t, t)
(4)                 ax.fill(t, t)
(4)                 assert ([p.get_facecolor() for p in ax.patches]
(12)                        == [mpl.colors.to_rgba(c) for c in ['r', 'r', 'g', 'y']])
(4)                 assert [p.get_hatch() for p in ax.patches] == ['xx', 'O', 'O', '|-']
(0)             def test_property_collision_plot():
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_prop_cycle('linewidth', [2, 4])
(4)                 t = range(10)
(4)                 for c in range(1, 4):
(8)                     ax.plot(t, t, lw=0.1)
(4)                 ax.plot(t, t)
(4)                 ax.plot(t, t)
(4)                 assert [l.get_linewidth() for l in ax.lines] == [0.1, 0.1, 0.1, 2, 4]
(0)             def test_property_collision_fill():
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_prop_cycle(linewidth=[2, 3, 4, 5, 6], facecolor='bgcmy')
(4)                 t = range(10)
(4)                 for c in range(1, 4):
(8)                     ax.fill(t, t, lw=0.1)
(4)                 ax.fill(t, t)
(4)                 ax.fill(t, t)
(4)                 assert ([p.get_facecolor() for p in ax.patches]
(12)                        == [mpl.colors.to_rgba(c) for c in 'bgcmy'])
(4)                 assert [p.get_linewidth() for p in ax.patches] == [0.1, 0.1, 0.1, 5, 6]
(0)             def test_valid_input_forms():
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_prop_cycle(None)
(4)                 ax.set_prop_cycle(cycler('linewidth', [1, 2]))
(4)                 ax.set_prop_cycle('color', 'rgywkbcm')
(4)                 ax.set_prop_cycle('lw', (1, 2))
(4)                 ax.set_prop_cycle('linewidth', [1, 2])
(4)                 ax.set_prop_cycle('linewidth', iter([1, 2]))
(4)                 ax.set_prop_cycle('linewidth', np.array([1, 2]))
(4)                 ax.set_prop_cycle('color', np.array([[1, 0, 0],
(41)                                                     [0, 1, 0],
(41)                                                     [0, 0, 1]]))
```

```
95: (4)                 ax.set_prop_cycle('dashes', [[], [13, 2], [8, 3, 1, 3]])
96: (4)                 ax.set_prop_cycle(lw=[1, 2], color=['k', 'w'], ls=['-', '--'])
97: (4)                 ax.set_prop_cycle(lw=np.array([1, 2]),
98: (22)                                  color=np.array(['k', 'w']),
99: (22)                                  ls=np.array(['-', '--']))
100: (0)            def test_cycle_reset():
101: (4)                fig, ax = plt.subplots()
102: (4)                prop0 = StringIO()
103: (4)                prop1 = StringIO()
104: (4)                prop2 = StringIO()
105: (4)                with contextlib.redirect_stdout(prop0):
106: (8)                    plt.getp(ax.plot([1, 2], label="label")[0])
107: (4)                ax.set_prop_cycle(linewidth=[10, 9, 4])
108: (4)                with contextlib.redirect_stdout(prop1):
109: (8)                    plt.getp(ax.plot([1, 2], label="label")[0])
110: (4)                assert prop1.getvalue() != prop0.getvalue()
111: (4)                ax.set_prop_cycle(None)
112: (4)                with contextlib.redirect_stdout(prop2):
113: (8)                    plt.getp(ax.plot([1, 2], label="label")[0])
114: (4)                assert prop2.getvalue() == prop0.getvalue()
115: (0)            def test_invalid_input_forms():
116: (4)                fig, ax = plt.subplots()
117: (4)                with pytest.raises((TypeError, ValueError)):
118: (8)                    ax.set_prop_cycle(1)
119: (4)                with pytest.raises((TypeError, ValueError)):
120: (8)                    ax.set_prop_cycle([1, 2])
121: (4)                with pytest.raises((TypeError, ValueError)):
122: (8)                    ax.set_prop_cycle('color', 'fish')
123: (4)                with pytest.raises((TypeError, ValueError)):
124: (8)                    ax.set_prop_cycle('linewidth', 1)
125: (4)                with pytest.raises((TypeError, ValueError)):
126: (8)                    ax.set_prop_cycle('linewidth', {1, 2})
127: (4)                with pytest.raises((TypeError, ValueError)):
128: (8)                    ax.set_prop_cycle(linewidth=1, color='r')
129: (4)                with pytest.raises((TypeError, ValueError)):
130: (8)                    ax.set_prop_cycle('foobar', [1, 2])
131: (4)                with pytest.raises((TypeError, ValueError)):
132: (8)                    ax.set_prop_cycle(foobar=[1, 2])
133: (4)                with pytest.raises((TypeError, ValueError)):
134: (8)                    ax.set_prop_cycle(cycler(foobar=[1, 2]))
135: (4)                with pytest.raises(ValueError):
136: (8)                    ax.set_prop_cycle(cycler(color='rgb', c='cmy'))
```

File 179 - test_dates.py:

```
1: (0)              import datetime
2: (0)              import dateutil.tz
3: (0)              import dateutil.rrule
4: (0)              import functools
5: (0)              import numpy as np
6: (0)              import pytest
7: (0)              from matplotlib import rc_context, style
8: (0)              import matplotlib.dates as mdates
9: (0)              import matplotlib.pyplot as plt
10: (0)             from matplotlib.testing.decorators import image_comparison
11: (0)             import matplotlib.ticker as mticker
12: (0)             def test_date_numpyx():
13: (4)                 base = datetime.datetime(2017, 1, 1)
14: (4)                 time = [base + datetime.timedelta(days=x) for x in range(0, 3)]
15: (4)                 timenp = np.array(time, dtype='datetime64[ns]')
16: (4)                 data = np.array([0., 2., 1.])
17: (4)                 fig = plt.figure(figsize=(10, 2))
18: (4)                 ax = fig.add_subplot(1, 1, 1)
19: (4)                 h, = ax.plot(time, data)
20: (4)                 hnp, = ax.plot(timenp, data)
21: (4)                 np.testing.assert_equal(h.get_xdata(orig=False),
hnp.get_xdata(orig=False))
```

```
 22: (4)                 fig = plt.figure(figsize=(10, 2))
 23: (4)                 ax = fig.add_subplot(1, 1, 1)
 24: (4)                 h, = ax.plot(data, time)
 25: (4)                 hnp, = ax.plot(data, timenp)
 26: (4)                 np.testing.assert_equal(h.get_ydata(orig=False),
hnp.get_ydata(orig=False))
 27: (0)             @pytest.mark.parametrize('t0', [datetime.datetime(2017, 1, 1, 0, 1, 1),
 28: (32)                                            [datetime.datetime(2017, 1, 1, 0, 1, 1),
 29: (33)                                             datetime.datetime(2017, 1, 1, 1, 1, 1)],
 30: (32)                                            [[datetime.datetime(2017, 1, 1, 0, 1, 1),
 31: (34)                                              datetime.datetime(2017, 1, 1, 1, 1, 1)],
 32: (33)                                             [datetime.datetime(2017, 1, 1, 2, 1, 1),
 33: (34)                                              datetime.datetime(2017, 1, 1, 3, 1, 1)]]])
 34: (0)             @pytest.mark.parametrize('dtype', ['datetime64[s]',
 35: (35)                                               'datetime64[us]',
 36: (35)                                               'datetime64[ms]',
 37: (35)                                               'datetime64[ns]'])
 38: (0)             def test_date_date2num_numpy(t0, dtype):
 39: (4)                 time = mdates.date2num(t0)
 40: (4)                 tnp = np.array(t0, dtype=dtype)
 41: (4)                 nptime = mdates.date2num(tnp)
 42: (4)                 np.testing.assert_equal(time, nptime)
 43: (0)             @pytest.mark.parametrize('dtype', ['datetime64[s]',
 44: (35)                                               'datetime64[us]',
 45: (35)                                               'datetime64[ms]',
 46: (35)                                               'datetime64[ns]'])
 47: (0)             def test_date2num_NaT(dtype):
 48: (4)                 t0 = datetime.datetime(2017, 1, 1, 0, 1, 1)
 49: (4)                 tmpl = [mdates.date2num(t0), np.nan]
 50: (4)                 tnp = np.array([t0, 'NaT'], dtype=dtype)
 51: (4)                 nptime = mdates.date2num(tnp)
 52: (4)                 np.testing.assert_array_equal(tmpl, nptime)
 53: (0)             @pytest.mark.parametrize('units', ['s', 'ms', 'us', 'ns'])
 54: (0)             def test_date2num_NaT_scalar(units):
 55: (4)                 tmpl = mdates.date2num(np.datetime64('NaT', units))
 56: (4)                 assert np.isnan(tmpl)
 57: (0)             def test_date2num_masked():
 58: (4)                 base = datetime.datetime(2022, 12, 15)
 59: (4)                 dates = np.ma.array([base + datetime.timedelta(days=(2 * i))
 60: (25)                                     for i in range(7)], mask=[0, 1, 1, 0, 0, 0, 1])
 61: (4)                 npdates = mdates.date2num(dates)
 62: (4)                 np.testing.assert_array_equal(np.ma.getmask(npdates),
 63: (34)                                              (False, True, True, False, False, False,
 64: (35)                                               True))
 65: (4)                 base = datetime.datetime(2022, 12, 15, tzinfo=mdates.UTC)
 66: (4)                 dates = np.ma.array([base + datetime.timedelta(days=(2 * i))
 67: (25)                                     for i in range(7)], mask=[0, 1, 1, 0, 0, 0, 1])
 68: (4)                 npdates = mdates.date2num(dates)
 69: (4)                 np.testing.assert_array_equal(np.ma.getmask(npdates),
 70: (34)                                              (False, True, True, False, False, False,
 71: (35)                                               True))
 72: (0)             def test_date_empty():
 73: (4)                 fig, ax = plt.subplots()
 74: (4)                 ax.xaxis_date()
 75: (4)                 fig.draw_without_rendering()
 76: (4)                 np.testing.assert_allclose(ax.get_xlim(),
 77: (31)                                           [mdates.date2num(np.datetime64('1970-01-01')),
 78: (32)                                            mdates.date2num(np.datetime64('1970-01-02'))])
 79: (4)                 mdates._reset_epoch_test_example()
 80: (4)                 mdates.set_epoch('0000-12-31')
 81: (4)                 fig, ax = plt.subplots()
 82: (4)                 ax.xaxis_date()
 83: (4)                 fig.draw_without_rendering()
 84: (4)                 np.testing.assert_allclose(ax.get_xlim(),
 85: (31)                                           [mdates.date2num(np.datetime64('1970-01-01')),
 86: (32)                                            mdates.date2num(np.datetime64('1970-01-02'))])
 87: (4)                 mdates._reset_epoch_test_example()
 88: (0)             def test_date_not_empty():
 89: (4)                 fig = plt.figure()
```

```
 90: (4)                 ax = fig.add_subplot()
 91: (4)                 ax.plot([50, 70], [1, 2])
 92: (4)                 ax.xaxis.axis_date()
 93: (4)                 np.testing.assert_allclose(ax.get_xlim(), [50, 70])
 94: (0)             def test_axhline():
 95: (4)                 fig, ax = plt.subplots()
 96: (4)                 ax.axhline(1.5)
 97: (4)                 ax.plot([np.datetime64('2016-01-01'), np.datetime64('2016-01-02')], [1,
 2])
 98: (4)                 np.testing.assert_allclose(ax.get_xlim(),
 99: (31)                                           [mdates.date2num(np.datetime64('2016-01-01')),
 100: (32)                                           mdates.date2num(np.datetime64('2016-01-02'))])
 101: (4)                mdates._reset_epoch_test_example()
 102: (4)                mdates.set_epoch('0000-12-31')
 103: (4)                fig, ax = plt.subplots()
 104: (4)                ax.axhline(1.5)
 105: (4)                ax.plot([np.datetime64('2016-01-01'), np.datetime64('2016-01-02')], [1,
 2])
 106: (4)                np.testing.assert_allclose(ax.get_xlim(),
 107: (31)                                          [mdates.date2num(np.datetime64('2016-01-01')),
 108: (32)                                           mdates.date2num(np.datetime64('2016-01-02'))])
 109: (4)                mdates._reset_epoch_test_example()
 110: (0)            @image_comparison(['date_axhspan.png'])
 111: (0)            def test_date_axhspan():
 112: (4)                t0 = datetime.datetime(2009, 1, 20)
 113: (4)                tf = datetime.datetime(2009, 1, 21)
 114: (4)                fig, ax = plt.subplots()
 115: (4)                ax.axhspan(t0, tf, facecolor="blue", alpha=0.25)
 116: (4)                ax.set_ylim(t0 - datetime.timedelta(days=5),
 117: (16)                           tf + datetime.timedelta(days=5))
 118: (4)                fig.subplots_adjust(left=0.25)
 119: (0)            @image_comparison(['date_axvspan.png'])
 120: (0)            def test_date_axvspan():
 121: (4)                t0 = datetime.datetime(2000, 1, 20)
 122: (4)                tf = datetime.datetime(2010, 1, 21)
 123: (4)                fig, ax = plt.subplots()
 124: (4)                ax.axvspan(t0, tf, facecolor="blue", alpha=0.25)
 125: (4)                ax.set_xlim(t0 - datetime.timedelta(days=720),
 126: (16)                           tf + datetime.timedelta(days=720))
 127: (4)                fig.autofmt_xdate()
 128: (0)            @image_comparison(['date_axhline.png'])
 129: (0)            def test_date_axhline():
 130: (4)                t0 = datetime.datetime(2009, 1, 20)
 131: (4)                tf = datetime.datetime(2009, 1, 31)
 132: (4)                fig, ax = plt.subplots()
 133: (4)                ax.axhline(t0, color="blue", lw=3)
 134: (4)                ax.set_ylim(t0 - datetime.timedelta(days=5),
 135: (16)                           tf + datetime.timedelta(days=5))
 136: (4)                fig.subplots_adjust(left=0.25)
 137: (0)            @image_comparison(['date_axvline.png'])
 138: (0)            def test_date_axvline():
 139: (4)                t0 = datetime.datetime(2000, 1, 20)
 140: (4)                tf = datetime.datetime(2000, 1, 21)
 141: (4)                fig, ax = plt.subplots()
 142: (4)                ax.axvline(t0, color="red", lw=3)
 143: (4)                ax.set_xlim(t0 - datetime.timedelta(days=5),
 144: (16)                           tf + datetime.timedelta(days=5))
 145: (4)                fig.autofmt_xdate()
 146: (0)            def test_too_many_date_ticks(caplog):
 147: (4)                caplog.set_level("WARNING")
 148: (4)                t0 = datetime.datetime(2000, 1, 20)
 149: (4)                tf = datetime.datetime(2000, 1, 20)
 150: (4)                fig, ax = plt.subplots()
 151: (4)                with pytest.warns(UserWarning) as rec:
 152: (8)                    ax.set_xlim((t0, tf), auto=True)
 153: (8)                    assert len(rec) == 1
 154: (8)                    assert ('Attempting to set identical low and high xlims'
 155: (16)                           in str(rec[0].message))
 156: (4)                ax.plot([], [])
```

```
157: (4)                ax.xaxis.set_major_locator(mdates.DayLocator())
158: (4)                v = ax.xaxis.get_major_locator()()
159: (4)                assert len(v) > 1000
160: (4)                assert caplog.records and all(
161: (8)                    record.name == "matplotlib.ticker" and record.levelname == "WARNING"
162: (8)                    for record in caplog.records)
163: (4)                assert len(caplog.records) > 0
164: (0)            def _new_epoch_decorator(thefunc):
165: (4)                @functools.wraps(thefunc)
166: (4)                def wrapper():
167: (8)                    mdates._reset_epoch_test_example()
168: (8)                    mdates.set_epoch('2000-01-01')
169: (8)                    thefunc()
170: (8)                    mdates._reset_epoch_test_example()
171: (4)                return wrapper
172: (0)            @image_comparison(['RRuleLocator_bounds.png'])
173: (0)            def test_RRuleLocator():
174: (4)                import matplotlib.testing.jpl_units as units
175: (4)                units.register()
176: (4)                t0 = datetime.datetime(1000, 1, 1)
177: (4)                tf = datetime.datetime(6000, 1, 1)
178: (4)                fig = plt.figure()
179: (4)                ax = plt.subplot()
180: (4)                ax.set_autoscale_on(True)
181: (4)                ax.plot([t0, tf], [0.0, 1.0], marker='o')
182: (4)                rrule = mdates.rrulewrapper(dateutil.rrule.YEARLY, interval=500)
183: (4)                locator = mdates.RRuleLocator(rrule)
184: (4)                ax.xaxis.set_major_locator(locator)
185: (4)                ax.xaxis.set_major_formatter(mdates.AutoDateFormatter(locator))
186: (4)                ax.autoscale_view()
187: (4)                fig.autofmt_xdate()
188: (0)            def test_RRuleLocator_dayrange():
189: (4)                loc = mdates.DayLocator()
190: (4)                x1 = datetime.datetime(year=1, month=1, day=1, tzinfo=mdates.UTC)
191: (4)                y1 = datetime.datetime(year=1, month=1, day=16, tzinfo=mdates.UTC)
192: (4)                loc.tick_values(x1, y1)
193: (0)            def test_RRuleLocator_close_minmax():
194: (4)                rrule = mdates.rrulewrapper(dateutil.rrule.SECONDLY, interval=5)
195: (4)                loc = mdates.RRuleLocator(rrule)
196: (4)                d1 = datetime.datetime(year=2020, month=1, day=1)
197: (4)                d2 = datetime.datetime(year=2020, month=1, day=1, microsecond=1)
198: (4)                expected = ['2020-01-01 00:00:00+00:00',
199: (16)                            '2020-01-01 00:00:00.000001+00:00']
200: (4)                assert list(map(str, mdates.num2date(loc.tick_values(d1, d2)))) ==
expected
201: (0)            @image_comparison(['DateFormatter_fractionalSeconds.png'])
202: (0)            def test_DateFormatter():
203: (4)                import matplotlib.testing.jpl_units as units
204: (4)                units.register()
205: (4)                t0 = datetime.datetime(2001, 1, 1, 0, 0, 0)
206: (4)                tf = datetime.datetime(2001, 1, 1, 0, 0, 1)
207: (4)                fig = plt.figure()
208: (4)                ax = plt.subplot()
209: (4)                ax.set_autoscale_on(True)
210: (4)                ax.plot([t0, tf], [0.0, 1.0], marker='o')
211: (4)                ax.autoscale_view()
212: (4)                fig.autofmt_xdate()
213: (0)            def test_locator_set_formatter():
214: (4)                """
215: (4)                Test if setting the locator only will update the AutoDateFormatter to use
216: (4)                the new locator.
217: (4)                """
218: (4)                plt.rcParams["date.autoformatter.minute"] = "%d %H:%M"
219: (4)                t = [datetime.datetime(2018, 9, 30, 8, 0),
220: (9)                     datetime.datetime(2018, 9, 30, 8, 59),
221: (9)                     datetime.datetime(2018, 9, 30, 10, 30)]
222: (4)                x = [2, 3, 1]
223: (4)                fig, ax = plt.subplots()
224: (4)                ax.plot(t, x)
```

```
225: (4)                ax.xaxis.set_major_locator(mdates.MinuteLocator((0, 30)))
226: (4)                fig.canvas.draw()
227: (4)                ticklabels = [tl.get_text() for tl in ax.get_xticklabels()]
228: (4)                expected = ['30 08:00', '30 08:30', '30 09:00',
229: (16)                           '30 09:30', '30 10:00', '30 10:30']
230: (4)                assert ticklabels == expected
231: (4)                ax.xaxis.set_major_locator(mticker.NullLocator())
232: (4)                ax.xaxis.set_minor_locator(mdates.MinuteLocator((5, 55)))
233: (4)                decoy_loc = mdates.MinuteLocator((12, 27))
234: (4)                ax.xaxis.set_minor_formatter(mdates.AutoDateFormatter(decoy_loc))
235: (4)                ax.xaxis.set_minor_locator(mdates.MinuteLocator((15, 45)))
236: (4)                fig.canvas.draw()
237: (4)                ticklabels = [tl.get_text() for tl in ax.get_xticklabels(which="minor")]
238: (4)                expected = ['30 08:15', '30 08:45', '30 09:15', '30 09:45', '30 10:15']
239: (4)                assert ticklabels == expected
240: (0)            def test_date_formatter_callable():
241: (4)                class _Locator:
242: (8)                    def _get_unit(self): return -11
243: (4)                def callable_formatting_function(dates, _):
244: (8)                    return [dt.strftime('%d-%m//%Y') for dt in dates]
245: (4)                formatter = mdates.AutoDateFormatter(_Locator())
246: (4)                formatter.scaled[-10] = callable_formatting_function
247: (4)                assert formatter([datetime.datetime(2014, 12, 25)]) == ['25-12//2014']
248: (0)            @pytest.mark.parametrize('delta, expected', [
249: (4)                (datetime.timedelta(weeks=52 * 200),
250: (5)                 [r'$\mathdefault{%d}$' % year for year in range(1990, 2171, 20)]),
251: (4)                (datetime.timedelta(days=30),
252: (5)                 [r'$\mathdefault{1990{-}01{-}%02d}$' % day for day in range(1, 32, 3)]),
253: (4)                (datetime.timedelta(hours=20),
254: (5)                 [r'$\mathdefault{01{-}01\;%02d}$' % hour for hour in range(0, 21, 2)]),
255: (4)                (datetime.timedelta(minutes=10),
256: (5)                 [r'$\mathdefault{01\;00{:}%02d}$' % minu for minu in range(0, 11)]),
257: (0)            ])
258: (0)            def test_date_formatter_usetex(delta, expected):
259: (4)                style.use("default")
260: (4)                d1 = datetime.datetime(1990, 1, 1)
261: (4)                d2 = d1 + delta
262: (4)                locator = mdates.AutoDateLocator(interval_multiples=False)
263: (4)                locator.create_dummy_axis()
264: (4)                locator.axis.set_view_interval(mdates.date2num(d1), mdates.date2num(d2))
265: (4)                formatter = mdates.AutoDateFormatter(locator, usetex=True)
266: (4)                assert [formatter(loc) for loc in locator()] == expected
267: (0)            def test_drange():
268: (4)                """
269: (4)                This test should check if drange works as expected, and if all the
270: (4)                rounding errors are fixed
271: (4)                """
272: (4)                start = datetime.datetime(2011, 1, 1, tzinfo=mdates.UTC)
273: (4)                end = datetime.datetime(2011, 1, 2, tzinfo=mdates.UTC)
274: (4)                delta = datetime.timedelta(hours=1)
275: (4)                assert len(mdates.drange(start, end, delta)) == 24
276: (4)                end = end - datetime.timedelta(microseconds=1)
277: (4)                assert len(mdates.drange(start, end, delta)) == 24
278: (4)                end = end + datetime.timedelta(microseconds=2)
279: (4)                assert len(mdates.drange(start, end, delta)) == 25
280: (4)                end = datetime.datetime(2011, 1, 2, tzinfo=mdates.UTC)
281: (4)                delta = datetime.timedelta(hours=4)
282: (4)                daterange = mdates.drange(start, end, delta)
283: (4)                assert len(daterange) == 6
284: (4)                assert mdates.num2date(daterange[-1]) == (end - delta)
285: (0)            @_new_epoch_decorator
286: (0)            def test_auto_date_locator():
287: (4)                def _create_auto_date_locator(date1, date2):
288: (8)                    locator = mdates.AutoDateLocator(interval_multiples=False)
289: (8)                    locator.create_dummy_axis()
290: (8)                    locator.axis.set_view_interval(*mdates.date2num([date1, date2]))
291: (8)                    return locator
292: (4)                d1 = datetime.datetime(1990, 1, 1)
293: (4)                results = ([datetime.timedelta(weeks=52 * 200),
```

```
294: (16)                            ['1990-01-01 00:00:00+00:00', '2010-01-01 00:00:00+00:00',
295: (17)                             '2030-01-01 00:00:00+00:00', '2050-01-01 00:00:00+00:00',
296: (17)                             '2070-01-01 00:00:00+00:00', '2090-01-01 00:00:00+00:00',
297: (17)                             '2110-01-01 00:00:00+00:00', '2130-01-01 00:00:00+00:00',
298: (17)                             '2150-01-01 00:00:00+00:00', '2170-01-01 00:00:00+00:00']
299: (16)                            ],
300: (15)                           [datetime.timedelta(weeks=52),
301: (16)                            ['1990-01-01 00:00:00+00:00', '1990-02-01 00:00:00+00:00',
302: (17)                             '1990-03-01 00:00:00+00:00', '1990-04-01 00:00:00+00:00',
303: (17)                             '1990-05-01 00:00:00+00:00', '1990-06-01 00:00:00+00:00',
304: (17)                             '1990-07-01 00:00:00+00:00', '1990-08-01 00:00:00+00:00',
305: (17)                             '1990-09-01 00:00:00+00:00', '1990-10-01 00:00:00+00:00',
306: (17)                             '1990-11-01 00:00:00+00:00', '1990-12-01 00:00:00+00:00']
307: (16)                            ],
308: (15)                           [datetime.timedelta(days=141),
309: (16)                            ['1990-01-05 00:00:00+00:00', '1990-01-26 00:00:00+00:00',
310: (17)                             '1990-02-16 00:00:00+00:00', '1990-03-09 00:00:00+00:00',
311: (17)                             '1990-03-30 00:00:00+00:00', '1990-04-20 00:00:00+00:00',
312: (17)                             '1990-05-11 00:00:00+00:00']
313: (16)                            ],
314: (15)                           [datetime.timedelta(days=40),
315: (16)                            ['1990-01-03 00:00:00+00:00', '1990-01-10 00:00:00+00:00',
316: (17)                             '1990-01-17 00:00:00+00:00', '1990-01-24 00:00:00+00:00',
317: (17)                             '1990-01-31 00:00:00+00:00', '1990-02-07 00:00:00+00:00']
318: (16)                            ],
319: (15)                           [datetime.timedelta(hours=40),
320: (16)                            ['1990-01-01 00:00:00+00:00', '1990-01-01 04:00:00+00:00',
321: (17)                             '1990-01-01 08:00:00+00:00', '1990-01-01 12:00:00+00:00',
322: (17)                             '1990-01-01 16:00:00+00:00', '1990-01-01 20:00:00+00:00',
323: (17)                             '1990-01-02 00:00:00+00:00', '1990-01-02 04:00:00+00:00',
324: (17)                             '1990-01-02 08:00:00+00:00', '1990-01-02 12:00:00+00:00',
325: (17)                             '1990-01-02 16:00:00+00:00']
326: (16)                            ],
327: (15)                           [datetime.timedelta(minutes=20),
328: (16)                            ['1990-01-01 00:00:00+00:00', '1990-01-01 00:05:00+00:00',
329: (17)                             '1990-01-01 00:10:00+00:00', '1990-01-01 00:15:00+00:00',
330: (17)                             '1990-01-01 00:20:00+00:00']
331: (16)                            ],
332: (15)                           [datetime.timedelta(seconds=40),
333: (16)                            ['1990-01-01 00:00:00+00:00', '1990-01-01 00:00:05+00:00',
334: (17)                             '1990-01-01 00:00:10+00:00', '1990-01-01 00:00:15+00:00',
335: (17)                             '1990-01-01 00:00:20+00:00', '1990-01-01 00:00:25+00:00',
336: (17)                             '1990-01-01 00:00:30+00:00', '1990-01-01 00:00:35+00:00',
337: (17)                             '1990-01-01 00:00:40+00:00']
338: (16)                            ],
339: (15)                           [datetime.timedelta(microseconds=1500),
340: (16)                            ['1989-12-31 23:59:59.999500+00:00',
341: (17)                             '1990-01-01 00:00:00+00:00',
342: (17)                             '1990-01-01 00:00:00.000500+00:00',
343: (17)                             '1990-01-01 00:00:00.001000+00:00',
344: (17)                             '1990-01-01 00:00:00.001500+00:00',
345: (17)                             '1990-01-01 00:00:00.002000+00:00']
346: (16)                            ],
347: (15)                           )
348: (4)                 for t_delta, expected in results:
349: (8)                     d2 = d1 + t_delta
350: (8)                     locator = _create_auto_date_locator(d1, d2)
351: (8)                     assert list(map(str, mdates.num2date(locator()))) == expected
352: (4)                 locator = mdates.AutoDateLocator(interval_multiples=False)
353: (4)                 assert locator.maxticks == {0: 11, 1: 12, 3: 11, 4: 12, 5: 11, 6: 11, 7:
8}
354: (4)                 locator = mdates.AutoDateLocator(maxticks={dateutil.rrule.MONTHLY: 5})
355: (4)                 assert locator.maxticks == {0: 11, 1: 5, 3: 11, 4: 12, 5: 11, 6: 11, 7: 8}
356: (4)                 locator = mdates.AutoDateLocator(maxticks=5)
357: (4)                 assert locator.maxticks == {0: 5, 1: 5, 3: 5, 4: 5, 5: 5, 6: 5, 7: 5}
358: (0)             @_new_epoch_decorator
359: (0)             def test_auto_date_locator_intmult():
360: (4)                 def _create_auto_date_locator(date1, date2):
361: (8)                     locator = mdates.AutoDateLocator(interval_multiples=True)
```

```
(8)                     locator.create_dummy_axis()
(8)                     locator.axis.set_view_interval(*mdates.date2num([date1, date2]))
(8)                     return locator
(4)                 results = ([datetime.timedelta(weeks=52 * 200),
(16)                            ['1980-01-01 00:00:00+00:00', '2000-01-01 00:00:00+00:00',
(17)                             '2020-01-01 00:00:00+00:00', '2040-01-01 00:00:00+00:00',
(17)                             '2060-01-01 00:00:00+00:00', '2080-01-01 00:00:00+00:00',
(17)                             '2100-01-01 00:00:00+00:00', '2120-01-01 00:00:00+00:00',
(17)                             '2140-01-01 00:00:00+00:00', '2160-01-01 00:00:00+00:00',
(17)                             '2180-01-01 00:00:00+00:00', '2200-01-01 00:00:00+00:00']
(16)                            ],
(15)                           [datetime.timedelta(weeks=52),
(16)                            ['1997-01-01 00:00:00+00:00', '1997-02-01 00:00:00+00:00',
(17)                             '1997-03-01 00:00:00+00:00', '1997-04-01 00:00:00+00:00',
(17)                             '1997-05-01 00:00:00+00:00', '1997-06-01 00:00:00+00:00',
(17)                             '1997-07-01 00:00:00+00:00', '1997-08-01 00:00:00+00:00',
(17)                             '1997-09-01 00:00:00+00:00', '1997-10-01 00:00:00+00:00',
(17)                             '1997-11-01 00:00:00+00:00', '1997-12-01 00:00:00+00:00']
(16)                            ],
(15)                           [datetime.timedelta(days=141),
(16)                            ['1997-01-01 00:00:00+00:00', '1997-01-15 00:00:00+00:00',
(17)                             '1997-02-01 00:00:00+00:00', '1997-02-15 00:00:00+00:00',
(17)                             '1997-03-01 00:00:00+00:00', '1997-03-15 00:00:00+00:00',
(17)                             '1997-04-01 00:00:00+00:00', '1997-04-15 00:00:00+00:00',
(17)                             '1997-05-01 00:00:00+00:00', '1997-05-15 00:00:00+00:00']
(16)                            ],
(15)                           [datetime.timedelta(days=40),
(16)                            ['1997-01-01 00:00:00+00:00', '1997-01-05 00:00:00+00:00',
(17)                             '1997-01-09 00:00:00+00:00', '1997-01-13 00:00:00+00:00',
(17)                             '1997-01-17 00:00:00+00:00', '1997-01-21 00:00:00+00:00',
(17)                             '1997-01-25 00:00:00+00:00', '1997-01-29 00:00:00+00:00',
(17)                             '1997-02-01 00:00:00+00:00', '1997-02-05 00:00:00+00:00',
(17)                             '1997-02-09 00:00:00+00:00']
(16)                            ],
(15)                           [datetime.timedelta(hours=40),
(16)                            ['1997-01-01 00:00:00+00:00', '1997-01-01 04:00:00+00:00',
(17)                             '1997-01-01 08:00:00+00:00', '1997-01-01 12:00:00+00:00',
(17)                             '1997-01-01 16:00:00+00:00', '1997-01-01 20:00:00+00:00',
(17)                             '1997-01-02 00:00:00+00:00', '1997-01-02 04:00:00+00:00',
(17)                             '1997-01-02 08:00:00+00:00', '1997-01-02 12:00:00+00:00',
(17)                             '1997-01-02 16:00:00+00:00']
(16)                            ],
(15)                           [datetime.timedelta(minutes=20),
(16)                            ['1997-01-01 00:00:00+00:00', '1997-01-01 00:05:00+00:00',
(17)                             '1997-01-01 00:10:00+00:00', '1997-01-01 00:15:00+00:00',
(17)                             '1997-01-01 00:20:00+00:00']
(16)                            ],
(15)                           [datetime.timedelta(seconds=40),
(16)                            ['1997-01-01 00:00:00+00:00', '1997-01-01 00:00:05+00:00',
(17)                             '1997-01-01 00:00:10+00:00', '1997-01-01 00:00:15+00:00',
(17)                             '1997-01-01 00:00:20+00:00', '1997-01-01 00:00:25+00:00',
(17)                             '1997-01-01 00:00:30+00:00', '1997-01-01 00:00:35+00:00',
(17)                             '1997-01-01 00:00:40+00:00']
(16)                            ],
(15)                           [datetime.timedelta(microseconds=1500),
(16)                            ['1996-12-31 23:59:59.999500+00:00',
(17)                             '1997-01-01 00:00:00+00:00',
(17)                             '1997-01-01 00:00:00.000500+00:00',
(17)                             '1997-01-01 00:00:00.001000+00:00',
(17)                             '1997-01-01 00:00:00.001500+00:00',
(17)                             '1997-01-01 00:00:00.002000+00:00']
(16)                            ],
(15)                           )
(4)                 d1 = datetime.datetime(1997, 1, 1)
(4)                 for t_delta, expected in results:
(8)                     d2 = d1 + t_delta
(8)                     locator = _create_auto_date_locator(d1, d2)
(8)                     assert list(map(str, mdates.num2date(locator()))) == expected
(0)             def test_concise_formatter_subsecond():
```

```
(4)                 locator = mdates.AutoDateLocator(interval_multiples=True)
(4)                 formatter = mdates.ConciseDateFormatter(locator)
(4)                 year_1996 = 9861.0
(4)                 strings = formatter.format_ticks([
(8)                     year_1996,
(8)                     year_1996 + 500 / mdates.MUSECONDS_PER_DAY,
(8)                     year_1996 + 900 / mdates.MUSECONDS_PER_DAY])
(4)                 assert strings == ['00:00', '00.0005', '00.0009']
(0)             def test_concise_formatter():
(4)                 def _create_auto_date_locator(date1, date2):
(8)                     fig, ax = plt.subplots()
(8)                     locator = mdates.AutoDateLocator(interval_multiples=True)
(8)                     formatter = mdates.ConciseDateFormatter(locator)
(8)                     ax.yaxis.set_major_locator(locator)
(8)                     ax.yaxis.set_major_formatter(formatter)
(8)                     ax.set_ylim(date1, date2)
(8)                     fig.canvas.draw()
(8)                     sts = [st.get_text() for st in ax.get_yticklabels()]
(8)                     return sts
(4)                 d1 = datetime.datetime(1997, 1, 1)
(4)                 results = ([datetime.timedelta(weeks=52 * 200),
(16)                            [str(t) for t in range(1980, 2201, 20)]
(16)                            ],
(15)                           [datetime.timedelta(weeks=52),
(16)                            ['1997', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug',
(17)                             'Sep', 'Oct', 'Nov', 'Dec']
(16)                            ],
(15)                           [datetime.timedelta(days=141),
(16)                            ['Jan', '15', 'Feb', '15', 'Mar', '15', 'Apr', '15',
(17)                             'May', '15']
(16)                            ],
(15)                           [datetime.timedelta(days=40),
(16)                            ['Jan', '05', '09', '13', '17', '21', '25', '29', 'Feb',
(17)                             '05', '09']
(16)                            ],
(15)                           [datetime.timedelta(hours=40),
(16)                            ['Jan-01', '04:00', '08:00', '12:00', '16:00', '20:00',
(17)                             'Jan-02', '04:00', '08:00', '12:00', '16:00']
(16)                            ],
(15)                           [datetime.timedelta(minutes=20),
(16)                            ['00:00', '00:05', '00:10', '00:15', '00:20']
(16)                            ],
(15)                           [datetime.timedelta(seconds=40),
(16)                            ['00:00', '05', '10', '15', '20', '25', '30', '35', '40']
(16)                            ],
(15)                           [datetime.timedelta(seconds=2),
(16)                            ['59.5', '00:00', '00.5', '01.0', '01.5', '02.0', '02.5']
(16)                            ],
(15)                           )
(4)                 for t_delta, expected in results:
(8)                     d2 = d1 + t_delta
(8)                     strings = _create_auto_date_locator(d1, d2)
(8)                     assert strings == expected
(0)             @pytest.mark.parametrize('t_delta, expected', [
(4)                 (datetime.timedelta(seconds=0.01), '1997-Jan-01 00:00'),
(4)                 (datetime.timedelta(minutes=1), '1997-Jan-01 00:01'),
(4)                 (datetime.timedelta(hours=1), '1997-Jan-01'),
(4)                 (datetime.timedelta(days=1), '1997-Jan-02'),
(4)                 (datetime.timedelta(weeks=1), '1997-Jan'),
(4)                 (datetime.timedelta(weeks=26), ''),
(4)                 (datetime.timedelta(weeks=520), '')
(0)             ])
(0)             def test_concise_formatter_show_offset(t_delta, expected):
(4)                 d1 = datetime.datetime(1997, 1, 1)
(4)                 d2 = d1 + t_delta
(4)                 fig, ax = plt.subplots()
(4)                 locator = mdates.AutoDateLocator()
(4)                 formatter = mdates.ConciseDateFormatter(locator)
(4)                 ax.xaxis.set_major_locator(locator)
```

```
500: (4)                ax.xaxis.set_major_formatter(formatter)
501: (4)                ax.plot([d1, d2], [0, 0])
502: (4)                fig.canvas.draw()
503: (4)                assert formatter.get_offset() == expected
504: (0)            def test_concise_converter_stays():
505: (4)                x = [datetime.datetime(2000, 1, 1), datetime.datetime(2020, 2, 20)]
506: (4)                y = [0, 1]
507: (4)                fig, ax = plt.subplots()
508: (4)                ax.plot(x, y)
509: (4)                ax.xaxis.converter = conv = mdates.ConciseDateConverter()
510: (4)                assert ax.xaxis.units is None
511: (4)                ax.set_xlim(*x)
512: (4)                assert ax.xaxis.converter == conv
513: (0)            def test_offset_changes():
514: (4)                fig, ax = plt.subplots()
515: (4)                d1 = datetime.datetime(1997, 1, 1)
516: (4)                d2 = d1 + datetime.timedelta(weeks=520)
517: (4)                locator = mdates.AutoDateLocator()
518: (4)                formatter = mdates.ConciseDateFormatter(locator)
519: (4)                ax.xaxis.set_major_locator(locator)
520: (4)                ax.xaxis.set_major_formatter(formatter)
521: (4)                ax.plot([d1, d2], [0, 0])
522: (4)                fig.draw_without_rendering()
523: (4)                assert formatter.get_offset() == ''
524: (4)                ax.set_xlim(d1, d1 + datetime.timedelta(weeks=3))
525: (4)                fig.draw_without_rendering()
526: (4)                assert formatter.get_offset() == '1997-Jan'
527: (4)                ax.set_xlim(d1 + datetime.timedelta(weeks=7),
528: (16)                           d1 + datetime.timedelta(weeks=30))
529: (4)                fig.draw_without_rendering()
530: (4)                assert formatter.get_offset() == '1997'
531: (4)                ax.set_xlim(d1, d1 + datetime.timedelta(weeks=520))
532: (4)                fig.draw_without_rendering()
533: (4)                assert formatter.get_offset() == ''
534: (0)            @pytest.mark.parametrize('t_delta, expected', [
535: (4)                (datetime.timedelta(weeks=52 * 200),
536: (5)                 ['$\\mathdefault{%d}$' % (t, ) for t in range(1980, 2201, 20)]),
537: (4)                (datetime.timedelta(days=40),
538: (5)                 ['Jan', '$\\mathdefault{05}$', '$\\mathdefault{09}$',
539: (6)                  '$\\mathdefault{13}$', '$\\mathdefault{17}$', '$\\mathdefault{21}$',
540: (6)                  '$\\mathdefault{25}$', '$\\mathdefault{29}$', 'Feb',
541: (6)                  '$\\mathdefault{05}$', '$\\mathdefault{09}$']),
542: (4)                (datetime.timedelta(hours=40),
543: (5)                 ['Jan$\\mathdefault{{-}01}$', '$\\mathdefault{04{:}00}$',
544: (6)                  '$\\mathdefault{08{:}00}$', '$\\mathdefault{12{:}00}$',
545: (6)                  '$\\mathdefault{16{:}00}$', '$\\mathdefault{20{:}00}$',
546: (6)                  'Jan$\\mathdefault{{-}02}$', '$\\mathdefault{04{:}00}$',
547: (6)                  '$\\mathdefault{08{:}00}$', '$\\mathdefault{12{:}00}$',
548: (6)                  '$\\mathdefault{16{:}00}$']),
549: (4)                (datetime.timedelta(seconds=2),
550: (5)                 ['$\\mathdefault{59.5}$', '$\\mathdefault{00{:}00}$',
551: (6)                  '$\\mathdefault{00.5}$', '$\\mathdefault{01.0}$',
552: (6)                  '$\\mathdefault{01.5}$', '$\\mathdefault{02.0}$',
553: (6)                  '$\\mathdefault{02.5}$']),
554: (0)            ])
555: (0)            def test_concise_formatter_usetex(t_delta, expected):
556: (4)                d1 = datetime.datetime(1997, 1, 1)
557: (4)                d2 = d1 + t_delta
558: (4)                locator = mdates.AutoDateLocator(interval_multiples=True)
559: (4)                locator.create_dummy_axis()
560: (4)                locator.axis.set_view_interval(mdates.date2num(d1), mdates.date2num(d2))
561: (4)                formatter = mdates.ConciseDateFormatter(locator, usetex=True)
562: (4)                assert formatter.format_ticks(locator()) == expected
563: (0)            def test_concise_formatter_formats():
564: (4)                formats = ['%Y', '%m/%Y', 'day: %d',
565: (15)                           '%H hr %M min', '%H hr %M min', '%S.%f sec']
566: (4)                def _create_auto_date_locator(date1, date2):
567: (8)                    fig, ax = plt.subplots()
568: (8)                    locator = mdates.AutoDateLocator(interval_multiples=True)
```

```
(8)                     formatter = mdates.ConciseDateFormatter(locator, formats=formats)
(8)                     ax.yaxis.set_major_locator(locator)
(8)                     ax.yaxis.set_major_formatter(formatter)
(8)                     ax.set_ylim(date1, date2)
(8)                     fig.canvas.draw()
(8)                     sts = [st.get_text() for st in ax.get_yticklabels()]
(8)                     return sts
(4)                 d1 = datetime.datetime(1997, 1, 1)
(4)                 results = (
(8)                     [datetime.timedelta(weeks=52 * 200), [str(t) for t in range(1980,
(9)                      2201, 20)]],
(8)                     [datetime.timedelta(weeks=52), [
(12)                        '1997', '02/1997', '03/1997', '04/1997', '05/1997', '06/1997',
(12)                        '07/1997', '08/1997', '09/1997', '10/1997', '11/1997', '12/1997',
(12)                        ]],
(8)                     [datetime.timedelta(days=141), [
(12)                        '01/1997', 'day: 15', '02/1997', 'day: 15', '03/1997', 'day: 15',
(12)                        '04/1997', 'day: 15', '05/1997', 'day: 15',
(12)                        ]],
(8)                     [datetime.timedelta(days=40), [
(12)                        '01/1997', 'day: 05', 'day: 09', 'day: 13', 'day: 17', 'day: 21',
(12)                        'day: 25', 'day: 29', '02/1997', 'day: 05', 'day: 09',
(12)                        ]],
(8)                     [datetime.timedelta(hours=40), [
(12)                        'day: 01', '04 hr 00 min', '08 hr 00 min', '12 hr 00 min',
(12)                        '16 hr 00 min', '20 hr 00 min', 'day: 02', '04 hr 00 min',
(12)                        '08 hr 00 min', '12 hr 00 min', '16 hr 00 min',
(12)                        ]],
(8)                     [datetime.timedelta(minutes=20), ['00 hr 00 min', '00 hr 05 min',
(9)                      '00 hr 10 min', '00 hr 15 min', '00 hr 20 min']],
(8)                     [datetime.timedelta(seconds=40), [
(12)                        '00 hr 00 min', '05.000000 sec', '10.000000 sec',
(12)                        '15.000000 sec', '20.000000 sec', '25.000000 sec',
(12)                        '30.000000 sec', '35.000000 sec', '40.000000 sec',
(12)                        ]],
(8)                     [datetime.timedelta(seconds=2), [
(12)                        '59.500000 sec', '00 hr 00 min', '00.500000 sec', '01.000000 sec',
(12)                        '01.500000 sec', '02.000000 sec', '02.500000 sec',
(12)                        ]],
(8)                     )
(4)                 for t_delta, expected in results:
(8)                     d2 = d1 + t_delta
(8)                     strings = _create_auto_date_locator(d1, d2)
(8)                     assert strings == expected
(0)             def test_concise_formatter_zformats():
(4)                 zero_formats = ['', "'%y", '%B', '%m-%d', '%S', '%S.%f']
(4)                 def _create_auto_date_locator(date1, date2):
(8)                     fig, ax = plt.subplots()
(8)                     locator = mdates.AutoDateLocator(interval_multiples=True)
(8)                     formatter = mdates.ConciseDateFormatter(
(12)                        locator, zero_formats=zero_formats)
(8)                     ax.yaxis.set_major_locator(locator)
(8)                     ax.yaxis.set_major_formatter(formatter)
(8)                     ax.set_ylim(date1, date2)
(8)                     fig.canvas.draw()
(8)                     sts = [st.get_text() for st in ax.get_yticklabels()]
(8)                     return sts
(4)                 d1 = datetime.datetime(1997, 1, 1)
(4)                 results = ([datetime.timedelta(weeks=52 * 200),
(16)                            [str(t) for t in range(1980, 2201, 20)]
(16)                            ],
(15)                           [datetime.timedelta(weeks=52),
(16)                            ["'97", 'Feb', 'Mar', 'Apr', 'May', 'Jun',
(20)                                'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
(16)                            ],
(15)                           [datetime.timedelta(days=141),
(16)                            ['January', '15', 'February', '15', 'March',
(20)                                '15', 'April', '15', 'May', '15']
(16)                            ],
```

```
(15)                            [datetime.timedelta(days=40),
(16)                             ['January', '05', '09', '13', '17', '21',
(20)                                 '25', '29', 'February', '05', '09']
(16)                             ],
(15)                            [datetime.timedelta(hours=40),
(16)                             ['01-01', '04:00', '08:00', '12:00', '16:00', '20:00',
(20)                                 '01-02', '04:00', '08:00', '12:00', '16:00']
(16)                             ],
(15)                            [datetime.timedelta(minutes=20),
(16)                             ['00', '00:05', '00:10', '00:15', '00:20']
(16)                             ],
(15)                            [datetime.timedelta(seconds=40),
(16)                             ['00', '05', '10', '15', '20', '25', '30', '35', '40']
(16)                             ],
(15)                            [datetime.timedelta(seconds=2),
(16)                             ['59.5', '00.0', '00.5', '01.0', '01.5', '02.0', '02.5']
(16)                             ],
(15)                            )
(4)                 for t_delta, expected in results:
(8)                     d2 = d1 + t_delta
(8)                     strings = _create_auto_date_locator(d1, d2)
(8)                     assert strings == expected
(0)             def test_concise_formatter_tz():
(4)                 def _create_auto_date_locator(date1, date2, tz):
(8)                     fig, ax = plt.subplots()
(8)                     locator = mdates.AutoDateLocator(interval_multiples=True)
(8)                     formatter = mdates.ConciseDateFormatter(locator, tz=tz)
(8)                     ax.yaxis.set_major_locator(locator)
(8)                     ax.yaxis.set_major_formatter(formatter)
(8)                     ax.set_ylim(date1, date2)
(8)                     fig.canvas.draw()
(8)                     sts = [st.get_text() for st in ax.get_yticklabels()]
(8)                     return sts, ax.yaxis.get_offset_text().get_text()
(4)                 d1 = datetime.datetime(1997, 1, 1).replace(tzinfo=datetime.timezone.utc)
(4)                 results = ([datetime.timedelta(hours=40),
(16)                             ['03:00', '07:00', '11:00', '15:00', '19:00', '23:00',
(17)                              '03:00', '07:00', '11:00', '15:00', '19:00'],
(16)                             "1997-Jan-02"
(16)                             ],
(15)                            [datetime.timedelta(minutes=20),
(16)                             ['03:00', '03:05', '03:10', '03:15', '03:20'],
(16)                             "1997-Jan-01"
(16)                             ],
(15)                            [datetime.timedelta(seconds=40),
(16)                             ['03:00', '05', '10', '15', '20', '25', '30', '35', '40'],
(16)                             "1997-Jan-01 03:00"
(16)                             ],
(15)                            [datetime.timedelta(seconds=2),
(16)                             ['59.5', '03:00', '00.5', '01.0', '01.5', '02.0', '02.5'],
(16)                             "1997-Jan-01 03:00"
(16)                             ],
(15)                            )
(4)                 new_tz = datetime.timezone(datetime.timedelta(hours=3))
(4)                 for t_delta, expected_strings, expected_offset in results:
(8)                     d2 = d1 + t_delta
(8)                     strings, offset = _create_auto_date_locator(d1, d2, new_tz)
(8)                     assert strings == expected_strings
(8)                     assert offset == expected_offset
(0)             def test_auto_date_locator_intmult_tz():
(4)                 def _create_auto_date_locator(date1, date2, tz):
(8)                     locator = mdates.AutoDateLocator(interval_multiples=True, tz=tz)
(8)                     locator.create_dummy_axis()
(8)                     locator.axis.set_view_interval(*mdates.date2num([date1, date2]))
(8)                     return locator
(4)                 results = ([datetime.timedelta(weeks=52*200),
(16)                             ['1980-01-01 00:00:00-08:00', '2000-01-01 00:00:00-08:00',
(17)                              '2020-01-01 00:00:00-08:00', '2040-01-01 00:00:00-08:00',
(17)                              '2060-01-01 00:00:00-08:00', '2080-01-01 00:00:00-08:00',
(17)                              '2100-01-01 00:00:00-08:00', '2120-01-01 00:00:00-08:00',
```

```
 707: (17)                            '2140-01-01 00:00:00-08:00', '2160-01-01 00:00:00-08:00',
 708: (17)                            '2180-01-01 00:00:00-08:00', '2200-01-01 00:00:00-08:00']
 709: (16)                           ],
 710: (15)                          [datetime.timedelta(weeks=52),
 711: (16)                           ['1997-01-01 00:00:00-08:00', '1997-02-01 00:00:00-08:00',
 712: (17)                            '1997-03-01 00:00:00-08:00', '1997-04-01 00:00:00-08:00',
 713: (17)                            '1997-05-01 00:00:00-07:00', '1997-06-01 00:00:00-07:00',
 714: (17)                            '1997-07-01 00:00:00-07:00', '1997-08-01 00:00:00-07:00',
 715: (17)                            '1997-09-01 00:00:00-07:00', '1997-10-01 00:00:00-07:00',
 716: (17)                            '1997-11-01 00:00:00-08:00', '1997-12-01 00:00:00-08:00']
 717: (16)                           ],
 718: (15)                          [datetime.timedelta(days=141),
 719: (16)                           ['1997-01-01 00:00:00-08:00', '1997-01-15 00:00:00-08:00',
 720: (17)                            '1997-02-01 00:00:00-08:00', '1997-02-15 00:00:00-08:00',
 721: (17)                            '1997-03-01 00:00:00-08:00', '1997-03-15 00:00:00-08:00',
 722: (17)                            '1997-04-01 00:00:00-08:00', '1997-04-15 00:00:00-07:00',
 723: (17)                            '1997-05-01 00:00:00-07:00', '1997-05-15 00:00:00-07:00']
 724: (16)                           ],
 725: (15)                          [datetime.timedelta(days=40),
 726: (16)                           ['1997-01-01 00:00:00-08:00', '1997-01-05 00:00:00-08:00',
 727: (17)                            '1997-01-09 00:00:00-08:00', '1997-01-13 00:00:00-08:00',
 728: (17)                            '1997-01-17 00:00:00-08:00', '1997-01-21 00:00:00-08:00',
 729: (17)                            '1997-01-25 00:00:00-08:00', '1997-01-29 00:00:00-08:00',
 730: (17)                            '1997-02-01 00:00:00-08:00', '1997-02-05 00:00:00-08:00',
 731: (17)                            '1997-02-09 00:00:00-08:00']
 732: (16)                           ],
 733: (15)                          [datetime.timedelta(hours=40),
 734: (16)                           ['1997-01-01 00:00:00-08:00', '1997-01-01 04:00:00-08:00',
 735: (17)                            '1997-01-01 08:00:00-08:00', '1997-01-01 12:00:00-08:00',
 736: (17)                            '1997-01-01 16:00:00-08:00', '1997-01-01 20:00:00-08:00',
 737: (17)                            '1997-01-02 00:00:00-08:00', '1997-01-02 04:00:00-08:00',
 738: (17)                            '1997-01-02 08:00:00-08:00', '1997-01-02 12:00:00-08:00',
 739: (17)                            '1997-01-02 16:00:00-08:00']
 740: (16)                           ],
 741: (15)                          [datetime.timedelta(minutes=20),
 742: (16)                           ['1997-01-01 00:00:00-08:00', '1997-01-01 00:05:00-08:00',
 743: (17)                            '1997-01-01 00:10:00-08:00', '1997-01-01 00:15:00-08:00',
 744: (17)                            '1997-01-01 00:20:00-08:00']
 745: (16)                           ],
 746: (15)                          [datetime.timedelta(seconds=40),
 747: (16)                           ['1997-01-01 00:00:00-08:00', '1997-01-01 00:00:05-08:00',
 748: (17)                            '1997-01-01 00:00:10-08:00', '1997-01-01 00:00:15-08:00',
 749: (17)                            '1997-01-01 00:00:20-08:00', '1997-01-01 00:00:25-08:00',
 750: (17)                            '1997-01-01 00:00:30-08:00', '1997-01-01 00:00:35-08:00',
 751: (17)                            '1997-01-01 00:00:40-08:00']
 752: (16)                           ]
 753: (15)                          )
 754: (4)                 tz = dateutil.tz.gettz('Canada/Pacific')
 755: (4)                 d1 = datetime.datetime(1997, 1, 1, tzinfo=tz)
 756: (4)                 for t_delta, expected in results:
 757: (8)                     with rc_context({'_internal.classic_mode': False}):
 758: (12)                        d2 = d1 + t_delta
 759: (12)                        locator = _create_auto_date_locator(d1, d2, tz)
 760: (12)                        st = list(map(str, mdates.num2date(locator(), tz=tz)))
 761: (12)                        assert st == expected
 762: (0)             @image_comparison(['date_inverted_limit.png'])
 763: (0)             def test_date_inverted_limit():
 764: (4)                 t0 = datetime.datetime(2009, 1, 20)
 765: (4)                 tf = datetime.datetime(2009, 1, 31)
 766: (4)                 fig, ax = plt.subplots()
 767: (4)                 ax.axhline(t0, color="blue", lw=3)
 768: (4)                 ax.set_ylim(t0 - datetime.timedelta(days=5),
 769: (16)                            tf + datetime.timedelta(days=5))
 770: (4)                 ax.invert_yaxis()
 771: (4)                 fig.subplots_adjust(left=0.25)
 772: (0)             def _test_date2num_dst(date_range, tz_convert):
 773: (4)                 BRUSSELS = dateutil.tz.gettz('Europe/Brussels')
 774: (4)                 UTC = mdates.UTC
 775: (4)                 dtstart = datetime.datetime(2014, 3, 30, 0, 0, tzinfo=UTC)
```

```
(4)                 interval = datetime.timedelta(minutes=33, seconds=45)
(4)                 interval_days = interval.seconds / 86400
(4)                 N = 8
(4)                 dt_utc = date_range(start=dtstart, freq=interval, periods=N)
(4)                 dt_bxl = tz_convert(dt_utc, BRUSSELS)
(4)                 t0 = 735322.0 + mdates.date2num(np.datetime64('0000-12-31'))
(4)                 expected_ordinalf = [t0 + (i * interval_days) for i in range(N)]
(4)                 actual_ordinalf = list(mdates.date2num(dt_bxl))
(4)                 assert actual_ordinalf == expected_ordinalf
(0)             def test_date2num_dst():
(4)                 class dt_tzaware(datetime.datetime):
(8)                     """
(8)                     This bug specifically occurs because of the normalization behavior of
(8)                     pandas Timestamp objects, so in order to replicate it, we need a
(8)                     datetime-like object that applies timezone normalization after
(8)                     subtraction.
(8)                     """
(8)                     def __sub__(self, other):
(12)                        r = super().__sub__(other)
(12)                        tzinfo = getattr(r, 'tzinfo', None)
(12)                        if tzinfo is not None:
(16)                            localizer = getattr(tzinfo, 'normalize', None)
(16)                            if localizer is not None:
(20)                                r = tzinfo.normalize(r)
(12)                        if isinstance(r, datetime.datetime):
(16)                            r = self.mk_tzaware(r)
(12)                        return r
(8)                     def __add__(self, other):
(12)                        return self.mk_tzaware(super().__add__(other))
(8)                     def astimezone(self, tzinfo):
(12)                        dt = super().astimezone(tzinfo)
(12)                        return self.mk_tzaware(dt)
(8)                     @classmethod
(8)                     def mk_tzaware(cls, datetime_obj):
(12)                        kwargs = {}
(12)                        attrs = ('year',
(21)                                 'month',
(21)                                 'day',
(21)                                 'hour',
(21)                                 'minute',
(21)                                 'second',
(21)                                 'microsecond',
(21)                                 'tzinfo')
(12)                        for attr in attrs:
(16)                            val = getattr(datetime_obj, attr, None)
(16)                            if val is not None:
(20)                                kwargs[attr] = val
(12)                        return cls(**kwargs)
(4)                 def date_range(start, freq, periods):
(8)                     dtstart = dt_tzaware.mk_tzaware(start)
(8)                     return [dtstart + (i * freq) for i in range(periods)]
(4)                 def tz_convert(dt_list, tzinfo):
(8)                     return [d.astimezone(tzinfo) for d in dt_list]
(4)                 _test_date2num_dst(date_range, tz_convert)
(0)             def test_date2num_dst_pandas(pd):
(4)                 def tz_convert(*args):
(8)                     return pd.DatetimeIndex.tz_convert(*args).astype(object)
(4)                 _test_date2num_dst(pd.date_range, tz_convert)
(0)             def _test_rrulewrapper(attach_tz, get_tz):
(4)                 SYD = get_tz('Australia/Sydney')
(4)                 dtstart = attach_tz(datetime.datetime(2017, 4, 1, 0), SYD)
(4)                 dtend = attach_tz(datetime.datetime(2017, 4, 4, 0), SYD)
(4)                 rule = mdates.rrulewrapper(freq=dateutil.rrule.DAILY, dtstart=dtstart)
(4)                 act = rule.between(dtstart, dtend)
(4)                 exp = [datetime.datetime(2017, 4, 1, 13, tzinfo=dateutil.tz.tzutc()),
(11)                       datetime.datetime(2017, 4, 2, 14, tzinfo=dateutil.tz.tzutc())]
(4)                 assert act == exp
(0)             def test_rrulewrapper():
(4)                 def attach_tz(dt, zi):
```

```
845: (8)                    return dt.replace(tzinfo=zi)
846: (4)                _test_rrulewrapper(attach_tz, dateutil.tz.gettz)
847: (4)                SYD = dateutil.tz.gettz('Australia/Sydney')
848: (4)                dtstart = datetime.datetime(2017, 4, 1, 0)
849: (4)                dtend = datetime.datetime(2017, 4, 4, 0)
850: (4)                rule = mdates.rrulewrapper(freq=dateutil.rrule.DAILY, dtstart=dtstart,
851: (31)                                           tzinfo=SYD, until=dtend)
852: (4)                assert rule.after(dtstart) == datetime.datetime(2017, 4, 2, 0, 0,
853: (52)                                                                  tzinfo=SYD)
854: (4)                assert rule.before(dtend) == datetime.datetime(2017, 4, 3, 0, 0,
855: (51)                                                                 tzinfo=SYD)
856: (4)                assert rule._base_tzinfo == SYD
857: (4)                assert rule._interval == 1
858: (0)            @pytest.mark.pytz
859: (0)            def test_rrulewrapper_pytz():
860: (4)                pytz = pytest.importorskip("pytz")
861: (4)                def attach_tz(dt, zi):
862: (8)                    return zi.localize(dt)
863: (4)                _test_rrulewrapper(attach_tz, pytz.timezone)
864: (0)            @pytest.mark.pytz
865: (0)            def test_yearlocator_pytz():
866: (4)                pytz = pytest.importorskip("pytz")
867: (4)                tz = pytz.timezone('America/New_York')
868: (4)                x = [tz.localize(datetime.datetime(2010, 1, 1))
869: (9)                     + datetime.timedelta(i) for i in range(2000)]
870: (4)                locator = mdates.AutoDateLocator(interval_multiples=True, tz=tz)
871: (4)                locator.create_dummy_axis()
872: (4)                locator.axis.set_view_interval(mdates.date2num(x[0])-1.0,
873: (35)                                              mdates.date2num(x[-1])+1.0)
874: (4)                t = np.array([733408.208333, 733773.208333, 734138.208333,
875: (18)                             734503.208333, 734869.208333, 735234.208333, 735599.208333])
876: (4)                t = t + mdates.date2num(np.datetime64('0000-12-31'))
877: (4)                np.testing.assert_allclose(t, locator())
878: (4)                expected = ['2009-01-01 00:00:00-05:00',
879: (16)                           '2010-01-01 00:00:00-05:00', '2011-01-01 00:00:00-05:00',
880: (16)                           '2012-01-01 00:00:00-05:00', '2013-01-01 00:00:00-05:00',
881: (16)                           '2014-01-01 00:00:00-05:00', '2015-01-01 00:00:00-05:00']
882: (4)                st = list(map(str, mdates.num2date(locator(), tz=tz)))
883: (4)                assert st == expected
884: (4)                assert np.allclose(locator.tick_values(x[0], x[1]), np.array(
885: (8)                    [14610.20833333, 14610.33333333, 14610.45833333, 14610.58333333,
886: (9)                     14610.70833333, 14610.83333333, 14610.95833333, 14611.08333333,
887: (9)                     14611.20833333]))
888: (4)                assert np.allclose(locator.get_locator(x[1], x[0]).tick_values(x[0],
x[1]),
889: (23)                                  np.array(
890: (8)                    [14610.20833333, 14610.33333333, 14610.45833333, 14610.58333333,
891: (9)                     14610.70833333, 14610.83333333, 14610.95833333, 14611.08333333,
892: (9)                     14611.20833333]))
893: (0)            def test_YearLocator():
894: (4)                def _create_year_locator(date1, date2, **kwargs):
895: (8)                    locator = mdates.YearLocator(**kwargs)
896: (8)                    locator.create_dummy_axis()
897: (8)                    locator.axis.set_view_interval(mdates.date2num(date1),
898: (39)                                                  mdates.date2num(date2))
899: (8)                    return locator
900: (4)                d1 = datetime.datetime(1990, 1, 1)
901: (4)                results = ([datetime.timedelta(weeks=52 * 200),
902: (16)                           {'base': 20, 'month': 1, 'day': 1},
903: (16)                           ['1980-01-01 00:00:00+00:00', '2000-01-01 00:00:00+00:00',
904: (17)                            '2020-01-01 00:00:00+00:00', '2040-01-01 00:00:00+00:00',
905: (17)                            '2060-01-01 00:00:00+00:00', '2080-01-01 00:00:00+00:00',
906: (17)                            '2100-01-01 00:00:00+00:00', '2120-01-01 00:00:00+00:00',
907: (17)                            '2140-01-01 00:00:00+00:00', '2160-01-01 00:00:00+00:00',
908: (17)                            '2180-01-01 00:00:00+00:00', '2200-01-01 00:00:00+00:00']
909: (16)                           ],
910: (15)                          [datetime.timedelta(weeks=52 * 200),
911: (16)                           {'base': 20, 'month': 5, 'day': 16},
912: (16)                           ['1980-05-16 00:00:00+00:00', '2000-05-16 00:00:00+00:00',
```

```
(17)                            '2020-05-16 00:00:00+00:00', '2040-05-16 00:00:00+00:00',
(17)                            '2060-05-16 00:00:00+00:00', '2080-05-16 00:00:00+00:00',
(17)                            '2100-05-16 00:00:00+00:00', '2120-05-16 00:00:00+00:00',
(17)                            '2140-05-16 00:00:00+00:00', '2160-05-16 00:00:00+00:00',
(17)                            '2180-05-16 00:00:00+00:00', '2200-05-16 00:00:00+00:00']
(16)                           ],
(15)                          [datetime.timedelta(weeks=52 * 5),
(16)                           {'base': 20, 'month': 9, 'day': 25},
(16)                           ['1980-09-25 00:00:00+00:00', '2000-09-25 00:00:00+00:00']
(16)                           ],
(15)                          )
(4)                 for delta, arguments, expected in results:
(8)                     d2 = d1 + delta
(8)                     locator = _create_year_locator(d1, d2, **arguments)
(8)                     assert list(map(str, mdates.num2date(locator()))) == expected
(0)             def test_DayLocator():
(4)                 with pytest.raises(ValueError):
(8)                     mdates.DayLocator(interval=-1)
(4)                 with pytest.raises(ValueError):
(8)                     mdates.DayLocator(interval=-1.5)
(4)                 with pytest.raises(ValueError):
(8)                     mdates.DayLocator(interval=0)
(4)                 with pytest.raises(ValueError):
(8)                     mdates.DayLocator(interval=1.3)
(4)                 mdates.DayLocator(interval=1.0)
(0)             def test_tz_utc():
(4)                 dt = datetime.datetime(1970, 1, 1, tzinfo=mdates.UTC)
(4)                 assert dt.tzname() == 'UTC'
(0)             @pytest.mark.parametrize("x, tdelta",
(25)                                    [(1, datetime.timedelta(days=1)),
(26)                                     ([1, 1.5], [datetime.timedelta(days=1),
(38)                                                 datetime.timedelta(days=1.5)])])
(0)             def test_num2timedelta(x, tdelta):
(4)                 dt = mdates.num2timedelta(x)
(4)                 assert dt == tdelta
(0)             def test_datetime64_in_list():
(4)                 dt = [np.datetime64('2000-01-01'), np.datetime64('2001-01-01')]
(4)                 dn = mdates.date2num(dt)
(4)                 t = (np.array([730120.,  730486.]) +
(9)                      mdates.date2num(np.datetime64('0000-12-31')))
(4)                 np.testing.assert_equal(dn, t)
(0)             def test_change_epoch():
(4)                 date = np.datetime64('2000-01-01')
(4)                 mdates._reset_epoch_test_example()
(4)                 mdates.get_epoch()  # Set default.
(4)                 with pytest.raises(RuntimeError):
(8)                     mdates.set_epoch('0000-01-01')
(4)                 mdates._reset_epoch_test_example()
(4)                 mdates.set_epoch('1970-01-01')
(4)                 dt = (date - np.datetime64('1970-01-01')).astype('datetime64[D]')
(4)                 dt = dt.astype('int')
(4)                 np.testing.assert_equal(mdates.date2num(date), float(dt))
(4)                 mdates._reset_epoch_test_example()
(4)                 mdates.set_epoch('0000-12-31')
(4)                 np.testing.assert_equal(mdates.date2num(date), 730120.0)
(4)                 mdates._reset_epoch_test_example()
(4)                 mdates.set_epoch('1970-01-01T01:00:00')
(4)                 np.testing.assert_allclose(mdates.date2num(date), dt - 1./24.)
(4)                 mdates._reset_epoch_test_example()
(4)                 mdates.set_epoch('1970-01-01T00:00:00')
(4)                 np.testing.assert_allclose(
(8)                     mdates.date2num(np.datetime64('1970-01-01T12:00:00')),
(8)                     0.5)
(0)             def test_warn_notintervals():
(4)                 dates = np.arange('2001-01-10', '2001-03-04', dtype='datetime64[D]')
(4)                 locator = mdates.AutoDateLocator(interval_multiples=False)
(4)                 locator.intervald[3] = [2]
(4)                 locator.create_dummy_axis()
(4)                 locator.axis.set_view_interval(mdates.date2num(dates[0]),
```

```
982: (35)                                       mdates.date2num(dates[-1]))
983: (4)                 with pytest.warns(UserWarning, match="AutoDateLocator was unable"):
984: (8)                     locs = locator()
985: (0)             def test_change_converter():
986: (4)                 plt.rcParams['date.converter'] = 'concise'
987: (4)                 dates = np.arange('2020-01-01', '2020-05-01', dtype='datetime64[D]')
988: (4)                 fig, ax = plt.subplots()
989: (4)                 ax.plot(dates, np.arange(len(dates)))
990: (4)                 fig.canvas.draw()
991: (4)                 assert ax.get_xticklabels()[0].get_text() == 'Jan'
992: (4)                 assert ax.get_xticklabels()[1].get_text() == '15'
993: (4)                 plt.rcParams['date.converter'] = 'auto'
994: (4)                 fig, ax = plt.subplots()
995: (4)                 ax.plot(dates, np.arange(len(dates)))
996: (4)                 fig.canvas.draw()
997: (4)                 assert ax.get_xticklabels()[0].get_text() == 'Jan 01 2020'
998: (4)                 assert ax.get_xticklabels()[1].get_text() == 'Jan 15 2020'
999: (4)                 with pytest.raises(ValueError):
1000: (8)                    plt.rcParams['date.converter'] = 'boo'
1001: (0)            def test_change_interval_multiples():
1002: (4)                plt.rcParams['date.interval_multiples'] = False
1003: (4)                dates = np.arange('2020-01-10', '2020-05-01', dtype='datetime64[D]')
1004: (4)                fig, ax = plt.subplots()
1005: (4)                ax.plot(dates, np.arange(len(dates)))
1006: (4)                fig.canvas.draw()
1007: (4)                assert ax.get_xticklabels()[0].get_text() == 'Jan 10 2020'
1008: (4)                assert ax.get_xticklabels()[1].get_text() == 'Jan 24 2020'
1009: (4)                plt.rcParams['date.interval_multiples'] = 'True'
1010: (4)                fig, ax = plt.subplots()
1011: (4)                ax.plot(dates, np.arange(len(dates)))
1012: (4)                fig.canvas.draw()
1013: (4)                assert ax.get_xticklabels()[0].get_text() == 'Jan 15 2020'
1014: (4)                assert ax.get_xticklabels()[1].get_text() == 'Feb 01 2020'
1015: (0)            def test_DateLocator():
1016: (4)                locator = mdates.DateLocator()
1017: (4)                assert locator.nonsingular(0, np.inf) == (0, 1)
1018: (4)                assert locator.nonsingular(0, 1) == (0, 1)
1019: (4)                assert locator.nonsingular(1, 0) == (0, 1)
1020: (4)                assert locator.nonsingular(0, 0) == (-2, 2)
1021: (4)                locator.create_dummy_axis()
1022: (4)                assert locator.datalim_to_dt() == (
1023: (8)                    datetime.datetime(1970, 1, 1, 0, 0, tzinfo=datetime.timezone.utc),
1024: (8)                    datetime.datetime(1970, 1, 2, 0, 0, tzinfo=datetime.timezone.utc))
1025: (4)                assert locator.tz == mdates.UTC
1026: (4)                tz_str = 'Iceland'
1027: (4)                iceland_tz = dateutil.tz.gettz(tz_str)
1028: (4)                assert locator.tz != iceland_tz
1029: (4)                locator.set_tzinfo('Iceland')
1030: (4)                assert locator.tz == iceland_tz
1031: (4)                locator.create_dummy_axis()
1032: (4)                locator.axis.set_data_interval(*mdates.date2num(["2022-01-10",
1033: (53)                                                               "2022-01-08"]))
1034: (4)                assert locator.datalim_to_dt() == (
1035: (8)                    datetime.datetime(2022, 1, 8, 0, 0, tzinfo=iceland_tz),
1036: (8)                    datetime.datetime(2022, 1, 10, 0, 0, tzinfo=iceland_tz))
1037: (4)                plt.rcParams['timezone'] = tz_str
1038: (4)                locator = mdates.DateLocator()
1039: (4)                assert locator.tz == iceland_tz
1040: (4)                with pytest.raises(ValueError, match="Aiceland is not a valid timezone"):
1041: (8)                    mdates.DateLocator(tz="Aiceland")
1042: (4)                with pytest.raises(TypeError,
1043: (23)                                  match="tz must be string or tzinfo subclass."):
1044: (8)                    mdates.DateLocator(tz=1)
1045: (0)            def test_datestr2num():
1046: (4)                assert mdates.datestr2num('2022-01-10') == 19002.0
1047: (4)                dt = datetime.date(year=2022, month=1, day=10)
1048: (4)                assert mdates.datestr2num('2022-01', default=dt) == 19002.0
1049: (4)                assert np.all(mdates.datestr2num(
1050: (8)                    ['2022-01', '2022-02'], default=dt
```

```
1051: (8)                   ) == np.array([19002., 19033.]))
1052: (4)               assert mdates.datestr2num([]).size == 0
1053: (4)               assert mdates.datestr2num([], datetime.date(year=2022,
1054: (48)                                                      month=1, day=10)).size == 0
1055: (0)           @pytest.mark.parametrize('kwarg',
1056: (25)                                  ('formats', 'zero_formats', 'offset_formats'))
1057: (0)           def test_concise_formatter_exceptions(kwarg):
1058: (4)               locator = mdates.AutoDateLocator()
1059: (4)               kwargs = {kwarg: ['', '%Y']}
1060: (4)               match = f"{kwarg} argument must be a list"
1061: (4)               with pytest.raises(ValueError, match=match):
1062: (8)                   mdates.ConciseDateFormatter(locator, **kwargs)
1063: (0)           def test_concise_formatter_call():
1064: (4)               locator = mdates.AutoDateLocator()
1065: (4)               formatter = mdates.ConciseDateFormatter(locator)
1066: (4)               assert formatter(19002.0) == '2022'
1067: (4)               assert formatter.format_data_short(19002.0) == '2022-01-10 00:00:00'
1068: (0)           def test_datetime_masked():
1069: (4)               x = np.array([datetime.datetime(2017, 1, n) for n in range(1, 6)])
1070: (4)               y = np.array([1, 2, 3, 4, 5])
1071: (4)               m = np.ma.masked_greater(y, 0)
1072: (4)               fig, ax = plt.subplots()
1073: (4)               ax.plot(x, m)
1074: (4)               assert ax.get_xlim() == (0, 1)
1075: (0)           @pytest.mark.parametrize('val', (-1000000, 10000000))
1076: (0)           def test_num2date_error(val):
1077: (4)               with pytest.raises(ValueError, match=f"Date ordinal {val} converts"):
1078: (8)                   mdates.num2date(val)
1079: (0)           def test_num2date_roundoff():
1080: (4)               assert mdates.num2date(100000.0000578702) == datetime.datetime(
1081: (8)                   2243, 10, 17, 0, 0, 4, 999980, tzinfo=datetime.timezone.utc)
1082: (4)               assert mdates.num2date(100000.0000578703) == datetime.datetime(
1083: (8)                   2243, 10, 17, 0, 0, 5, tzinfo=datetime.timezone.utc)
1084: (0)           def test_DateFormatter_settz():
1085: (4)               time = mdates.date2num(datetime.datetime(2011, 1, 1, 0, 0,
1086: (45)                                                  tzinfo=mdates.UTC))
1087: (4)               formatter = mdates.DateFormatter('%Y-%b-%d %H:%M')
1088: (4)               assert formatter(time) == '2011-Jan-01 00:00'
1089: (4)               formatter.set_tzinfo('Pacific/Kiritimati')
1090: (4)               assert formatter(time) == '2011-Jan-01 14:00'
```

File 180 - test_datetime.py:

```
1: (0)              import datetime
2: (0)              import numpy as np
3: (0)              import pytest
4: (0)              import matplotlib.pyplot as plt
5: (0)              import matplotlib as mpl
6: (0)              class TestDatetimePlotting:
7: (4)                  @mpl.style.context("default")
8: (4)                  def test_annotate(self):
9: (8)                      mpl.rcParams["date.converter"] = 'concise'
10: (8)                     fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, layout="constrained")
11: (8)                     start_date = datetime.datetime(2023, 10, 1)
12: (8)                     dates = [start_date + datetime.timedelta(days=i) for i in range(31)]
13: (8)                     data = list(range(1, 32))
14: (8)                     test_text = "Test Text"
15: (8)                     ax1.plot(dates, data)
16: (8)                     ax1.annotate(text=test_text, xy=(dates[15], data[15]))
17: (8)                     ax2.plot(data, dates)
18: (8)                     ax2.annotate(text=test_text, xy=(data[5], dates[26]))
19: (8)                     ax3.plot(dates, dates)
20: (8)                     ax3.annotate(text=test_text, xy=(dates[15], dates[3]))
21: (8)                     ax4.plot(dates, dates)
22: (8)                     ax4.annotate(text=test_text, xy=(dates[5], dates[30]),
23: (24)                                 xytext=(dates[1], dates[7]),
arrowprops=dict(facecolor='red'))
```

```
(4)    @pytest.mark.xfail(reason="Test for arrow not written yet")
(4)    @mpl.style.context("default")
(4)    def test_arrow(self):
(8)        fig, ax = plt.subplots()
(8)        ax.arrow(...)
(4)    @mpl.style.context("default")
(4)    def test_axhline(self):
(8)        mpl.rcParams["date.converter"] = 'concise'
(8)        fig, (ax1, ax2, ax3) = plt.subplots(3, 1, layout='constrained')
(8)        ax1.set_ylim(bottom=datetime.datetime(2020, 4, 1),
(21)                    top=datetime.datetime(2020, 8, 1))
(8)        ax2.set_ylim(bottom=np.datetime64('2005-01-01'),
(21)                    top=np.datetime64('2005-04-01'))
(8)        ax3.set_ylim(bottom=datetime.datetime(2023, 9, 1),
(21)                    top=datetime.datetime(2023, 11, 1))
(8)        ax1.axhline(y=datetime.datetime(2020, 6, 3), xmin=0.5, xmax=0.7)
(8)        ax2.axhline(np.datetime64('2005-02-25T03:30'), xmin=0.1, xmax=0.9)
(8)        ax3.axhline(y=datetime.datetime(2023, 10, 24), xmin=0.4, xmax=0.7)
(4)    @mpl.style.context("default")
(4)    def test_axhspan(self):
(8)        mpl.rcParams["date.converter"] = 'concise'
(8)        start_date = datetime.datetime(2023, 1, 1)
(8)        dates = [start_date + datetime.timedelta(days=i) for i in range(31)]
(8)        numbers = list(range(1, 32))
(8)        fig, (ax1, ax2, ax3) = plt.subplots(3, 1,
(44)                                           constrained_layout=True,
(44)                                           figsize=(10, 12))
(8)        ax1.plot(dates, numbers, marker='o', color='blue')
(8)        for i in range(0, 31, 2):
(12)           ax1.axhspan(ymin=i+1, ymax=i+2, facecolor='green', alpha=0.5)
(8)        ax1.set_title('Datetime vs. Number')
(8)        ax1.set_xlabel('Date')
(8)        ax1.set_ylabel('Number')
(8)        ax2.plot(numbers, dates, marker='o', color='blue')
(8)        for i in range(0, 31, 2):
(12)           ymin = start_date + datetime.timedelta(days=i)
(12)           ymax = ymin + datetime.timedelta(days=1)
(12)           ax2.axhspan(ymin=ymin, ymax=ymax, facecolor='green', alpha=0.5)
(8)        ax2.set_title('Number vs. Datetime')
(8)        ax2.set_xlabel('Number')
(8)        ax2.set_ylabel('Date')
(8)        ax3.plot(dates, dates, marker='o', color='blue')
(8)        for i in range(0, 31, 2):
(12)           ymin = start_date + datetime.timedelta(days=i)
(12)           ymax = ymin + datetime.timedelta(days=1)
(12)           ax3.axhspan(ymin=ymin, ymax=ymax, facecolor='green', alpha=0.5)
(8)        ax3.set_title('Datetime vs. Datetime')
(8)        ax3.set_xlabel('Date')
(8)        ax3.set_ylabel('Date')
(4)    @pytest.mark.xfail(reason="Test for axline not written yet")
(4)    @mpl.style.context("default")
(4)    def test_axline(self):
(8)        fig, ax = plt.subplots()
(8)        ax.axline(...)
(4)    @mpl.style.context("default")
(4)    def test_axvline(self):
(8)        mpl.rcParams["date.converter"] = 'concise'
(8)        fig, (ax1, ax2, ax3) = plt.subplots(3, 1, layout='constrained')
(8)        ax1.set_xlim(left=datetime.datetime(2020, 4, 1),
(21)                    right=datetime.datetime(2020, 8, 1))
(8)        ax2.set_xlim(left=np.datetime64('2005-01-01'),
(21)                    right=np.datetime64('2005-04-01'))
(8)        ax3.set_xlim(left=datetime.datetime(2023, 9, 1),
(21)                    right=datetime.datetime(2023, 11, 1))
(8)        ax1.axvline(x=datetime.datetime(2020, 6, 3), ymin=0.5, ymax=0.7)
(8)        ax2.axvline(np.datetime64('2005-02-25T03:30'), ymin=0.1, ymax=0.9)
(8)        ax3.axvline(x=datetime.datetime(2023, 10, 24), ymin=0.4, ymax=0.7)
(4)    @mpl.style.context("default")
(4)    def test_axvspan(self):
```

```
 93: (8)                    mpl.rcParams["date.converter"] = 'concise'
 94: (8)                    start_date = datetime.datetime(2023, 1, 1)
 95: (8)                    dates = [start_date + datetime.timedelta(days=i) for i in range(31)]
 96: (8)                    numbers = list(range(1, 32))
 97: (8)                    fig, (ax1, ax2, ax3) = plt.subplots(3, 1,
 98: (44)                                                       constrained_layout=True,
 99: (44)                                                       figsize=(10, 12))
 100: (8)                   ax1.plot(dates, numbers, marker='o', color='blue')
 101: (8)                   for i in range(0, 31, 2):
 102: (12)                      xmin = start_date + datetime.timedelta(days=i)
 103: (12)                      xmax = xmin + datetime.timedelta(days=1)
 104: (12)                      ax1.axvspan(xmin=xmin, xmax=xmax, facecolor='red', alpha=0.5)
 105: (8)                   ax1.set_title('Datetime vs. Number')
 106: (8)                   ax1.set_xlabel('Date')
 107: (8)                   ax1.set_ylabel('Number')
 108: (8)                   ax2.plot(numbers, dates, marker='o', color='blue')
 109: (8)                   for i in range(0, 31, 2):
 110: (12)                      ax2.axvspan(xmin=i+1, xmax=i+2, facecolor='red', alpha=0.5)
 111: (8)                   ax2.set_title('Number vs. Datetime')
 112: (8)                   ax2.set_xlabel('Number')
 113: (8)                   ax2.set_ylabel('Date')
 114: (8)                   ax3.plot(dates, dates, marker='o', color='blue')
 115: (8)                   for i in range(0, 31, 2):
 116: (12)                      xmin = start_date + datetime.timedelta(days=i)
 117: (12)                      xmax = xmin + datetime.timedelta(days=1)
 118: (12)                      ax3.axvspan(xmin=xmin, xmax=xmax, facecolor='red', alpha=0.5)
 119: (8)                   ax3.set_title('Datetime vs. Datetime')
 120: (8)                   ax3.set_xlabel('Date')
 121: (8)                   ax3.set_ylabel('Date')
 122: (4)               @mpl.style.context("default")
 123: (4)               def test_bar(self):
 124: (8)                   mpl.rcParams["date.converter"] = "concise"
 125: (8)                   fig, (ax1, ax2) = plt.subplots(2, 1, layout="constrained")
 126: (8)                   x_dates = np.array(
 127: (12)                      [
 128: (16)                          datetime.datetime(2020, 6, 30),
 129: (16)                          datetime.datetime(2020, 7, 22),
 130: (16)                          datetime.datetime(2020, 8, 3),
 131: (16)                          datetime.datetime(2020, 9, 14),
 132: (12)                      ],
 133: (12)                      dtype=np.datetime64,
 134: (8)                   )
 135: (8)                   x_ranges = [8800, 2600, 8500, 7400]
 136: (8)                   x = np.datetime64(datetime.datetime(2020, 6, 1))
 137: (8)                   ax1.bar(x_dates, x_ranges, width=np.timedelta64(4, "D"))
 138: (8)                   ax2.bar(np.arange(4), x_dates - x, bottom=x)
 139: (4)               @mpl.style.context("default")
 140: (4)               def test_bar_label(self):
 141: (8)                   date_list = [datetime.datetime(2023, 1, 1) +
 142: (21)                               datetime.timedelta(days=i) for i in range(5)]
 143: (8)                   values = [10, 20, 15, 25, 30]
 144: (8)                   fig, ax = plt.subplots(1, 1, figsize=(10, 8), layout='constrained')
 145: (8)                   bars = ax.bar(date_list, values)
 146: (8)                   ax.bar_label(bars, labels=[f'{val}%' for val in values],
 147: (21)                               label_type='edge', color='black')
 148: (4)               @mpl.style.context("default")
 149: (4)               def test_barbs(self):
 150: (8)                   plt.rcParams["date.converter"] = 'concise'
 151: (8)                   start_date = datetime.datetime(2022, 2, 8, 22)
 152: (8)                   dates = [start_date + datetime.timedelta(hours=i) for i in range(12)]
 153: (8)                   numbers = np.sin(np.linspace(0, 2 * np.pi, 12))
 154: (8)                   u = np.ones(12) * 10
 155: (8)                   v = np.arange(0, 120, 10)
 156: (8)                   fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 6))
 157: (8)                   axes[0].barbs(dates, numbers, u, v, length=7)
 158: (8)                   axes[0].set_title('Datetime vs. Numeric Data')
 159: (8)                   axes[0].set_xlabel('Datetime')
 160: (8)                   axes[0].set_ylabel('Numeric Data')
 161: (8)                   axes[1].barbs(numbers, dates, u, v, length=7)
```

```
162: (8)                    axes[1].set_title('Numeric vs. Datetime Data')
163: (8)                    axes[1].set_xlabel('Numeric Data')
164: (8)                    axes[1].set_ylabel('Datetime')
165: (4)                @mpl.style.context("default")
166: (4)                def test_barh(self):
167: (8)                    mpl.rcParams["date.converter"] = 'concise'
168: (8)                    fig, (ax1, ax2) = plt.subplots(2, 1, layout='constrained')
169: (8)                    birth_date = np.array([datetime.datetime(2020, 4, 10),
170: (31)                                           datetime.datetime(2020, 5, 30),
171: (31)                                           datetime.datetime(2020, 10, 12),
172: (31)                                           datetime.datetime(2020, 11, 15)])
173: (8)                    year_start = datetime.datetime(2020, 1, 1)
174: (8)                    year_end = datetime.datetime(2020, 12, 31)
175: (8)                    age = [21, 53, 20, 24]
176: (8)                    ax1.set_xlabel('Age')
177: (8)                    ax1.set_ylabel('Birth Date')
178: (8)                    ax1.barh(birth_date, width=age, height=datetime.timedelta(days=10))
179: (8)                    ax2.set_xlim(left=year_start, right=year_end)
180: (8)                    ax2.set_xlabel('Birth Date')
181: (8)                    ax2.set_ylabel('Order of Birth Dates')
182: (8)                    ax2.barh(np.arange(4), birth_date-year_start, left=year_start)
183: (4)                @pytest.mark.xfail(reason="Test for boxplot not written yet")
184: (4)                @mpl.style.context("default")
185: (4)                def test_boxplot(self):
186: (8)                    fig, ax = plt.subplots()
187: (8)                    ax.boxplot(...)
188: (4)                @mpl.style.context("default")
189: (4)                def test_broken_barh(self):
190: (8)                    mpl.rcParams["date.converter"] = 'concise'
191: (8)                    fig, ax = plt.subplots()
192: (8)                    ax.broken_barh([(datetime.datetime(2023, 1, 4),
datetime.timedelta(days=2)),
193: (24)                                   (datetime.datetime(2023, 1, 8),
datetime.timedelta(days=3))],
194: (24)                                   (10, 9), facecolors='tab:blue')
195: (8)                    ax.broken_barh([(datetime.datetime(2023, 1, 2),
datetime.timedelta(days=1)),
196: (25)                                    (datetime.datetime(2023, 1, 4),
datetime.timedelta(days=4))],
197: (25)                                    (20, 9), facecolors=('tab:red'))
198: (4)                @mpl.style.context("default")
199: (4)                def test_bxp(self):
200: (8)                    mpl.rcParams["date.converter"] = 'concise'
201: (8)                    fig, ax = plt.subplots()
202: (8)                    data = [{
203: (12)                       "med": datetime.datetime(2020, 1, 15),
204: (12)                       "q1": datetime.datetime(2020, 1, 10),
205: (12)                       "q3": datetime.datetime(2020, 1, 20),
206: (12)                       "whislo": datetime.datetime(2020, 1, 5),
207: (12)                       "whishi": datetime.datetime(2020, 1, 25),
208: (12)                       "fliers": [
209: (16)                           datetime.datetime(2020, 1, 3),
210: (16)                           datetime.datetime(2020, 1, 27)
211: (12)                       ]
212: (8)                    }]
213: (8)                    ax.bxp(data, vert=False)
214: (8)                    ax.xaxis.set_major_formatter(mpl.dates.DateFormatter("%Y-%m-%d"))
215: (8)                    ax.set_title('Box plot with datetime data')
216: (4)                @pytest.mark.xfail(reason="Test for clabel not written yet")
217: (4)                @mpl.style.context("default")
218: (4)                def test_clabel(self):
219: (8)                    fig, ax = plt.subplots()
220: (8)                    ax.clabel(...)
221: (4)                @mpl.style.context("default")
222: (4)                def test_contour(self):
223: (8)                    mpl.rcParams["date.converter"] = "concise"
224: (8)                    range_threshold = 10
225: (8)                    fig, (ax1, ax2, ax3) = plt.subplots(3, 1, layout="constrained")
226: (8)                    x_dates = np.array(
```

```
227: (12)                        [datetime.datetime(2023, 10, delta) for delta in range(1,
range_threshold)]
228: (8)                    )
229: (8)                    y_dates = np.array(
230: (12)                        [datetime.datetime(2023, 10, delta) for delta in range(1,
range_threshold)]
231: (8)                    )
232: (8)                    x_ranges = np.array(range(1, range_threshold))
233: (8)                    y_ranges = np.array(range(1, range_threshold))
234: (8)                    X_dates, Y_dates = np.meshgrid(x_dates, y_dates)
235: (8)                    X_ranges, Y_ranges = np.meshgrid(x_ranges, y_ranges)
236: (8)                    Z_ranges = np.cos(X_ranges / 4) + np.sin(Y_ranges / 4)
237: (8)                    ax1.contour(X_dates, Y_dates, Z_ranges)
238: (8)                    ax2.contour(X_dates, Y_ranges, Z_ranges)
239: (8)                    ax3.contour(X_ranges, Y_dates, Z_ranges)
240: (4)                @mpl.style.context("default")
241: (4)                def test_contourf(self):
242: (8)                    mpl.rcParams["date.converter"] = "concise"
243: (8)                    range_threshold = 10
244: (8)                    fig, (ax1, ax2, ax3) = plt.subplots(3, 1, layout="constrained")
245: (8)                    x_dates = np.array(
246: (12)                        [datetime.datetime(2023, 10, delta) for delta in range(1,
range_threshold)]
247: (8)                    )
248: (8)                    y_dates = np.array(
249: (12)                        [datetime.datetime(2023, 10, delta) for delta in range(1,
range_threshold)]
250: (8)                    )
251: (8)                    x_ranges = np.array(range(1, range_threshold))
252: (8)                    y_ranges = np.array(range(1, range_threshold))
253: (8)                    X_dates, Y_dates = np.meshgrid(x_dates, y_dates)
254: (8)                    X_ranges, Y_ranges = np.meshgrid(x_ranges, y_ranges)
255: (8)                    Z_ranges = np.cos(X_ranges / 4) + np.sin(Y_ranges / 4)
256: (8)                    ax1.contourf(X_dates, Y_dates, Z_ranges)
257: (8)                    ax2.contourf(X_dates, Y_ranges, Z_ranges)
258: (8)                    ax3.contourf(X_ranges, Y_dates, Z_ranges)
259: (4)                @mpl.style.context("default")
260: (4)                def test_errorbar(self):
261: (8)                    mpl.rcParams["date.converter"] = "concise"
262: (8)                    fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, layout="constrained")
263: (8)                    limit = 7
264: (8)                    start_date = datetime.datetime(2023, 1, 1)
265: (8)                    x_dates = np.array([datetime.datetime(2023, 10, d) for d in range(1,
limit)])
266: (8)                    y_dates = np.array([datetime.datetime(2023, 10, d) for d in range(1,
limit)])
267: (8)                    x_date_error = datetime.timedelta(days=1)
268: (8)                    y_date_error = datetime.timedelta(days=1)
269: (8)                    x_values = list(range(1, limit))
270: (8)                    y_values = list(range(1, limit))
271: (8)                    x_value_error = 0.5
272: (8)                    y_value_error = 0.5
273: (8)                    ax1.errorbar(x_dates, y_values,
274: (21)                                 yerr=y_value_error,
275: (21)                                 capsize=10,
276: (21)                                 barsabove=True,
277: (21)                                 label='Data')
278: (8)                    ax2.errorbar(x_values, y_dates,
279: (21)                                 xerr=x_value_error, yerr=y_date_error,
280: (21)                                 errorevery=(1, 2),
281: (21)                                 fmt='-o', label='Data')
282: (8)                    ax3.errorbar(x_dates, y_dates,
283: (21)                                 xerr=x_date_error, yerr=y_date_error,
284: (21)                                 lolims=True, xlolims=True,
285: (21)                                 label='Data')
286: (8)                    ax4.errorbar(x_dates, y_values,
287: (21)                                 xerr=x_date_error, yerr=y_value_error,
288: (21)                                 uplims=True, xuplims=True,
289: (21)                                 label='Data')
```

```
(4)     @mpl.style.context("default")
(4)     def test_eventplot(self):
(8)         mpl.rcParams["date.converter"] = "concise"
(8)         fig, (ax1, ax2, ax3) = plt.subplots(3, 1, layout="constrained")
(8)         x_dates1 = np.array([datetime.datetime(2020, 6, 30),
(29)                              datetime.datetime(2020, 7, 22),
(29)                              datetime.datetime(2020, 8, 3),
(29)                              datetime.datetime(2020, 9, 14),],
(28)                             dtype=np.datetime64,
(28)                             )
(8)         ax1.eventplot(x_dates1)
(8)         np.random.seed(19680801)
(8)         start_date = datetime.datetime(2020, 7, 1)
(8)         end_date = datetime.datetime(2020, 10, 15)
(8)         date_range = end_date - start_date
(8)         dates1 = start_date + np.random.rand(30) * date_range
(8)         dates2 = start_date + np.random.rand(10) * date_range
(8)         dates3 = start_date + np.random.rand(50) * date_range
(8)         colors1 = ['C1', 'C2', 'C3']
(8)         lineoffsets1 = np.array([1, 6, 8])
(8)         linelengths1 = [5, 2, 3]
(8)         ax2.eventplot([dates1, dates2, dates3],
(22)                      colors=colors1,
(22)                      lineoffsets=lineoffsets1,
(22)                      linelengths=linelengths1)
(8)         lineoffsets2 = np.array([
(12)            datetime.datetime(2020, 7, 1),
(12)            datetime.datetime(2020, 7, 15),
(12)            datetime.datetime(2020, 8, 1)
(8)         ], dtype=np.datetime64)
(8)         ax3.eventplot([dates1, dates2, dates3],
(22)                      colors=colors1,
(22)                      lineoffsets=lineoffsets2,
(22)                      linelengths=linelengths1)
(4)     @mpl.style.context("default")
(4)     def test_fill(self):
(8)         mpl.rcParams["date.converter"] = "concise"
(8)         fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, layout="constrained")
(8)         np.random.seed(19680801)
(8)         x_base_date = datetime.datetime(2023, 1, 1)
(8)         x_dates = [x_base_date]
(8)         for _ in range(1, 5):
(12)            x_base_date += datetime.timedelta(days=np.random.randint(1, 5))
(12)            x_dates.append(x_base_date)
(8)         y_base_date = datetime.datetime(2023, 1, 1)
(8)         y_dates = [y_base_date]
(8)         for _ in range(1, 5):
(12)            y_base_date += datetime.timedelta(days=np.random.randint(1, 5))
(12)            y_dates.append(y_base_date)
(8)         x_values = np.random.rand(5) * 5
(8)         y_values = np.random.rand(5) * 5 - 2
(8)         ax1.fill(x_dates, y_values)
(8)         ax2.fill(x_values, y_dates)
(8)         ax3.fill(x_values, y_values)
(8)         ax4.fill(x_dates, y_dates)
(4)     @mpl.style.context("default")
(4)     def test_fill_between(self):
(8)         mpl.rcParams["date.converter"] = "concise"
(8)         np.random.seed(19680801)
(8)         y_base_date = datetime.datetime(2023, 1, 1)
(8)         y_dates1 = [y_base_date]
(8)         for i in range(1, 10):
(12)            y_base_date += datetime.timedelta(days=np.random.randint(1, 5))
(12)            y_dates1.append(y_base_date)
(8)         y_dates2 = [y_base_date]
(8)         for i in range(1, 10):
(12)            y_base_date += datetime.timedelta(days=np.random.randint(1, 5))
(12)            y_dates2.append(y_base_date)
(8)         x_values = np.random.rand(10) * 10
```

```
(8)                 x_values.sort()
(8)                 y_values1 = np.random.rand(10) * 10
(8)                 y_values2 = y_values1 + np.random.rand(10) * 10
(8)                 y_values1.sort()
(8)                 y_values2.sort()
(8)                 x_base_date = datetime.datetime(2023, 1, 1)
(8)                 x_dates = [x_base_date]
(8)                 for i in range(1, 10):
(12)                    x_base_date += datetime.timedelta(days=np.random.randint(1, 10))
(12)                    x_dates.append(x_base_date)
(8)                 fig, (ax1, ax2, ax3) = plt.subplots(3, 1, layout="constrained")
(8)                 ax1.fill_between(x_values, y_dates1, y_dates2)
(8)                 ax2.fill_between(x_dates, y_values1, y_values2)
(8)                 ax3.fill_between(x_dates, y_dates1, y_dates2)
(4)             @mpl.style.context("default")
(4)             def test_fill_betweenx(self):
(8)                 mpl.rcParams["date.converter"] = "concise"
(8)                 np.random.seed(19680801)
(8)                 x_base_date = datetime.datetime(2023, 1, 1)
(8)                 x_dates1 = [x_base_date]
(8)                 for i in range(1, 10):
(12)                    x_base_date += datetime.timedelta(days=np.random.randint(1, 5))
(12)                    x_dates1.append(x_base_date)
(8)                 x_dates2 = [x_base_date]
(8)                 for i in range(1, 10):
(12)                    x_base_date += datetime.timedelta(days=np.random.randint(1, 5))
(12)                    x_dates2.append(x_base_date)
(8)                 y_values = np.random.rand(10) * 10
(8)                 y_values.sort()
(8)                 x_values1 = np.random.rand(10) * 10
(8)                 x_values2 = x_values1 + np.random.rand(10) * 10
(8)                 x_values1.sort()
(8)                 x_values2.sort()
(8)                 y_base_date = datetime.datetime(2023, 1, 1)
(8)                 y_dates = [y_base_date]
(8)                 for i in range(1, 10):
(12)                    y_base_date += datetime.timedelta(days=np.random.randint(1, 10))
(12)                    y_dates.append(y_base_date)
(8)                 fig, (ax1, ax2, ax3) = plt.subplots(1, 3, layout="constrained")
(8)                 ax1.fill_betweenx(y_values, x_dates1, x_dates2)
(8)                 ax2.fill_betweenx(y_dates, x_values1, x_values2)
(8)                 ax3.fill_betweenx(y_dates, x_dates1, x_dates2)
(4)             @pytest.mark.xfail(reason="Test for hexbin not written yet")
(4)             @mpl.style.context("default")
(4)             def test_hexbin(self):
(8)                 fig, ax = plt.subplots()
(8)                 ax.hexbin(...)
(4)             @mpl.style.context("default")
(4)             def test_hist(self):
(8)                 mpl.rcParams["date.converter"] = 'concise'
(8)                 start_date = datetime.datetime(2023, 10, 1)
(8)                 time_delta = datetime.timedelta(days=1)
(8)                 values1 = np.random.randint(1, 10, 30)
(8)                 values2 = np.random.randint(1, 10, 30)
(8)                 values3 = np.random.randint(1, 10, 30)
(8)                 bin_edges = [start_date + i * time_delta for i in range(31)]
(8)                 fig, (ax1, ax2, ax3) = plt.subplots(3, 1, constrained_layout=True)
(8)                 ax1.hist(
(12)                    [start_date + i * time_delta for i in range(30)],
(12)                    bins=10,
(12)                    weights=values1
(8)                 )
(8)                 ax2.hist(
(12)                    [start_date + i * time_delta for i in range(30)],
(12)                    bins=10,
(12)                    weights=values2
(8)                 )
(8)                 ax3.hist(
(12)                    [start_date + i * time_delta for i in range(30)],
```

```
(12)                        bins=10,
(12)                        weights=values3
(8)                     )
(8)                     fig, (ax4, ax5, ax6) = plt.subplots(3, 1, constrained_layout=True)
(8)                     ax4.hist(
(12)                        [start_date + i * time_delta for i in range(30)],
(12)                        bins=bin_edges,
(12)                        weights=values1
(8)                     )
(8)                     ax5.hist(
(12)                        [start_date + i * time_delta for i in range(30)],
(12)                        bins=bin_edges,
(12)                        weights=values2
(8)                     )
(8)                     ax6.hist(
(12)                        [start_date + i * time_delta for i in range(30)],
(12)                        bins=bin_edges,
(12)                        weights=values3
(8)                     )
(4)                 @pytest.mark.xfail(reason="Test for hist2d not written yet")
(4)                 @mpl.style.context("default")
(4)                 def test_hist2d(self):
(8)                     fig, ax = plt.subplots()
(8)                     ax.hist2d(...)
(4)                 @mpl.style.context("default")
(4)                 def test_hlines(self):
(8)                     mpl.rcParams["date.converter"] = 'concise'
(8)                     fig, axs = plt.subplots(2, 4, layout='constrained')
(8)                     dateStrs = ['2023-03-08',
(20)                                 '2023-04-09',
(20)                                 '2023-05-13',
(20)                                 '2023-07-28',
(20)                                 '2023-12-24']
(8)                     dates = [datetime.datetime(2023, m*2, 10) for m in range(1, 6)]
(8)                     date_start = [datetime.datetime(2023, 6, d) for d in range(5, 30, 5)]
(8)                     date_end = [datetime.datetime(2023, 7, d) for d in range(5, 30, 5)]
(8)                     npDates = [np.datetime64(s) for s in dateStrs]
(8)                     axs[0, 0].hlines(y=dates,
(25)                                     xmin=[0.1, 0.2, 0.3, 0.4, 0.5],
(25)                                     xmax=[0.5, 0.6, 0.7, 0.8, 0.9])
(8)                     axs[0, 1].hlines(dates,
(25)                                     xmin=datetime.datetime(2020, 5, 10),
(25)                                     xmax=datetime.datetime(2020, 5, 31))
(8)                     axs[0, 2].hlines(dates,
(25)                                     xmin=date_start,
(25)                                     xmax=date_end)
(8)                     axs[0, 3].hlines(dates,
(25)                                     xmin=0.45,
(25)                                     xmax=0.65)
(8)                     axs[1, 0].hlines(y=npDates,
(25)                                     xmin=[0.5, 0.6, 0.7, 0.8, 0.9],
(25)                                     xmax=[0.1, 0.2, 0.3, 0.4, 0.5])
(8)                     axs[1, 2].hlines(y=npDates,
(25)                                     xmin=date_start,
(25)                                     xmax=date_end)
(8)                     axs[1, 1].hlines(npDates,
(25)                                     xmin=datetime.datetime(2020, 5, 10),
(25)                                     xmax=datetime.datetime(2020, 5, 31))
(8)                     axs[1, 3].hlines(npDates,
(25)                                     xmin=0.45,
(25)                                     xmax=0.65)
(4)                 @mpl.style.context("default")
(4)                 def test_imshow(self):
(8)                     fig, ax = plt.subplots()
(8)                     a = np.diag(range(5))
(8)                     dt_start = datetime.datetime(2010, 11, 1)
(8)                     dt_end = datetime.datetime(2010, 11, 11)
(8)                     extent = (dt_start, dt_end, dt_start, dt_end)
(8)                     ax.imshow(a, extent=extent)
```

```
497: (8)                    ax.tick_params(axis="x", labelrotation=90)
498: (4)                @pytest.mark.xfail(reason="Test for loglog not written yet")
499: (4)                @mpl.style.context("default")
500: (4)                def test_loglog(self):
501: (8)                    fig, ax = plt.subplots()
502: (8)                    ax.loglog(...)
503: (4)                @mpl.style.context("default")
504: (4)                def test_matshow(self):
505: (8)                    a = np.diag(range(5))
506: (8)                    dt_start = datetime.datetime(1980, 4, 15)
507: (8)                    dt_end = datetime.datetime(2020, 11, 11)
508: (8)                    extent = (dt_start, dt_end, dt_start, dt_end)
509: (8)                    fig, ax = plt.subplots()
510: (8)                    ax.matshow(a, extent=extent)
511: (8)                    for label in ax.get_xticklabels():
512: (12)                       label.set_rotation(90)
513: (4)                @pytest.mark.xfail(reason="Test for pcolor not written yet")
514: (4)                @mpl.style.context("default")
515: (4)                def test_pcolor(self):
516: (8)                    fig, ax = plt.subplots()
517: (8)                    ax.pcolor(...)
518: (4)                @pytest.mark.xfail(reason="Test for pcolorfast not written yet")
519: (4)                @mpl.style.context("default")
520: (4)                def test_pcolorfast(self):
521: (8)                    fig, ax = plt.subplots()
522: (8)                    ax.pcolorfast(...)
523: (4)                @pytest.mark.xfail(reason="Test for pcolormesh not written yet")
524: (4)                @mpl.style.context("default")
525: (4)                def test_pcolormesh(self):
526: (8)                    fig, ax = plt.subplots()
527: (8)                    ax.pcolormesh(...)
528: (4)                @mpl.style.context("default")
529: (4)                def test_plot(self):
530: (8)                    mpl.rcParams["date.converter"] = 'concise'
531: (8)                    N = 6
532: (8)                    fig, (ax1, ax2, ax3) = plt.subplots(3, 1, layout='constrained')
533: (8)                    x = np.array([datetime.datetime(2023, 9, n) for n in range(1, N)])
534: (8)                    ax1.plot(x, range(1, N))
535: (8)                    ax2.plot(range(1, N), x)
536: (8)                    ax3.plot(x, x)
537: (4)                @mpl.style.context("default")
538: (4)                def test_plot_date(self):
539: (8)                    mpl.rcParams["date.converter"] = "concise"
540: (8)                    range_threshold = 10
541: (8)                    fig, (ax1, ax2, ax3) = plt.subplots(3, 1, layout="constrained")
542: (8)                    x_dates = np.array(
543: (12)                       [datetime.datetime(2023, 10, delta) for delta in range(1,
range_threshold)]
544: (8)                    )
545: (8)                    y_dates = np.array(
546: (12)                       [datetime.datetime(2023, 10, delta) for delta in range(1,
range_threshold)]
547: (8)                    )
548: (8)                    x_ranges = np.array(range(1, range_threshold))
549: (8)                    y_ranges = np.array(range(1, range_threshold))
550: (8)                    with pytest.warns(mpl.MatplotlibDeprecationWarning):
551: (12)                       ax1.plot_date(x_dates, y_dates)
552: (12)                       ax2.plot_date(x_dates, y_ranges)
553: (12)                       ax3.plot_date(x_ranges, y_dates)
554: (4)                @pytest.mark.xfail(reason="Test for quiver not written yet")
555: (4)                @mpl.style.context("default")
556: (4)                def test_quiver(self):
557: (8)                    fig, ax = plt.subplots()
558: (8)                    ax.quiver(...)
559: (4)                @mpl.style.context("default")
560: (4)                def test_scatter(self):
561: (8)                    mpl.rcParams["date.converter"] = 'concise'
562: (8)                    base = datetime.datetime(2005, 2, 1)
563: (8)                    dates = [base + datetime.timedelta(hours=(2 * i)) for i in range(10)]
```

```
564: (8)                    N = len(dates)
565: (8)                    np.random.seed(19680801)
566: (8)                    y = np.cumsum(np.random.randn(N))
567: (8)                    fig, axs = plt.subplots(3, 1, layout='constrained', figsize=(6, 6))
568: (8)                    axs[0].scatter(dates, y)
569: (8)                    for label in axs[0].get_xticklabels():
570: (12)                       label.set_rotation(40)
571: (12)                       label.set_horizontalalignment('right')
572: (8)                    axs[1].scatter(y, dates)
573: (8)                    axs[2].scatter(dates, dates)
574: (8)                    for label in axs[2].get_xticklabels():
575: (12)                       label.set_rotation(40)
576: (12)                       label.set_horizontalalignment('right')
577: (4)                @pytest.mark.xfail(reason="Test for semilogx not written yet")
578: (4)                @mpl.style.context("default")
579: (4)                def test_semilogx(self):
580: (8)                    fig, ax = plt.subplots()
581: (8)                    ax.semilogx(...)
582: (4)                @pytest.mark.xfail(reason="Test for semilogy not written yet")
583: (4)                @mpl.style.context("default")
584: (4)                def test_semilogy(self):
585: (8)                    fig, ax = plt.subplots()
586: (8)                    ax.semilogy(...)
587: (4)                @mpl.style.context("default")
588: (4)                def test_stackplot(self):
589: (8)                    mpl.rcParams["date.converter"] = 'concise'
590: (8)                    N = 10
591: (8)                    stacked_nums = np.tile(np.arange(1, N), (4, 1))
592: (8)                    dates = np.array([datetime.datetime(2020 + i, 1, 1) for i in range(N -
1)])
593: (8)                    fig, ax = plt.subplots(layout='constrained')
594: (8)                    ax.stackplot(dates, stacked_nums)
595: (4)                @mpl.style.context("default")
596: (4)                def test_stairs(self):
597: (8)                    mpl.rcParams["date.converter"] = 'concise'
598: (8)                    start_date = datetime.datetime(2023, 12, 1)
599: (8)                    time_delta = datetime.timedelta(days=1)
600: (8)                    baseline_date = datetime.datetime(1980, 1, 1)
601: (8)                    bin_edges = [start_date + i * time_delta for i in range(31)]
602: (8)                    edge_int = np.arange(31)
603: (8)                    np.random.seed(123456)
604: (8)                    values1 = np.random.randint(1, 100, 30)
605: (8)                    values2 = [start_date + datetime.timedelta(days=int(i))
606: (19)                              for i in np.random.randint(1, 10000, 30)]
607: (8)                    values3 = [start_date + datetime.timedelta(days=int(i))
608: (19)                              for i in np.random.randint(-10000, 10000, 30)]
609: (8)                    fig, (ax1, ax2, ax3) = plt.subplots(3, 1, constrained_layout=True)
610: (8)                    ax1.stairs(values1, edges=bin_edges)
611: (8)                    ax2.stairs(values2, edges=edge_int, baseline=baseline_date)
612: (8)                    ax3.stairs(values3, edges=bin_edges, baseline=baseline_date)
613: (4)                @mpl.style.context("default")
614: (4)                def test_stem(self):
615: (8)                    mpl.rcParams["date.converter"] = "concise"
616: (8)                    fig, (ax1, ax2, ax3, ax4, ax5, ax6) = plt.subplots(6, 1,
layout="constrained")
617: (8)                    limit_value = 10
618: (8)                    above = datetime.datetime(2023, 9, 18)
619: (8)                    below = datetime.datetime(2023, 11, 18)
620: (8)                    x_ranges = np.arange(1, limit_value)
621: (8)                    y_ranges = np.arange(1, limit_value)
622: (8)                    x_dates = np.array(
623: (12)                       [datetime.datetime(2023, 10, n) for n in range(1, limit_value)]
624: (8)                    )
625: (8)                    y_dates = np.array(
626: (12)                       [datetime.datetime(2023, 10, n) for n in range(1, limit_value)]
627: (8)                    )
628: (8)                    ax1.stem(x_dates, y_dates, bottom=above)
629: (8)                    ax2.stem(x_dates, y_ranges, bottom=5)
630: (8)                    ax3.stem(x_ranges, y_dates, bottom=below)
```

```
(8)                 ax4.stem(x_ranges, y_dates, orientation="horizontal", bottom=above)
(8)                 ax5.stem(x_dates, y_ranges, orientation="horizontal", bottom=5)
(8)                 ax6.stem(x_ranges, y_dates, orientation="horizontal", bottom=below)
(4)             @mpl.style.context("default")
(4)             def test_step(self):
(8)                 mpl.rcParams["date.converter"] = "concise"
(8)                 N = 6
(8)                 fig, (ax1, ax2, ax3) = plt.subplots(3, 1, layout='constrained')
(8)                 x = np.array([datetime.datetime(2023, 9, n) for n in range(1, N)])
(8)                 ax1.step(x, range(1, N))
(8)                 ax2.step(range(1, N), x)
(8)                 ax3.step(x, x)
(4)             @pytest.mark.xfail(reason="Test for streamplot not written yet")
(4)             @mpl.style.context("default")
(4)             def test_streamplot(self):
(8)                 fig, ax = plt.subplots()
(8)                 ax.streamplot(...)
(4)             @mpl.style.context("default")
(4)             def test_text(self):
(8)                 mpl.rcParams["date.converter"] = 'concise'
(8)                 fig, (ax1, ax2, ax3) = plt.subplots(3, 1, layout="constrained")
(8)                 limit_value = 10
(8)                 font_properties = {'family': 'serif', 'size': 12, 'weight': 'bold'}
(8)                 test_date = datetime.datetime(2023, 10, 1)
(8)                 x_data = np.array(range(1, limit_value))
(8)                 y_data = np.array(range(1, limit_value))
(8)                 x_dates = np.array(
(12)                    [datetime.datetime(2023, 10, n) for n in range(1, limit_value)]
(8)                 )
(8)                 y_dates = np.array(
(12)                    [datetime.datetime(2023, 10, n) for n in range(1, limit_value)]
(8)                 )
(8)                 ax1.plot(x_dates, y_data)
(8)                 ax1.text(test_date, 5, "Inserted Text", **font_properties)
(8)                 ax2.plot(x_data, y_dates)
(8)                 ax2.text(7, test_date, "Inserted Text", **font_properties)
(8)                 ax3.plot(x_dates, y_dates)
(8)                 ax3.text(test_date, test_date, "Inserted Text", **font_properties)
(4)             @pytest.mark.xfail(reason="Test for tricontour not written yet")
(4)             @mpl.style.context("default")
(4)             def test_tricontour(self):
(8)                 fig, ax = plt.subplots()
(8)                 ax.tricontour(...)
(4)             @pytest.mark.xfail(reason="Test for tricontourf not written yet")
(4)             @mpl.style.context("default")
(4)             def test_tricontourf(self):
(8)                 fig, ax = plt.subplots()
(8)                 ax.tricontourf(...)
(4)             @pytest.mark.xfail(reason="Test for tripcolor not written yet")
(4)             @mpl.style.context("default")
(4)             def test_tripcolor(self):
(8)                 fig, ax = plt.subplots()
(8)                 ax.tripcolor(...)
(4)             @pytest.mark.xfail(reason="Test for triplot not written yet")
(4)             @mpl.style.context("default")
(4)             def test_triplot(self):
(8)                 fig, ax = plt.subplots()
(8)                 ax.triplot(...)
(4)             @pytest.mark.xfail(reason="Test for violin not written yet")
(4)             @mpl.style.context("default")
(4)             def test_violin(self):
(8)                 fig, ax = plt.subplots()
(8)                 ax.violin(...)
(4)             @pytest.mark.xfail(reason="Test for violinplot not written yet")
(4)             @mpl.style.context("default")
(4)             def test_violinplot(self):
(8)                 fig, ax = plt.subplots()
(8)                 ax.violinplot(...)
(4)             @mpl.style.context("default")
```

```
700: (4)                def test_vlines(self):
701: (8)                    mpl.rcParams["date.converter"] = 'concise'
702: (8)                    fig, (ax1, ax2, ax3) = plt.subplots(3, 1, layout='constrained')
703: (8)                    ax1.set_xlim(left=datetime.datetime(2023, 1, 1),
704: (21)                                 right=datetime.datetime(2023, 6, 30))
705: (8)                    ax1.vlines(x=[datetime.datetime(2023, 2, 10),
706: (22)                                  datetime.datetime(2023, 5, 18),
707: (22)                                  datetime.datetime(2023, 6, 6)],
708: (19)                               ymin=[0, 0.25, 0.5],
709: (19)                               ymax=[0.25, 0.5, 0.75])
710: (8)                    ax2.set_xlim(left=0,
711: (21)                                 right=0.5)
712: (8)                    ax2.vlines(x=[0.3, 0.35],
713: (19)                               ymin=[np.datetime64('2023-03-20'), np.datetime64('2023-03-
31')],
714: (19)                               ymax=[np.datetime64('2023-05-01'), np.datetime64('2023-05-
16')])
715: (8)                    ax3.set_xlim(left=datetime.datetime(2023, 7, 1),
716: (21)                                 right=datetime.datetime(2023, 12, 31))
717: (8)                    ax3.vlines(x=[datetime.datetime(2023, 9, 1), datetime.datetime(2023,
12, 10)],
718: (19)                               ymin=datetime.datetime(2023, 1, 15),
719: (19)                               ymax=datetime.datetime(2023, 1, 30))
```

File 181 - test_determinism.py:

```
1: (0)              """
2: (0)              Test output reproducibility.
3: (0)              """
4: (0)              import os
5: (0)              import sys
6: (0)              import pytest
7: (0)              import matplotlib as mpl
8: (0)              import matplotlib.testing.compare
9: (0)              from matplotlib import pyplot as plt
10: (0)             from matplotlib.testing._markers import needs_ghostscript, needs_usetex
11: (0)             from matplotlib.testing import subprocess_run_for_testing
12: (0)             def _save_figure(objects='mhi', fmt="pdf", usetex=False):
13: (4)                 mpl.use(fmt)
14: (4)                 mpl.rcParams.update({'svg.hashsalt': 'asdf', 'text.usetex': usetex})
15: (4)                 fig = plt.figure()
16: (4)                 if 'm' in objects:
17: (8)                     ax1 = fig.add_subplot(1, 6, 1)
18: (8)                     x = range(10)
19: (8)                     ax1.plot(x, [1] * 10, marker='D')
20: (8)                     ax1.plot(x, [2] * 10, marker='x')
21: (8)                     ax1.plot(x, [3] * 10, marker='^')
22: (8)                     ax1.plot(x, [4] * 10, marker='H')
23: (8)                     ax1.plot(x, [5] * 10, marker='v')
24: (4)                 if 'h' in objects:
25: (8)                     ax2 = fig.add_subplot(1, 6, 2)
26: (8)                     bars = (ax2.bar(range(1, 5), range(1, 5)) +
27: (16)                            ax2.bar(range(1, 5), [6] * 4, bottom=range(1, 5)))
28: (8)                     ax2.set_xticks([1.5, 2.5, 3.5, 4.5])
29: (8)                     patterns = ('-', '+', 'x', '\\', '*', 'o', 'O', '.')
30: (8)                     for bar, pattern in zip(bars, patterns):
31: (12)                        bar.set_hatch(pattern)
32: (4)                 if 'i' in objects:
33: (8)                     A = [[1, 2, 3], [2, 3, 1], [3, 1, 2]]
34: (8)                     fig.add_subplot(1, 6, 3).imshow(A, interpolation='nearest')
35: (8)                     A = [[1, 3, 2], [1, 2, 3], [3, 1, 2]]
36: (8)                     fig.add_subplot(1, 6, 4).imshow(A, interpolation='bilinear')
37: (8)                     A = [[2, 3, 1], [1, 2, 3], [2, 1, 3]]
38: (8)                     fig.add_subplot(1, 6, 5).imshow(A, interpolation='bicubic')
39: (4)                 x = range(5)
40: (4)                 ax = fig.add_subplot(1, 6, 6)
41: (4)                 ax.plot(x, x)
```

```
 42: (4)                ax.set_title('A string $1+2+\\sigma$')
 43: (4)                ax.set_xlabel('A string $1+2+\\sigma$')
 44: (4)                ax.set_ylabel('A string $1+2+\\sigma$')
 45: (4)                stdout = getattr(sys.stdout, 'buffer', sys.stdout)
 46: (4)                fig.savefig(stdout, format=fmt)
 47: (0)            @pytest.mark.parametrize(
 48: (4)                "objects, fmt, usetex", [
 49: (8)                    ("", "pdf", False),
 50: (8)                    ("m", "pdf", False),
 51: (8)                    ("h", "pdf", False),
 52: (8)                    ("i", "pdf", False),
 53: (8)                    ("mhi", "pdf", False),
 54: (8)                    ("mhi", "ps", False),
 55: (8)                    pytest.param(
 56: (12)                       "mhi", "ps", True, marks=[needs_usetex, needs_ghostscript]),
 57: (8)                    ("mhi", "svg", False),
 58: (8)                    pytest.param("mhi", "svg", True, marks=needs_usetex),
 59: (4)                ]
 60: (0)            )
 61: (0)            def test_determinism_check(objects, fmt, usetex):
 62: (4)                """
 63: (4)                Output three times the same graphs and checks that the outputs are exactly
 64: (4)                the same.
 65: (4)                Parameters
 66: (4)                ----------
 67: (4)                objects : str
 68: (8)                    Objects to be included in the test document: 'm' for markers, 'h' for
 69: (8)                    hatch patterns, 'i' for images.
 70: (4)                fmt : {"pdf", "ps", "svg"}
 71: (8)                    Output format.
 72: (4)                """
 73: (4)                plots = [
 74: (8)                    subprocess_run_for_testing(
 75: (12)                       [sys.executable, "-R", "-c",
 76: (13)                        f"from matplotlib.tests.test_determinism import _save_figure;"
 77: (13)                        f"_save_figure({objects!r}, {fmt!r}, {usetex})"],
 78: (12)                       env={**os.environ, "SOURCE_DATE_EPOCH": "946684800",
 79: (17)                            "MPLBACKEND": "Agg"},
 80: (12)                       text=False, capture_output=True, check=True).stdout
 81: (8)                    for _ in range(3)
 82: (4)                ]
 83: (4)                for p in plots[1:]:
 84: (8)                    if fmt == "ps" and usetex:
 85: (12)                       if p != plots[0]:
 86: (16)                           pytest.skip("failed, maybe due to ghostscript timestamps")
 87: (8)                    else:
 88: (12)                       assert p == plots[0]
 89: (0)            @pytest.mark.parametrize(
 90: (4)                "fmt, string", [
 91: (8)                    ("pdf", b"/CreationDate (D:20000101000000Z)"),
 92: (8)                    ("ps", b"%%CreationDate: Sat Jan 01 00:00:00 2000"),
 93: (4)                ]
 94: (0)            )
 95: (0)            def test_determinism_source_date_epoch(fmt, string):
 96: (4)                """
 97: (4)                Test SOURCE_DATE_EPOCH support. Output a document with the environment
 98: (4)                variable SOURCE_DATE_EPOCH set to 2000-01-01 00:00 UTC and check that the
 99: (4)                document contains the timestamp that corresponds to this date (given as an
100: (4)                argument).
101: (4)                Parameters
102: (4)                ----------
103: (4)                fmt : {"pdf", "ps", "svg"}
104: (8)                    Output format.
105: (4)                string : bytes
106: (8)                    Timestamp string for 2000-01-01 00:00 UTC.
107: (4)                """
108: (4)                buf = subprocess_run_for_testing(
109: (8)                    [sys.executable, "-R", "-c",
110: (9)                     f"from matplotlib.tests.test_determinism import _save_figure; "
```

```
111: (9)                     f"_save_figure('', {fmt!r})"],
112: (8)                    env={**os.environ, "SOURCE_DATE_EPOCH": "946684800",
113: (13)                        "MPLBACKEND": "Agg"}, capture_output=True, text=False,
check=True).stdout
114: (4)                assert string in buf

----------------------------------------

File 182 - test_doc.py:

1: (0)              import pytest
2: (0)              def test_sphinx_gallery_example_header():
3: (4)                  """
4: (4)                  We have copied EXAMPLE_HEADER and modified it to include meta keywords.
5: (4)                  This test monitors that the version we have copied is still the same as
6: (4)                  the EXAMPLE_HEADER in sphinx-gallery. If sphinx-gallery changes its
7: (4)                  EXAMPLE_HEADER, this test will start to fail. In that case, please update
8: (4)                  the monkey-patching of EXAMPLE_HEADER in conf.py.
9: (4)                  """
10: (4)                 pytest.importorskip('sphinx_gallery', minversion='0.16.0')
11: (4)                 from sphinx_gallery import gen_rst
12: (4)                 EXAMPLE_HEADER = """
13: (0)             .. DO NOT EDIT.
14: (0)             .. THIS FILE WAS AUTOMATICALLY GENERATED BY SPHINX-GALLERY.
15: (0)             .. TO MAKE CHANGES, EDIT THE SOURCE PYTHON FILE:
16: (0)             .. "{0}"
17: (0)             .. LINE NUMBERS ARE GIVEN BELOW.
18: (0)             .. only:: html
19: (4)                 .. note::
20: (8)                     :class: sphx-glr-download-link-note
21: (8)                     :ref:`Go to the end <sphx_glr_download_{1}>`
22: (8)                     to download the full example code.{2}
23: (0)             .. rst-class:: sphx-glr-example-title
24: (0)             .. _sphx_glr_{1}:
25: (0)             """
26: (4)                 assert gen_rst.EXAMPLE_HEADER == EXAMPLE_HEADER

----------------------------------------

File 183 - test_dviread.py:

1: (0)              import json
2: (0)              from pathlib import Path
3: (0)              import shutil
4: (0)              import matplotlib.dviread as dr
5: (0)              import pytest
6: (0)              def test_PsfontsMap(monkeypatch):
7: (4)                  monkeypatch.setattr(dr, 'find_tex_file', lambda x: x.decode())
8: (4)                  filename = str(Path(__file__).parent / 'baseline_images/dviread/test.map')
9: (4)                  fontmap = dr.PsfontsMap(filename)
10: (4)                 for n in [1, 2, 3, 4, 5]:
11: (8)                     key = b'TeXfont%d' % n
12: (8)                     entry = fontmap[key]
13: (8)                     assert entry.texname == key
14: (8)                     assert entry.psname == b'PSfont%d' % n
15: (8)                     if n not in [3, 5]:
16: (12)                        assert entry.encoding == 'font%d.enc' % n
17: (8)                     elif n == 3:
18: (12)                        assert entry.encoding == 'enc3.foo'
19: (8)                     if n not in [1, 5]:
20: (12)                        assert entry.filename == 'font%d.pfa' % n
21: (8)                     else:
22: (12)                        assert entry.filename == 'font%d.pfb' % n
23: (8)                     if n == 4:
24: (12)                        assert entry.effects == {'slant': -0.1, 'extend': 1.2}
25: (8)                     else:
26: (12)                        assert entry.effects == {}
27: (4)                 entry = fontmap[b'TeXfont6']
28: (4)                 assert entry.filename is None
```

```
29: (4)                 assert entry.encoding is None
30: (4)                 entry = fontmap[b'TeXfont7']
31: (4)                 assert entry.filename is None
32: (4)                 assert entry.encoding == 'font7.enc'
33: (4)                 entry = fontmap[b'TeXfont8']
34: (4)                 assert entry.filename == 'font8.pfb'
35: (4)                 assert entry.encoding is None
36: (4)                 entry = fontmap[b'TeXfont9']
37: (4)                 assert entry.psname == b'TeXfont9'
38: (4)                 assert entry.filename == '/absolute/font9.pfb'
39: (4)                 entry = fontmap[b'TeXfontA']
40: (4)                 assert entry.psname == b'PSfontA1'
41: (4)                 entry = fontmap[b'TeXfontB']
42: (4)                 assert entry.psname == b'PSfontB6'
43: (4)                 entry = fontmap[b'TeXfontC']
44: (4)                 assert entry.psname == b'PSfontC3'
45: (4)                 with pytest.raises(LookupError, match='no-such-font'):
46: (8)                     fontmap[b'no-such-font']
47: (4)                 with pytest.raises(LookupError, match='%'):
48: (8)                     fontmap[b'%']
49: (0)             @pytest.mark.skipif(shutil.which("kpsewhich") is None,
50: (20)                                reason="kpsewhich is not available")
51: (0)             def test_dviread():
52: (4)                 dirpath = Path(__file__).parent / 'baseline_images/dviread'
53: (4)                 with (dirpath / 'test.json').open() as f:
54: (8)                     correct = json.load(f)
55: (4)                 with dr.Dvi(str(dirpath / 'test.dvi'), None) as dvi:
56: (8)                     data = [{'text': [[t.x, t.y,
57: (27)                                          chr(t.glyph),
58: (27)                                          t.font.texname.decode('ascii'),
59: (27)                                          round(t.font.size, 2)]
60: (26)                                         for t in page.text],
61: (17)                              'boxes': [[b.x, b.y, b.height, b.width] for b in page.boxes]}
62: (16)                             for page in dvi]
63: (4)                 assert data == correct
```

File 184 - test_figure.py:

```
1: (0)              import copy
2: (0)              from datetime import datetime
3: (0)              import io
4: (0)              import pickle
5: (0)              import platform
6: (0)              from threading import Timer
7: (0)              from types import SimpleNamespace
8: (0)              import warnings
9: (0)              import numpy as np
10: (0)             import pytest
11: (0)             from PIL import Image
12: (0)             import matplotlib as mpl
13: (0)             from matplotlib import gridspec
14: (0)             from matplotlib.testing.decorators import image_comparison,
check_figures_equal
15: (0)             from matplotlib.axes import Axes
16: (0)             from matplotlib.backend_bases import KeyEvent, MouseEvent
17: (0)             from matplotlib.figure import Figure, FigureBase
18: (0)             from matplotlib.layout_engine import (ConstrainedLayoutEngine,
19: (38)                                                  TightLayoutEngine,
20: (38)                                                  PlaceHolderLayoutEngine)
21: (0)             from matplotlib.ticker import AutoMinorLocator, FixedFormatter,
ScalarFormatter
22: (0)             import matplotlib.pyplot as plt
23: (0)             import matplotlib.dates as mdates
24: (0)             @image_comparison(['figure_align_labels'], extensions=['png', 'svg'],
25: (18)                              tol=0 if platform.machine() == 'x86_64' else 0.01)
26: (0)             def test_align_labels():
27: (4)                 fig = plt.figure(layout='tight')
```

```
(4)                 gs = gridspec.GridSpec(3, 3)
(4)                 ax = fig.add_subplot(gs[0, :2])
(4)                 ax.plot(np.arange(0, 1e6, 1000))
(4)                 ax.set_ylabel('Ylabel0 0')
(4)                 ax = fig.add_subplot(gs[0, -1])
(4)                 ax.plot(np.arange(0, 1e4, 100))
(4)                 for i in range(3):
(8)                     ax = fig.add_subplot(gs[1, i])
(8)                     ax.set_ylabel('YLabel1 %d' % i)
(8)                     ax.set_xlabel('XLabel1 %d' % i)
(8)                     if i in [0, 2]:
(12)                        ax.xaxis.set_label_position("top")
(12)                        ax.xaxis.tick_top()
(8)                     if i == 0:
(12)                        for tick in ax.get_xticklabels():
(16)                            tick.set_rotation(90)
(8)                     if i == 2:
(12)                        ax.yaxis.set_label_position("right")
(12)                        ax.yaxis.tick_right()
(4)                 for i in range(3):
(8)                     ax = fig.add_subplot(gs[2, i])
(8)                     ax.set_xlabel(f'XLabel2 {i}')
(8)                     ax.set_ylabel(f'YLabel2 {i}')
(8)                     if i == 2:
(12)                        ax.plot(np.arange(0, 1e4, 10))
(12)                        ax.yaxis.set_label_position("right")
(12)                        ax.yaxis.tick_right()
(12)                        for tick in ax.get_xticklabels():
(16)                            tick.set_rotation(90)
(4)                 fig.align_labels()
(0)             @image_comparison(['figure_align_titles_tight.png',
(19)                               'figure_align_titles_constrained.png'],
(18)                              tol=0 if platform.machine() == 'x86_64' else 0.022,
(18)                              style='mpl20')
(0)             def test_align_titles():
(4)                 for layout in ['tight', 'constrained']:
(8)                     fig, axs = plt.subplots(1, 2, layout=layout, width_ratios=[2, 1])
(8)                     ax = axs[0]
(8)                     ax.plot(np.arange(0, 1e6, 1000))
(8)                     ax.set_title('Title0 left', loc='left')
(8)                     ax.set_title('Title0 center', loc='center')
(8)                     ax.set_title('Title0 right', loc='right')
(8)                     ax = axs[1]
(8)                     ax.plot(np.arange(0, 1e4, 100))
(8)                     ax.set_title('Title1')
(8)                     ax.set_xlabel('Xlabel0')
(8)                     ax.xaxis.set_label_position("top")
(8)                     ax.xaxis.tick_top()
(8)                     for tick in ax.get_xticklabels():
(12)                        tick.set_rotation(90)
(8)                     fig.align_titles()
(0)             def test_align_labels_stray_axes():
(4)                 fig, axs = plt.subplots(2, 2)
(4)                 for nn, ax in enumerate(axs.flat):
(8)                     ax.set_xlabel('Boo')
(8)                     ax.set_xlabel('Who')
(8)                     ax.plot(np.arange(4)**nn, np.arange(4)**nn)
(4)                 fig.align_ylabels()
(4)                 fig.align_xlabels()
(4)                 fig.draw_without_rendering()
(4)                 xn = np.zeros(4)
(4)                 yn = np.zeros(4)
(4)                 for nn, ax in enumerate(axs.flat):
(8)                     yn[nn] = ax.xaxis.label.get_position()[1]
(8)                     xn[nn] = ax.yaxis.label.get_position()[0]
(4)                 np.testing.assert_allclose(xn[:2], xn[2:])
(4)                 np.testing.assert_allclose(yn[::2], yn[1::2])
(4)                 fig, axs = plt.subplots(2, 2, constrained_layout=True)
(4)                 for nn, ax in enumerate(axs.flat):
```

```
97: (8)                     ax.set_xlabel('Boo')
98: (8)                     ax.set_xlabel('Who')
99: (8)                     pc = ax.pcolormesh(np.random.randn(10, 10))
100: (4)                fig.colorbar(pc, ax=ax)
101: (4)                fig.align_ylabels()
102: (4)                fig.align_xlabels()
103: (4)                fig.draw_without_rendering()
104: (4)                xn = np.zeros(4)
105: (4)                yn = np.zeros(4)
106: (4)                for nn, ax in enumerate(axs.flat):
107: (8)                    yn[nn] = ax.xaxis.label.get_position()[1]
108: (8)                    xn[nn] = ax.yaxis.label.get_position()[0]
109: (4)                np.testing.assert_allclose(xn[:2], xn[2:])
110: (4)                np.testing.assert_allclose(yn[::2], yn[1::2])
111: (0)            def test_figure_label():
112: (4)                plt.close('all')
113: (4)                fig_today = plt.figure('today')
114: (4)                plt.figure(3)
115: (4)                plt.figure('tomorrow')
116: (4)                plt.figure()
117: (4)                plt.figure(0)
118: (4)                plt.figure(1)
119: (4)                plt.figure(3)
120: (4)                assert plt.get_fignums() == [0, 1, 3, 4, 5]
121: (4)                assert plt.get_figlabels() == ['', 'today', '', 'tomorrow', '']
122: (4)                plt.close(10)
123: (4)                plt.close()
124: (4)                plt.close(5)
125: (4)                plt.close('tomorrow')
126: (4)                assert plt.get_fignums() == [0, 1]
127: (4)                assert plt.get_figlabels() == ['', 'today']
128: (4)                plt.figure(fig_today)
129: (4)                assert plt.gcf() == fig_today
130: (4)                with pytest.raises(ValueError):
131: (8)                    plt.figure(Figure())
132: (0)            def test_fignum_exists():
133: (4)                plt.figure('one')
134: (4)                plt.figure(2)
135: (4)                plt.figure('three')
136: (4)                plt.figure()
137: (4)                assert plt.fignum_exists('one')
138: (4)                assert plt.fignum_exists(2)
139: (4)                assert plt.fignum_exists('three')
140: (4)                assert plt.fignum_exists(4)
141: (4)                plt.close('one')
142: (4)                plt.close(4)
143: (4)                assert not plt.fignum_exists('one')
144: (4)                assert not plt.fignum_exists(4)
145: (0)            def test_clf_keyword():
146: (4)                text1 = 'A fancy plot'
147: (4)                text2 = 'Really fancy!'
148: (4)                fig0 = plt.figure(num=1)
149: (4)                fig0.suptitle(text1)
150: (4)                assert [t.get_text() for t in fig0.texts] == [text1]
151: (4)                fig1 = plt.figure(num=1, clear=False)
152: (4)                fig1.text(0.5, 0.5, text2)
153: (4)                assert fig0 is fig1
154: (4)                assert [t.get_text() for t in fig1.texts] == [text1, text2]
155: (4)                fig2, ax2 = plt.subplots(2, 1, num=1, clear=True)
156: (4)                assert fig0 is fig2
157: (4)                assert [t.get_text() for t in fig2.texts] == []
158: (0)            @image_comparison(['figure_today'],
159: (18)                             tol=0.015 if platform.machine() == 'arm64' else 0)
160: (0)            def test_figure():
161: (4)                fig = plt.figure('today')
162: (4)                ax = fig.add_subplot()
163: (4)                ax.set_title(fig.get_label())
164: (4)                ax.plot(np.arange(5))
165: (4)                plt.figure('tomorrow')
```

```
166: (4)                plt.plot([0, 1], [1, 0], 'r')
167: (4)                plt.figure('today')
168: (4)                plt.close('tomorrow')
169: (0)            @image_comparison(['figure_legend'])
170: (0)            def test_figure_legend():
171: (4)                fig, axs = plt.subplots(2)
172: (4)                axs[0].plot([0, 1], [1, 0], label='x', color='g')
173: (4)                axs[0].plot([0, 1], [0, 1], label='y', color='r')
174: (4)                axs[0].plot([0, 1], [0.5, 0.5], label='y', color='k')
175: (4)                axs[1].plot([0, 1], [1, 0], label='_y', color='r')
176: (4)                axs[1].plot([0, 1], [0, 1], label='z', color='b')
177: (4)                fig.legend()
178: (0)            def test_gca():
179: (4)                fig = plt.figure()
180: (4)                ax0 = fig.add_axes([0, 0, 1, 1])
181: (4)                assert fig.gca() is ax0
182: (4)                ax1 = fig.add_subplot(111)
183: (4)                assert fig.gca() is ax1
184: (4)                fig.add_axes(ax0)
185: (4)                assert fig.axes == [ax0, ax1]
186: (4)                assert fig.gca() is ax0
187: (4)                fig.sca(ax0)
188: (4)                assert fig.axes == [ax0, ax1]
189: (4)                fig.add_subplot(ax1)
190: (4)                assert fig.axes == [ax0, ax1]
191: (4)                assert fig.gca() is ax1
192: (0)            def test_add_subplot_subclass():
193: (4)                fig = plt.figure()
194: (4)                fig.add_subplot(axes_class=Axes)
195: (4)                with pytest.raises(ValueError):
196: (8)                    fig.add_subplot(axes_class=Axes, projection="3d")
197: (4)                with pytest.raises(ValueError):
198: (8)                    fig.add_subplot(axes_class=Axes, polar=True)
199: (4)                with pytest.raises(ValueError):
200: (8)                    fig.add_subplot(projection="3d", polar=True)
201: (4)                with pytest.raises(TypeError):
202: (8)                    fig.add_subplot(projection=42)
203: (0)            def test_add_subplot_invalid():
204: (4)                fig = plt.figure()
205: (4)                with pytest.raises(ValueError,
206: (23)                                   match='Number of columns must be a positive integer'):
207: (8)                    fig.add_subplot(2, 0, 1)
208: (4)                with pytest.raises(ValueError,
209: (23)                                   match='Number of rows must be a positive integer'):
210: (8)                    fig.add_subplot(0, 2, 1)
211: (4)                with pytest.raises(ValueError, match='num must be an integer with '
212: (41)                                                '1 <= num <= 4'):
213: (8)                    fig.add_subplot(2, 2, 0)
214: (4)                with pytest.raises(ValueError, match='num must be an integer with '
215: (41)                                                '1 <= num <= 4'):
216: (8)                    fig.add_subplot(2, 2, 5)
217: (4)                with pytest.raises(ValueError, match='num must be an integer with '
218: (41)                                                '1 <= num <= 4'):
219: (8)                    fig.add_subplot(2, 2, 0.5)
220: (4)                with pytest.raises(ValueError, match='must be a three-digit integer'):
221: (8)                    fig.add_subplot(42)
222: (4)                with pytest.raises(ValueError, match='must be a three-digit integer'):
223: (8)                    fig.add_subplot(1000)
224: (4)                with pytest.raises(TypeError, match='takes 1 or 3 positional arguments '
225: (40)                                               'but 2 were given'):
226: (8)                    fig.add_subplot(2, 2)
227: (4)                with pytest.raises(TypeError, match='takes 1 or 3 positional arguments '
228: (40)                                               'but 4 were given'):
229: (8)                    fig.add_subplot(1, 2, 3, 4)
230: (4)                with pytest.raises(ValueError,
231: (23)                                   match="Number of rows must be a positive integer, "
232: (29)                                         "not '2'"):
233: (8)                    fig.add_subplot('2', 2, 1)
234: (4)                with pytest.raises(ValueError,
```

```
(23)                                match='Number of columns must be a positive integer, '
(29)                                      'not 2.0'):
(8)                 fig.add_subplot(2, 2.0, 1)
(4)             _, ax = plt.subplots()
(4)             with pytest.raises(ValueError,
(23)                                match='The Axes must have been created in the '
(29)                                      'present figure'):
(8)                 fig.add_subplot(ax)
(0)         @image_comparison(['figure_suptitle'])
(0)         def test_suptitle():
(4)             fig, _ = plt.subplots()
(4)             fig.suptitle('hello', color='r')
(4)             fig.suptitle('title', color='g', rotation=30)
(0)         def test_suptitle_fontproperties():
(4)             fig, ax = plt.subplots()
(4)             fps = mpl.font_manager.FontProperties(size='large', weight='bold')
(4)             txt = fig.suptitle('fontprops title', fontproperties=fps)
(4)             assert txt.get_fontsize() == fps.get_size_in_points()
(4)             assert txt.get_weight() == fps.get_weight()
(0)         def test_suptitle_subfigures():
(4)             fig = plt.figure(figsize=(4, 3))
(4)             sf1, sf2 = fig.subfigures(1, 2)
(4)             sf2.set_facecolor('white')
(4)             sf1.subplots()
(4)             sf2.subplots()
(4)             fig.suptitle("This is a visible suptitle.")
(4)             assert sf1.get_facecolor() == (0.0, 0.0, 0.0, 0.0)
(4)             assert sf2.get_facecolor() == (1.0, 1.0, 1.0, 1.0)
(0)         def test_get_suptitle_supxlabel_supylabel():
(4)             fig, ax = plt.subplots()
(4)             assert fig.get_suptitle() == ""
(4)             assert fig.get_supxlabel() == ""
(4)             assert fig.get_supylabel() == ""
(4)             fig.suptitle('suptitle')
(4)             assert fig.get_suptitle() == 'suptitle'
(4)             fig.supxlabel('supxlabel')
(4)             assert fig.get_supxlabel() == 'supxlabel'
(4)             fig.supylabel('supylabel')
(4)             assert fig.get_supylabel() == 'supylabel'
(0)         @image_comparison(['alpha_background'],
(18)                          extensions=['png', 'svg'],
(18)                          savefig_kwarg={'facecolor': (0, 1, 0.4),
(33)                                         'edgecolor': 'none'})
(0)         def test_alpha():
(4)             fig = plt.figure(figsize=[2, 1])
(4)             fig.set_facecolor((0, 1, 0.4))
(4)             fig.patch.set_alpha(0.4)
(4)             fig.patches.append(mpl.patches.CirclePolygon(
(8)                 [20, 20], radius=15, alpha=0.6, facecolor='red'))
(0)         def test_too_many_figures():
(4)             with pytest.warns(RuntimeWarning):
(8)                 for i in range(mpl.rcParams['figure.max_open_warning'] + 1):
(12)                    plt.figure()
(0)         def test_iterability_axes_argument():
(4)             class MyAxes(Axes):
(8)                 def __init__(self, *args, myclass=None, **kwargs):
(12)                    Axes.__init__(self, *args, **kwargs)
(4)             class MyClass:
(8)                 def __getitem__(self, item):
(12)                    if item != 'a':
(16)                        raise ValueError("item should be a")
(8)                 def _as_mpl_axes(self):
(12)                    return MyAxes, {'myclass': self}
(4)             fig = plt.figure()
(4)             fig.add_subplot(1, 1, 1, projection=MyClass())
(4)             plt.close(fig)
(0)         def test_set_fig_size():
(4)             fig = plt.figure()
(4)             fig.set_figwidth(5)
```

```
(4)                 assert fig.get_figwidth() == 5
(4)                 fig.set_figheight(1)
(4)                 assert fig.get_figheight() == 1
(4)                 fig.set_size_inches(2, 4)
(4)                 assert fig.get_figwidth() == 2
(4)                 assert fig.get_figheight() == 4
(4)                 fig.set_size_inches((1, 3))
(4)                 assert fig.get_figwidth() == 1
(4)                 assert fig.get_figheight() == 3
(0)             def test_axes_remove():
(4)                 fig, axs = plt.subplots(2, 2)
(4)                 axs[-1, -1].remove()
(4)                 for ax in axs.ravel()[:-1]:
(8)                     assert ax in fig.axes
(4)                 assert axs[-1, -1] not in fig.axes
(4)                 assert len(fig.axes) == 3
(0)             def test_figaspect():
(4)                 w, h = plt.figaspect(np.float64(2) / np.float64(1))
(4)                 assert h / w == 2
(4)                 w, h = plt.figaspect(2)
(4)                 assert h / w == 2
(4)                 w, h = plt.figaspect(np.zeros((1, 2)))
(4)                 assert h / w == 0.5
(4)                 w, h = plt.figaspect(np.zeros((2, 2)))
(4)                 assert h / w == 1
(0)             @pytest.mark.parametrize('which', ['both', 'major', 'minor'])
(0)             def test_autofmt_xdate(which):
(4)                 date = ['3 Jan 2013', '4 Jan 2013', '5 Jan 2013', '6 Jan 2013',
(12)                        '7 Jan 2013', '8 Jan 2013', '9 Jan 2013', '10 Jan 2013',
(12)                        '11 Jan 2013', '12 Jan 2013', '13 Jan 2013', '14 Jan 2013']
(4)                 time = ['16:44:00', '16:45:00', '16:46:00', '16:47:00', '16:48:00',
(12)                        '16:49:00', '16:51:00', '16:52:00', '16:53:00', '16:55:00',
(12)                        '16:56:00', '16:57:00']
(4)                 angle = 60
(4)                 minors = [1, 2, 3, 4, 5, 6, 7]
(4)                 x = mdates.datestr2num(date)
(4)                 y = mdates.datestr2num(time)
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(x, y)
(4)                 ax.yaxis_date()
(4)                 ax.xaxis_date()
(4)                 ax.xaxis.set_minor_locator(AutoMinorLocator(2))
(4)                 with warnings.catch_warnings():
(8)                     warnings.filterwarnings(
(12)                        'ignore',
(12)                        'FixedFormatter should only be used together with FixedLocator')
(8)                     ax.xaxis.set_minor_formatter(FixedFormatter(minors))
(4)                 fig.autofmt_xdate(0.2, angle, 'right', which)
(4)                 if which in ('both', 'major'):
(8)                     for label in fig.axes[0].get_xticklabels(False, 'major'):
(12)                        assert int(label.get_rotation()) == angle
(4)                 if which in ('both', 'minor'):
(8)                     for label in fig.axes[0].get_xticklabels(True, 'minor'):
(12)                        assert int(label.get_rotation()) == angle
(0)             @mpl.style.context('default')
(0)             def test_change_dpi():
(4)                 fig = plt.figure(figsize=(4, 4))
(4)                 fig.draw_without_rendering()
(4)                 assert fig.canvas.renderer.height == 400
(4)                 assert fig.canvas.renderer.width == 400
(4)                 fig.dpi = 50
(4)                 fig.draw_without_rendering()
(4)                 assert fig.canvas.renderer.height == 200
(4)                 assert fig.canvas.renderer.width == 200
(0)             @pytest.mark.parametrize('width, height', [
(4)                 (1, np.nan),
(4)                 (-1, 1),
(4)                 (np.inf, 1)
(0)             ])
```

```
(0)            def test_invalid_figure_size(width, height):
(4)                with pytest.raises(ValueError):
(8)                    plt.figure(figsize=(width, height))
(4)                fig = plt.figure()
(4)                with pytest.raises(ValueError):
(8)                    fig.set_size_inches(width, height)
(0)            def test_invalid_figure_add_axes():
(4)                fig = plt.figure()
(4)                with pytest.raises(TypeError,
(23)                               match="missing 1 required positional argument:
'rect'"):
(8)                    fig.add_axes()
(4)                with pytest.raises(ValueError):
(8)                    fig.add_axes((.1, .1, .5, np.nan))
(4)                with pytest.raises(TypeError, match="multiple values for argument
'rect'"):
(8)                    fig.add_axes([0, 0, 1, 1], rect=[0, 0, 1, 1])
(4)                fig2, ax = plt.subplots()
(4)                with pytest.raises(ValueError,
(23)                               match="The Axes must have been created in the present "
(29)                                     "figure"):
(8)                    fig.add_axes(ax)
(4)                fig2.delaxes(ax)
(4)                with pytest.warns(mpl.MatplotlibDeprecationWarning,
(22)                              match="Passing more than one positional argument"):
(8)                    fig2.add_axes(ax, "extra positional argument")
(4)                with pytest.warns(mpl.MatplotlibDeprecationWarning,
(22)                              match="Passing more than one positional argument"):
(8)                    fig.add_axes([0, 0, 1, 1], "extra positional argument")
(0)            def test_subplots_shareax_loglabels():
(4)                fig, axs = plt.subplots(2, 2, sharex=True, sharey=True, squeeze=False)
(4)                for ax in axs.flat:
(8)                    ax.plot([10, 20, 30], [10, 20, 30])
(4)                ax.set_yscale("log")
(4)                ax.set_xscale("log")
(4)                for ax in axs[0, :]:
(8)                    assert 0 == len(ax.xaxis.get_ticklabels(which='both'))
(4)                for ax in axs[1, :]:
(8)                    assert 0 < len(ax.xaxis.get_ticklabels(which='both'))
(4)                for ax in axs[:, 1]:
(8)                    assert 0 == len(ax.yaxis.get_ticklabels(which='both'))
(4)                for ax in axs[:, 0]:
(8)                    assert 0 < len(ax.yaxis.get_ticklabels(which='both'))
(0)            def test_savefig():
(4)                fig = plt.figure()
(4)                msg = r"savefig\(\) takes 2 positional arguments but 3 were given"
(4)                with pytest.raises(TypeError, match=msg):
(8)                    fig.savefig("fname1.png", "fname2.png")
(0)            def test_savefig_warns():
(4)                fig = plt.figure()
(4)                for format in ['png', 'pdf', 'svg', 'tif', 'jpg']:
(8)                    with pytest.raises(TypeError):
(12)                       fig.savefig(io.BytesIO(), format=format, non_existent_kwarg=True)
(0)            def test_savefig_backend():
(4)                fig = plt.figure()
(4)                with pytest.raises(ModuleNotFoundError, match="No module named
'@absent'"):
(8)                    fig.savefig("test", backend="module://@absent")
(4)                with pytest.raises(ValueError,
(23)                               match="The 'pdf' backend does not support png output"):
(8)                    fig.savefig("test.png", backend="pdf")
(0)            @pytest.mark.parametrize('backend', [
(4)                pytest.param('Agg', marks=[pytest.mark.backend('Agg')]),
(4)                pytest.param('Cairo', marks=[pytest.mark.backend('Cairo')]),
(0)            ])
(0)            def test_savefig_pixel_ratio(backend):
(4)                fig, ax = plt.subplots()
(4)                ax.plot([1, 2, 3])
(4)                with io.BytesIO() as buf:
```

```
439: (8)                    fig.savefig(buf, format='png')
440: (8)                    ratio1 = Image.open(buf)
441: (8)                    ratio1.load()
442: (4)                fig, ax = plt.subplots()
443: (4)                ax.plot([1, 2, 3])
444: (4)                fig.canvas._set_device_pixel_ratio(2)
445: (4)                with io.BytesIO() as buf:
446: (8)                    fig.savefig(buf, format='png')
447: (8)                    ratio2 = Image.open(buf)
448: (8)                    ratio2.load()
449: (4)                assert ratio1 == ratio2
450: (0)            def test_savefig_preserve_layout_engine():
451: (4)                fig = plt.figure(layout='compressed')
452: (4)                fig.savefig(io.BytesIO(), bbox_inches='tight')
453: (4)                assert fig.get_layout_engine()._compress
454: (0)            def test_savefig_locate_colorbar():
455: (4)                fig, ax = plt.subplots()
456: (4)                pc = ax.pcolormesh(np.random.randn(2, 2))
457: (4)                cbar = fig.colorbar(pc, aspect=40)
458: (4)                fig.savefig(io.BytesIO(), bbox_inches=mpl.transforms.Bbox([[0, 0], [4,
4]]))
459: (4)                assert (cbar.ax.get_position(original=True).bounds !=
460: (12)                       cbar.ax.get_position(original=False).bounds)
461: (0)            @mpl.rc_context({"savefig.transparent": True})
462: (0)            @check_figures_equal(extensions=["png"])
463: (0)            def test_savefig_transparent(fig_test, fig_ref):
464: (4)                gs1 = fig_test.add_gridspec(3, 3, left=0.05, wspace=0.05)
465: (4)                f1 = fig_test.add_subfigure(gs1[:, :])
466: (4)                f2 = f1.add_subfigure(gs1[0, 0])
467: (4)                ax12 = f2.add_subplot(gs1[:, :])
468: (4)                ax1 = f1.add_subplot(gs1[:-1, :])
469: (4)                iax1 = ax1.inset_axes([.1, .2, .3, .4])
470: (4)                iax2 = iax1.inset_axes([.1, .2, .3, .4])
471: (4)                ax2 = fig_test.add_subplot(gs1[-1, :-1])
472: (4)                ax3 = fig_test.add_subplot(gs1[-1, -1])
473: (4)                for ax in [ax12, ax1, iax1, iax2, ax2, ax3]:
474: (8)                    ax.set(xticks=[], yticks=[])
475: (8)                    ax.spines[:].set_visible(False)
476: (0)            def test_figure_repr():
477: (4)                fig = plt.figure(figsize=(10, 20), dpi=10)
478: (4)                assert repr(fig) == "<Figure size 100x200 with 0 Axes>"
479: (0)            def test_valid_layouts():
480: (4)                fig = Figure(layout=None)
481: (4)                assert not fig.get_tight_layout()
482: (4)                assert not fig.get_constrained_layout()
483: (4)                fig = Figure(layout='tight')
484: (4)                assert fig.get_tight_layout()
485: (4)                assert not fig.get_constrained_layout()
486: (4)                fig = Figure(layout='constrained')
487: (4)                assert not fig.get_tight_layout()
488: (4)                assert fig.get_constrained_layout()
489: (0)            def test_invalid_layouts():
490: (4)                fig, ax = plt.subplots(layout="constrained")
491: (4)                with pytest.warns(UserWarning):
492: (8)                    fig.subplots_adjust(top=0.8)
493: (4)                assert isinstance(fig.get_layout_engine(), ConstrainedLayoutEngine)
494: (4)                wst = "The Figure parameters 'layout' and 'tight_layout'"
495: (4)                with pytest.warns(UserWarning, match=wst):
496: (8)                    fig = Figure(layout='tight', tight_layout=False)
497: (4)                assert isinstance(fig.get_layout_engine(), TightLayoutEngine)
498: (4)                wst = "The Figure parameters 'layout' and 'constrained_layout'"
499: (4)                with pytest.warns(UserWarning, match=wst):
500: (8)                    fig = Figure(layout='constrained', constrained_layout=False)
501: (4)                assert not isinstance(fig.get_layout_engine(), TightLayoutEngine)
502: (4)                assert isinstance(fig.get_layout_engine(), ConstrainedLayoutEngine)
503: (4)                with pytest.raises(ValueError,
504: (23)                                  match="Invalid value for 'layout'"):
505: (8)                    Figure(layout='foobar')
506: (4)                fig, ax = plt.subplots(layout="constrained")
```

```
(4)                 fig.set_layout_engine("tight")
(4)                 assert isinstance(fig.get_layout_engine(), TightLayoutEngine)
(4)                 fig.set_layout_engine("constrained")
(4)                 assert isinstance(fig.get_layout_engine(), ConstrainedLayoutEngine)
(4)                 fig, ax = plt.subplots(layout="constrained")
(4)                 pc = ax.pcolormesh(np.random.randn(2, 2))
(4)                 fig.colorbar(pc)
(4)                 with pytest.raises(RuntimeError, match='Colorbar layout of new layout'):
(8)                     fig.set_layout_engine("tight")
(4)                 fig.set_layout_engine("none")
(4)                 with pytest.raises(RuntimeError, match='Colorbar layout of new layout'):
(8)                     fig.set_layout_engine("tight")
(4)                 fig, ax = plt.subplots(layout="tight")
(4)                 pc = ax.pcolormesh(np.random.randn(2, 2))
(4)                 fig.colorbar(pc)
(4)                 with pytest.raises(RuntimeError, match='Colorbar layout of new layout'):
(8)                     fig.set_layout_engine("constrained")
(4)                 fig.set_layout_engine("none")
(4)                 assert isinstance(fig.get_layout_engine(), PlaceHolderLayoutEngine)
(4)                 with pytest.raises(RuntimeError, match='Colorbar layout of new layout'):
(8)                     fig.set_layout_engine("constrained")
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_tightlayout_autolayout_deconflict(fig_test, fig_ref):
(4)                 for fig, autolayout in zip([fig_ref, fig_test], [False, True]):
(8)                     with mpl.rc_context({'figure.autolayout': autolayout}):
(12)                        axes = fig.subplots(ncols=2)
(12)                        fig.tight_layout(w_pad=10)
(8)                     assert isinstance(fig.get_layout_engine(), PlaceHolderLayoutEngine)
(0)             @pytest.mark.parametrize('layout', ['constrained', 'compressed'])
(0)             def test_layout_change_warning(layout):
(4)                 """
(4)                 Raise a warning when a previously assigned layout changes to tight using
(4)                 plt.tight_layout().
(4)                 """
(4)                 fig, ax = plt.subplots(layout=layout)
(4)                 with pytest.warns(UserWarning, match='The figure layout has changed to'):
(8)                     plt.tight_layout()
(0)             def test_repeated_tightlayout():
(4)                 fig = Figure()
(4)                 fig.tight_layout()
(4)                 fig.tight_layout()
(4)                 fig.tight_layout()
(0)             @check_figures_equal(extensions=["png", "pdf"])
(0)             def test_add_artist(fig_test, fig_ref):
(4)                 fig_test.dpi = 100
(4)                 fig_ref.dpi = 100
(4)                 fig_test.subplots()
(4)                 l1 = plt.Line2D([.2, .7], [.7, .7], gid='l1')
(4)                 l2 = plt.Line2D([.2, .7], [.8, .8], gid='l2')
(4)                 r1 = plt.Circle((20, 20), 100, transform=None, gid='C1')
(4)                 r2 = plt.Circle((.7, .5), .05, gid='C2')
(4)                 r3 = plt.Circle((4.5, .8), .55, transform=fig_test.dpi_scale_trans,
(20)                               facecolor='crimson', gid='C3')
(4)                 for a in [l1, l2, r1, r2, r3]:
(8)                     fig_test.add_artist(a)
(4)                 l2.remove()
(4)                 ax2 = fig_ref.subplots()
(4)                 l1 = plt.Line2D([.2, .7], [.7, .7], transform=fig_ref.transFigure,
(20)                               gid='l1', zorder=21)
(4)                 r1 = plt.Circle((20, 20), 100, transform=None, clip_on=False, zorder=20,
(20)                               gid='C1')
(4)                 r2 = plt.Circle((.7, .5), .05, transform=fig_ref.transFigure, gid='C2',
(20)                               zorder=20)
(4)                 r3 = plt.Circle((4.5, .8), .55, transform=fig_ref.dpi_scale_trans,
(20)                               facecolor='crimson', clip_on=False, zorder=20, gid='C3')
(4)                 for a in [l1, r1, r2, r3]:
(8)                     ax2.add_artist(a)
(0)             @pytest.mark.parametrize("fmt", ["png", "pdf", "ps", "eps", "svg"])
(0)             def test_fspath(fmt, tmp_path):
```

```
(4)                 out = tmp_path / f"test.{fmt}"
(4)                 plt.savefig(out)
(4)                 with out.open("rb") as file:
(8)                     assert fmt.encode("ascii") in file.read(100).lower()
(0)             def test_tightbbox():
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_xlim(0, 1)
(4)                 t = ax.text(1., 0.5, 'This dangles over end')
(4)                 renderer = fig.canvas.get_renderer()
(4)                 x1Nom0 = 9.035  # inches
(4)                 assert abs(t.get_tightbbox(renderer).x1 - x1Nom0 * fig.dpi) < 2
(4)                 assert abs(ax.get_tightbbox(renderer).x1 - x1Nom0 * fig.dpi) < 2
(4)                 assert abs(fig.get_tightbbox(renderer).x1 - x1Nom0) < 0.05
(4)                 assert abs(fig.get_tightbbox(renderer).x0 - 0.679) < 0.05
(4)                 t.set_in_layout(False)
(4)                 x1Nom = 7.333
(4)                 assert abs(ax.get_tightbbox(renderer).x1 - x1Nom * fig.dpi) < 2
(4)                 assert abs(fig.get_tightbbox(renderer).x1 - x1Nom) < 0.05
(4)                 t.set_in_layout(True)
(4)                 x1Nom = 7.333
(4)                 assert abs(ax.get_tightbbox(renderer).x1 - x1Nom0 * fig.dpi) < 2
(4)                 assert abs(ax.get_tightbbox(renderer, bbox_extra_artists=[]).x1
(15)                            - x1Nom * fig.dpi) < 2
(0)             def test_axes_removal():
(4)                 fig, axs = plt.subplots(1, 2, sharex=True)
(4)                 axs[1].remove()
(4)                 axs[0].plot([datetime(2000, 1, 1), datetime(2000, 2, 1)], [0, 1])
(4)                 assert isinstance(axs[0].xaxis.get_major_formatter(),
(22)                                  mdates.AutoDateFormatter)
(4)                 fig, axs = plt.subplots(1, 2, sharex=True)
(4)                 axs[1].xaxis.set_major_formatter(ScalarFormatter())
(4)                 axs[1].remove()
(4)                 axs[0].plot([datetime(2000, 1, 1), datetime(2000, 2, 1)], [0, 1])
(4)                 assert isinstance(axs[0].xaxis.get_major_formatter(),
(22)                                  ScalarFormatter)
(0)             def test_removed_axis():
(4)                 fig, axs = plt.subplots(2, sharex=True)
(4)                 axs[0].remove()
(4)                 fig.canvas.draw()
(0)             @pytest.mark.parametrize('clear_meth', ['clear', 'clf'])
(0)             def test_figure_clear(clear_meth):
(4)                 fig = plt.figure()
(4)                 fig.clear()
(4)                 assert fig.axes == []
(4)                 ax = fig.add_subplot(111)
(4)                 getattr(fig, clear_meth)()
(4)                 assert fig.axes == []
(4)                 axes = [fig.add_subplot(2, 1, i+1) for i in range(2)]
(4)                 getattr(fig, clear_meth)()
(4)                 assert fig.axes == []
(4)                 gs = fig.add_gridspec(ncols=2, nrows=1)
(4)                 subfig = fig.add_subfigure(gs[0])
(4)                 subaxes = subfig.add_subplot(111)
(4)                 getattr(fig, clear_meth)()
(4)                 assert subfig not in fig.subfigs
(4)                 assert fig.axes == []
(4)                 subfig = fig.add_subfigure(gs[0])
(4)                 subaxes = subfig.add_subplot(111)
(4)                 mainaxes = fig.add_subplot(gs[1])
(4)                 mainaxes.remove()
(4)                 assert fig.axes == [subaxes]
(4)                 mainaxes = fig.add_subplot(gs[1])
(4)                 subaxes.remove()
(4)                 assert fig.axes == [mainaxes]
(4)                 assert subfig in fig.subfigs
(4)                 subaxes = subfig.add_subplot(111)
(4)                 assert mainaxes in fig.axes
(4)                 assert subaxes in fig.axes
(4)                 getattr(subfig, clear_meth)()
```

```
(4)                 assert subfig in fig.subfigs
(4)                 assert subaxes not in subfig.axes
(4)                 assert subaxes not in fig.axes
(4)                 assert mainaxes in fig.axes
(4)                 subaxes = subfig.add_subplot(111)
(4)                 getattr(fig, clear_meth)()
(4)                 assert fig.axes == []
(4)                 assert fig.subfigs == []
(4)                 subfigs = [fig.add_subfigure(gs[i]) for i in [0, 1]]
(4)                 subaxes = [sfig.add_subplot(111) for sfig in subfigs]
(4)                 assert all(ax in fig.axes for ax in subaxes)
(4)                 assert all(sfig in fig.subfigs for sfig in subfigs)
(4)                 getattr(subfigs[0], clear_meth)()
(4)                 assert subaxes[0] not in fig.axes
(4)                 assert subaxes[1] in fig.axes
(4)                 assert subfigs[1] in fig.subfigs
(4)                 getattr(subfigs[1], clear_meth)()
(4)                 subfigs = [fig.add_subfigure(gs[i]) for i in [0, 1]]
(4)                 subaxes = [sfig.add_subplot(111) for sfig in subfigs]
(4)                 assert all(ax in fig.axes for ax in subaxes)
(4)                 assert all(sfig in fig.subfigs for sfig in subfigs)
(4)                 getattr(fig, clear_meth)()
(4)                 assert fig.subfigs == []
(4)                 assert fig.axes == []
(0)             def test_clf_not_redefined():
(4)                 for klass in FigureBase.__subclasses__():
(8)                     assert 'clf' not in klass.__dict__
(0)             @mpl.style.context('mpl20')
(0)             def test_picking_does_not_stale():
(4)                 fig, ax = plt.subplots()
(4)                 ax.scatter([0], [0], [1000], picker=True)
(4)                 fig.canvas.draw()
(4)                 assert not fig.stale
(4)                 mouse_event = SimpleNamespace(x=ax.bbox.x0 + ax.bbox.width / 2,
(34)                                              y=ax.bbox.y0 + ax.bbox.height / 2,
(34)                                              inaxes=ax, guiEvent=None)
(4)                 fig.pick(mouse_event)
(4)                 assert not fig.stale
(0)             def test_add_subplot_twotuple():
(4)                 fig = plt.figure()
(4)                 ax1 = fig.add_subplot(3, 2, (3, 5))
(4)                 assert ax1.get_subplotspec().rowspan == range(1, 3)
(4)                 assert ax1.get_subplotspec().colspan == range(0, 1)
(4)                 ax2 = fig.add_subplot(3, 2, (4, 6))
(4)                 assert ax2.get_subplotspec().rowspan == range(1, 3)
(4)                 assert ax2.get_subplotspec().colspan == range(1, 2)
(4)                 ax3 = fig.add_subplot(3, 2, (3, 6))
(4)                 assert ax3.get_subplotspec().rowspan == range(1, 3)
(4)                 assert ax3.get_subplotspec().colspan == range(0, 2)
(4)                 ax4 = fig.add_subplot(3, 2, (4, 5))
(4)                 assert ax4.get_subplotspec().rowspan == range(1, 3)
(4)                 assert ax4.get_subplotspec().colspan == range(0, 2)
(4)                 with pytest.raises(IndexError):
(8)                     fig.add_subplot(3, 2, (6, 3))
(0)             @image_comparison(['tightbbox_box_aspect.svg'], style='mpl20',
(18)                              savefig_kwarg={'bbox_inches': 'tight',
(33)                                             'facecolor': 'teal'},
(18)                              remove_text=True)
(0)             def test_tightbbox_box_aspect():
(4)                 fig = plt.figure()
(4)                 gs = fig.add_gridspec(1, 2)
(4)                 ax1 = fig.add_subplot(gs[0, 0])
(4)                 ax2 = fig.add_subplot(gs[0, 1], projection='3d')
(4)                 ax1.set_box_aspect(.5)
(4)                 ax2.set_box_aspect((2, 1, 1))
(0)             @check_figures_equal(extensions=["svg", "pdf", "eps", "png"])
(0)             def test_animated_with_canvas_change(fig_test, fig_ref):
(4)                 ax_ref = fig_ref.subplots()
(4)                 ax_ref.plot(range(5))
```

```
(4)                 ax_test = fig_test.subplots()
(4)                 ax_test.plot(range(5), animated=True)
(0)             class TestSubplotMosaic:
(4)                 @check_figures_equal(extensions=["png"])
(4)                 @pytest.mark.parametrize(
(8)                     "x", [
(12)                        [["A", "A", "B"], ["C", "D", "B"]],
(12)                        [[1, 1, 2], [3, 4, 2]],
(12)                        (("A", "A", "B"), ("C", "D", "B")),
(12)                        ((1, 1, 2), (3, 4, 2))
(8)                     ]
(4)                 )
(4)                 def test_basic(self, fig_test, fig_ref, x):
(8)                     grid_axes = fig_test.subplot_mosaic(x)
(8)                     for k, ax in grid_axes.items():
(12)                        ax.set_title(k)
(8)                     labels = sorted(np.unique(x))
(8)                     assert len(labels) == len(grid_axes)
(8)                     gs = fig_ref.add_gridspec(2, 3)
(8)                     axA = fig_ref.add_subplot(gs[:1, :2])
(8)                     axA.set_title(labels[0])
(8)                     axB = fig_ref.add_subplot(gs[:, 2])
(8)                     axB.set_title(labels[1])
(8)                     axC = fig_ref.add_subplot(gs[1, 0])
(8)                     axC.set_title(labels[2])
(8)                     axD = fig_ref.add_subplot(gs[1, 1])
(8)                     axD.set_title(labels[3])
(4)                 @check_figures_equal(extensions=["png"])
(4)                 def test_all_nested(self, fig_test, fig_ref):
(8)                     x = [["A", "B"], ["C", "D"]]
(8)                     y = [["E", "F"], ["G", "H"]]
(8)                     fig_ref.set_layout_engine("constrained")
(8)                     fig_test.set_layout_engine("constrained")
(8)                     grid_axes = fig_test.subplot_mosaic([[x, y]])
(8)                     for ax in grid_axes.values():
(12)                        ax.set_title(ax.get_label())
(8)                     gs = fig_ref.add_gridspec(1, 2)
(8)                     gs_left = gs[0, 0].subgridspec(2, 2)
(8)                     for j, r in enumerate(x):
(12)                        for k, label in enumerate(r):
(16)                            fig_ref.add_subplot(gs_left[j, k]).set_title(label)
(8)                     gs_right = gs[0, 1].subgridspec(2, 2)
(8)                     for j, r in enumerate(y):
(12)                        for k, label in enumerate(r):
(16)                            fig_ref.add_subplot(gs_right[j, k]).set_title(label)
(4)                 @check_figures_equal(extensions=["png"])
(4)                 def test_nested(self, fig_test, fig_ref):
(8)                     fig_ref.set_layout_engine("constrained")
(8)                     fig_test.set_layout_engine("constrained")
(8)                     x = [["A", "B"], ["C", "D"]]
(8)                     y = [["F"], [x]]
(8)                     grid_axes = fig_test.subplot_mosaic(y)
(8)                     for k, ax in grid_axes.items():
(12)                        ax.set_title(k)
(8)                     gs = fig_ref.add_gridspec(2, 1)
(8)                     gs_n = gs[1, 0].subgridspec(2, 2)
(8)                     axA = fig_ref.add_subplot(gs_n[0, 0])
(8)                     axA.set_title("A")
(8)                     axB = fig_ref.add_subplot(gs_n[0, 1])
(8)                     axB.set_title("B")
(8)                     axC = fig_ref.add_subplot(gs_n[1, 0])
(8)                     axC.set_title("C")
(8)                     axD = fig_ref.add_subplot(gs_n[1, 1])
(8)                     axD.set_title("D")
(8)                     axF = fig_ref.add_subplot(gs[0, 0])
(8)                     axF.set_title("F")
(4)                 @check_figures_equal(extensions=["png"])
(4)                 def test_nested_tuple(self, fig_test, fig_ref):
(8)                     x = [["A", "B", "B"], ["C", "C", "D"]]
```

```
(8)                 xt = (("A", "B", "B"), ("C", "C", "D"))
(8)                 fig_ref.subplot_mosaic([["F"], [x]])
(8)                 fig_test.subplot_mosaic([["F"], [xt]])
(4)             def test_nested_width_ratios(self):
(8)                 x = [["A", [["B"],
(20)                            ["C"]]]]
(8)                 width_ratios = [2, 1]
(8)                 fig, axd = plt.subplot_mosaic(x, width_ratios=width_ratios)
(8)                 assert axd["A"].get_gridspec().get_width_ratios() == width_ratios
(8)                 assert axd["B"].get_gridspec().get_width_ratios() != width_ratios
(4)             def test_nested_height_ratios(self):
(8)                 x = [["A", [["B"],
(20)                            ["C"]]], ["D", "D"]]
(8)                 height_ratios = [1, 2]
(8)                 fig, axd = plt.subplot_mosaic(x, height_ratios=height_ratios)
(8)                 assert axd["D"].get_gridspec().get_height_ratios() == height_ratios
(8)                 assert axd["B"].get_gridspec().get_height_ratios() != height_ratios
(4)             @check_figures_equal(extensions=["png"])
(4)             @pytest.mark.parametrize(
(8)                 "x, empty_sentinel",
(8)                 [
(12)                    ([["A", None], [None, "B"]], None),
(12)                    ([["A", "."], ["."], "B"]], "SKIP"),
(12)                    ([["A", 0], [0, "B"]], 0),
(12)                    ([[1, None], [None, 2]], None),
(12)                    ([[1, "."], ["."], 2]], "SKIP"),
(12)                    ([[1, 0], [0, 2]], 0),
(8)                 ],
(4)             )
(4)             def test_empty(self, fig_test, fig_ref, x, empty_sentinel):
(8)                 if empty_sentinel != "SKIP":
(12)                    kwargs = {"empty_sentinel": empty_sentinel}
(8)                 else:
(12)                    kwargs = {}
(8)                 grid_axes = fig_test.subplot_mosaic(x, **kwargs)
(8)                 for k, ax in grid_axes.items():
(12)                    ax.set_title(k)
(8)                 labels = sorted(
(12)                    {name for row in x for name in row} - {empty_sentinel, "."}
(8)                 )
(8)                 assert len(labels) == len(grid_axes)
(8)                 gs = fig_ref.add_gridspec(2, 2)
(8)                 axA = fig_ref.add_subplot(gs[0, 0])
(8)                 axA.set_title(labels[0])
(8)                 axB = fig_ref.add_subplot(gs[1, 1])
(8)                 axB.set_title(labels[1])
(4)             def test_fail_list_of_str(self):
(8)                 with pytest.raises(ValueError, match='must be 2D'):
(12)                    plt.subplot_mosaic(['foo', 'bar'])
(8)                 with pytest.raises(ValueError, match='must be 2D'):
(12)                    plt.subplot_mosaic(['foo'])
(8)                 with pytest.raises(ValueError, match='must be 2D'):
(12)                    plt.subplot_mosaic([['foo', ('bar',)]])
(8)                 with pytest.raises(ValueError, match='must be 2D'):
(12)                    plt.subplot_mosaic([['a', 'b'], [('a', 'b'), 'c']])
(4)             @check_figures_equal(extensions=["png"])
(4)             @pytest.mark.parametrize("subplot_kw", [{}, {"projection": "polar"},
None])
(4)             def test_subplot_kw(self, fig_test, fig_ref, subplot_kw):
(8)                 x = [[1, 2]]
(8)                 grid_axes = fig_test.subplot_mosaic(x, subplot_kw=subplot_kw)
(8)                 subplot_kw = subplot_kw or {}
(8)                 gs = fig_ref.add_gridspec(1, 2)
(8)                 axA = fig_ref.add_subplot(gs[0, 0], **subplot_kw)
(8)                 axB = fig_ref.add_subplot(gs[0, 1], **subplot_kw)
(4)             @check_figures_equal(extensions=["png"])
(4)             @pytest.mark.parametrize("multi_value", ['BC', tuple('BC')])
(4)             def test_per_subplot_kw(self, fig_test, fig_ref, multi_value):
(8)                 x = 'AB;CD'
```

```
(8)                     grid_axes = fig_test.subplot_mosaic(
(12)                        x,
(12)                        subplot_kw={'facecolor': 'red'},
(12)                        per_subplot_kw={
(16)                            'D': {'facecolor': 'blue'},
(16)                            multi_value: {'facecolor': 'green'},
(12)                        }
(8)                     )
(8)                     gs = fig_ref.add_gridspec(2, 2)
(8)                     for color, spec in zip(['red', 'green', 'green', 'blue'], gs):
(12)                        fig_ref.add_subplot(spec, facecolor=color)
(4)                 def test_string_parser(self):
(8)                     normalize = Figure._normalize_grid_string
(8)                     assert normalize('ABC') == [['A', 'B', 'C']]
(8)                     assert normalize('AB;CC') == [['A', 'B'], ['C', 'C']]
(8)                     assert normalize('AB;CC;DE') == [['A', 'B'], ['C', 'C'], ['D', 'E']]
(8)                     assert normalize("""
(25)                                    ABC
(25)                                    """) == [['A', 'B', 'C']]
(8)                     assert normalize("""
(25)                                    AB
(25)                                    CC
(25)                                    """) == [['A', 'B'], ['C', 'C']]
(8)                     assert normalize("""
(25)                                    AB
(25)                                    CC
(25)                                    DE
(25)                                    """) == [['A', 'B'], ['C', 'C'], ['D', 'E']]
(4)                 def test_per_subplot_kw_expander(self):
(8)                     normalize = Figure._norm_per_subplot_kw
(8)                     assert normalize({"A": {}, "B": {}}) == {"A": {}, "B": {}}
(8)                     assert normalize({("A", "B"): {}}) == {"A": {}, "B": {}}
(8)                     with pytest.raises(
(16)                            ValueError, match=f'The key {"B"!r} appears multiple times'
(8)                     ):
(12)                        normalize({("A", "B"): {}, "B": {}})
(8)                     with pytest.raises(
(16)                            ValueError, match=f'The key {"B"!r} appears multiple times'
(8)                     ):
(12)                        normalize({"B": {}, ("A", "B"): {}})
(4)                 def test_extra_per_subplot_kw(self):
(8)                     with pytest.raises(
(16)                            ValueError, match=f'The keys {set("B")!r} are in'
(8)                     ):
(12)                        Figure().subplot_mosaic("A", per_subplot_kw={"B": {}})
(4)                 @check_figures_equal(extensions=["png"])
(4)                 @pytest.mark.parametrize("str_pattern",
(29)                                         ["AAA\nBBB", "\nAAA\nBBB\n", "ABC\nDEF"]
(29)                                         )
(4)                 def test_single_str_input(self, fig_test, fig_ref, str_pattern):
(8)                     grid_axes = fig_test.subplot_mosaic(str_pattern)
(8)                     grid_axes = fig_ref.subplot_mosaic(
(12)                        [list(ln) for ln in str_pattern.strip().split("\n")]
(8)                     )
(4)                 @pytest.mark.parametrize(
(8)                     "x,match",
(8)                     [
(12)                        (
(16)                            [["A", "."], ["."], "A"]],
(16)                            (
(20)                                "(?m)we found that the label .A. specifies a "
(20)                                + "non-rectangular or non-contiguous area."
(16)                            ),
(12)                        ),
(12)                        (
(16)                            [["A", "B"], [None, [["A", "B"], ["C", "D"]]]],
(16)                            "There are duplicate keys .* between the outer layout",
(12)                        ),
(12)                        ("AAA\nc\nBBB", "All of the rows must be the same length"),
```

```
(12)                        (
(16)                            [["A", [["B", "C"], ["D"]]], ["E", "E"]],
(16)                            "All of the rows must be the same length",
(12)                        ),
(8)                     ],
(4)                 )
(4)                 def test_fail(self, x, match):
(8)                     fig = plt.figure()
(8)                     with pytest.raises(ValueError, match=match):
(12)                        fig.subplot_mosaic(x)
(4)                 @check_figures_equal(extensions=["png"])
(4)                 def test_hashable_keys(self, fig_test, fig_ref):
(8)                     fig_test.subplot_mosaic([[object(), object()]])
(8)                     fig_ref.subplot_mosaic([["A", "B"]])
(4)                 @pytest.mark.parametrize('str_pattern',
(29)                                             ['abc', 'cab', 'bca', 'cba', 'acb', 'bac'])
(4)                 def test_user_order(self, str_pattern):
(8)                     fig = plt.figure()
(8)                     ax_dict = fig.subplot_mosaic(str_pattern)
(8)                     assert list(str_pattern) == list(ax_dict)
(8)                     assert list(fig.axes) == list(ax_dict.values())
(4)                 def test_nested_user_order(self):
(8)                     layout = [
(12)                        ["A", [["B", "C"],
(19)                               ["D", "E"]]],
(12)                        ["F", "G"],
(12)                        ["·", [["H", [["I"],
(26)                                      ["·"]]]]]
(8)                     ]
(8)                     fig = plt.figure()
(8)                     ax_dict = fig.subplot_mosaic(layout)
(8)                     assert list(ax_dict) == list("ABCDEFGHI")
(8)                     assert list(fig.axes) == list(ax_dict.values())
(4)                 def test_share_all(self):
(8)                     layout = [
(12)                        ["A", [["B", "C"],
(19)                               ["D", "E"]]],
(12)                        ["F", "G"],
(12)                        ["·", [["H", [["I"],
(26)                                      ["·"]]]]]
(8)                     ]
(8)                     fig = plt.figure()
(8)                     ax_dict = fig.subplot_mosaic(layout, sharex=True, sharey=True)
(8)                     ax_dict["A"].set(xscale="log", yscale="logit")
(8)                     assert all(ax.get_xscale() == "log" and ax.get_yscale() == "logit"
(19)                               for ax in ax_dict.values())
(0)             def test_reused_gridspec():
(4)                 """Test that these all use the same gridspec"""
(4)                 fig = plt.figure()
(4)                 ax1 = fig.add_subplot(3, 2, (3, 5))
(4)                 ax2 = fig.add_subplot(3, 2, 4)
(4)                 ax3 = plt.subplot2grid((3, 2), (2, 1), colspan=2, fig=fig)
(4)                 gs1 = ax1.get_subplotspec().get_gridspec()
(4)                 gs2 = ax2.get_subplotspec().get_gridspec()
(4)                 gs3 = ax3.get_subplotspec().get_gridspec()
(4)                 assert gs1 == gs2
(4)                 assert gs1 == gs3
(0)             @image_comparison(['test_subfigure.png'], style='mpl20',
(18)                              savefig_kwarg={'facecolor': 'teal'})
(0)             def test_subfigure():
(4)                 np.random.seed(19680801)
(4)                 fig = plt.figure(layout='constrained')
(4)                 sub = fig.subfigures(1, 2)
(4)                 axs = sub[0].subplots(2, 2)
(4)                 for ax in axs.flat:
(8)                     pc = ax.pcolormesh(np.random.randn(30, 30), vmin=-2, vmax=2)
(4)                 sub[0].colorbar(pc, ax=axs)
(4)                 sub[0].suptitle('Left Side')
(4)                 sub[0].set_facecolor('white')
```

```
(4)                 axs = sub[1].subplots(1, 3)
(4)                 for ax in axs.flat:
(8)                     pc = ax.pcolormesh(np.random.randn(30, 30), vmin=-2, vmax=2)
(4)                 sub[1].colorbar(pc, ax=axs, location='bottom')
(4)                 sub[1].suptitle('Right Side')
(4)                 sub[1].set_facecolor('white')
(4)                 fig.suptitle('Figure suptitle', fontsize='xx-large')
(4)                 leg = fig.legend(handles=[plt.Line2D([0], [0], label='Line{}'.format(i))
(21)                                 for i in range(5)], loc='center')
(4)                 sub[0].set_zorder(leg.get_zorder() - 1)
(4)                 sub[1].set_zorder(leg.get_zorder() + 1)
(0)             def test_subfigure_tightbbox():
(4)                 fig = plt.figure(layout='constrained')
(4)                 sub = fig.subfigures(1, 2)
(4)                 np.testing.assert_allclose(
(12)                        fig.get_tightbbox(fig.canvas.get_renderer()).width,
(12)                        8.0)
(0)             def test_subfigure_dpi():
(4)                 fig = plt.figure(dpi=100)
(4)                 sub_fig = fig.subfigures()
(4)                 assert sub_fig.get_dpi() == fig.get_dpi()
(4)                 sub_fig.set_dpi(200)
(4)                 assert sub_fig.get_dpi() == 200
(4)                 assert fig.get_dpi() == 200
(0)             @image_comparison(['test_subfigure_ss.png'], style='mpl20',
(18)                              savefig_kwarg={'facecolor': 'teal'}, tol=0.02)
(0)             def test_subfigure_ss():
(4)                 np.random.seed(19680801)
(4)                 fig = plt.figure(layout='constrained')
(4)                 gs = fig.add_gridspec(1, 2)
(4)                 sub = fig.add_subfigure(gs[0], facecolor='pink')
(4)                 axs = sub.subplots(2, 2)
(4)                 for ax in axs.flat:
(8)                     pc = ax.pcolormesh(np.random.randn(30, 30), vmin=-2, vmax=2)
(4)                 sub.colorbar(pc, ax=axs)
(4)                 sub.suptitle('Left Side')
(4)                 ax = fig.add_subplot(gs[1])
(4)                 ax.plot(np.arange(20))
(4)                 ax.set_title('Axes')
(4)                 fig.suptitle('Figure suptitle', fontsize='xx-large')
(0)             @image_comparison(['test_subfigure_double.png'], style='mpl20',
(18)                              savefig_kwarg={'facecolor': 'teal'})
(0)             def test_subfigure_double():
(4)                 np.random.seed(19680801)
(4)                 fig = plt.figure(layout='constrained', figsize=(10, 8))
(4)                 fig.suptitle('fig')
(4)                 subfigs = fig.subfigures(1, 2, wspace=0.07)
(4)                 subfigs[0].set_facecolor('coral')
(4)                 subfigs[0].suptitle('subfigs[0]')
(4)                 subfigs[1].set_facecolor('coral')
(4)                 subfigs[1].suptitle('subfigs[1]')
(4)                 subfigsnest = subfigs[0].subfigures(2, 1, height_ratios=[1, 1.4])
(4)                 subfigsnest[0].suptitle('subfigsnest[0]')
(4)                 subfigsnest[0].set_facecolor('r')
(4)                 axsnest0 = subfigsnest[0].subplots(1, 2, sharey=True)
(4)                 for ax in axsnest0:
(8)                     fontsize = 12
(8)                     pc = ax.pcolormesh(np.random.randn(30, 30), vmin=-2.5, vmax=2.5)
(8)                     ax.set_xlabel('x-label', fontsize=fontsize)
(8)                     ax.set_ylabel('y-label', fontsize=fontsize)
(8)                     ax.set_title('Title', fontsize=fontsize)
(4)                 subfigsnest[0].colorbar(pc, ax=axsnest0)
(4)                 subfigsnest[1].suptitle('subfigsnest[1]')
(4)                 subfigsnest[1].set_facecolor('g')
(4)                 axsnest1 = subfigsnest[1].subplots(3, 1, sharex=True)
(4)                 for nn, ax in enumerate(axsnest1):
(8)                     ax.set_ylabel(f'ylabel{nn}')
(4)                 subfigsnest[1].supxlabel('supxlabel')
(4)                 subfigsnest[1].supylabel('supylabel')
```

```
(4)             axsRight = subfigs[1].subplots(2, 2)
(0)         def test_subfigure_spanning():
(4)             fig = plt.figure(constrained_layout=True)
(4)             gs = fig.add_gridspec(3, 3)
(4)             sub_figs = [
(8)                 fig.add_subfigure(gs[0, 0]),
(8)                 fig.add_subfigure(gs[0:2, 1]),
(8)                 fig.add_subfigure(gs[2, 1:3]),
(8)                 fig.add_subfigure(gs[0:, 1:])
(4)             ]
(4)             w = 640
(4)             h = 480
(4)             np.testing.assert_allclose(sub_figs[0].bbox.min, [0., h * 2/3])
(4)             np.testing.assert_allclose(sub_figs[0].bbox.max, [w / 3, h])
(4)             np.testing.assert_allclose(sub_figs[1].bbox.min, [w / 3, h / 3])
(4)             np.testing.assert_allclose(sub_figs[1].bbox.max, [w * 2/3, h])
(4)             np.testing.assert_allclose(sub_figs[2].bbox.min, [w / 3, 0])
(4)             np.testing.assert_allclose(sub_figs[2].bbox.max, [w, h / 3])
(4)             for i in range(4):
(8)                 sub_figs[i].add_subplot()
(4)             fig.draw_without_rendering()
(0)         @mpl.style.context('mpl20')
(0)         def test_subfigure_ticks():
(4)             fig = plt.figure(constrained_layout=True, figsize=(10, 3))
(4)             (subfig_bl, subfig_br) = fig.subfigures(1, 2, wspace=0.01,
(44)                                                     width_ratios=[7, 2])
(4)             gs = subfig_bl.add_gridspec(nrows=1, ncols=14)
(4)             ax1 = subfig_bl.add_subplot(gs[0, :1])
(4)             ax1.scatter(x=[-56.46881504821776, 24.179891162109396], y=[1500, 3600])
(4)             ax2 = subfig_bl.add_subplot(gs[0, 1:3], sharey=ax1)
(4)             ax2.scatter(x=[-126.5357270050049, 94.68456736755368], y=[1500, 3600])
(4)             ax3 = subfig_bl.add_subplot(gs[0, 3:14], sharey=ax1)
(4)             fig.dpi = 120
(4)             fig.draw_without_rendering()
(4)             ticks120 = ax2.get_xticks()
(4)             fig.dpi = 300
(4)             fig.draw_without_rendering()
(4)             ticks300 = ax2.get_xticks()
(4)             np.testing.assert_allclose(ticks120, ticks300)
(0)         @image_comparison(['test_subfigure_scatter_size.png'], style='mpl20',
(19)                          remove_text=True)
(0)         def test_subfigure_scatter_size():
(4)             fig = plt.figure()
(4)             gs = fig.add_gridspec(1, 2)
(4)             ax0 = fig.add_subplot(gs[1])
(4)             ax0.scatter([1, 2, 3], [1, 2, 3], s=30, marker='s')
(4)             ax0.scatter([3, 4, 5], [1, 2, 3], s=[20, 30, 40], marker='s')
(4)             sfig = fig.add_subfigure(gs[0])
(4)             axs = sfig.subplots(1, 2)
(4)             for ax in [ax0, axs[0]]:
(8)                 ax.scatter([1, 2, 3], [1, 2, 3], s=30, marker='s', color='r')
(8)                 ax.scatter([3, 4, 5], [1, 2, 3], s=[20, 30, 40], marker='s',
color='g')
(0)         def test_subfigure_pdf():
(4)             fig = plt.figure(layout='constrained')
(4)             sub_fig = fig.subfigures()
(4)             ax = sub_fig.add_subplot(111)
(4)             b = ax.bar(1, 1)
(4)             ax.bar_label(b)
(4)             buffer = io.BytesIO()
(4)             fig.savefig(buffer, format='pdf')
(0)         def test_subfigures_wspace_hspace():
(4)             sub_figs = plt.figure().subfigures(2, 3, hspace=0.5, wspace=1/6.)
(4)             w = 640
(4)             h = 480
(4)             np.testing.assert_allclose(sub_figs[0, 0].bbox.min, [0., h * 0.6])
(4)             np.testing.assert_allclose(sub_figs[0, 0].bbox.max, [w * 0.3, h])
(4)             np.testing.assert_allclose(sub_figs[0, 1].bbox.min, [w * 0.35, h * 0.6])
(4)             np.testing.assert_allclose(sub_figs[0, 1].bbox.max, [w * 0.65, h])
```

```
1126: (4)               np.testing.assert_allclose(sub_figs[0, 2].bbox.min, [w * 0.7, h * 0.6])
1127: (4)               np.testing.assert_allclose(sub_figs[0, 2].bbox.max, [w, h])
1128: (4)               np.testing.assert_allclose(sub_figs[1, 0].bbox.min, [0, 0])
1129: (4)               np.testing.assert_allclose(sub_figs[1, 0].bbox.max, [w * 0.3, h * 0.4])
1130: (4)               np.testing.assert_allclose(sub_figs[1, 1].bbox.min, [w * 0.35, 0])
1131: (4)               np.testing.assert_allclose(sub_figs[1, 1].bbox.max, [w * 0.65, h * 0.4])
1132: (4)               np.testing.assert_allclose(sub_figs[1, 2].bbox.min, [w * 0.7, 0])
1133: (4)               np.testing.assert_allclose(sub_figs[1, 2].bbox.max, [w, h * 0.4])
1134: (0)           def test_subfigure_remove():
1135: (4)               fig = plt.figure()
1136: (4)               sfs = fig.subfigures(2, 2)
1137: (4)               sfs[1, 1].remove()
1138: (4)               assert len(fig.subfigs) == 3
1139: (0)           def test_add_subplot_kwargs():
1140: (4)               fig = plt.figure()
1141: (4)               ax = fig.add_subplot(1, 1, 1)
1142: (4)               ax1 = fig.add_subplot(1, 1, 1)
1143: (4)               assert ax is not None
1144: (4)               assert ax1 is not ax
1145: (4)               plt.close()
1146: (4)               fig = plt.figure()
1147: (4)               ax = fig.add_subplot(1, 1, 1, projection='polar')
1148: (4)               ax1 = fig.add_subplot(1, 1, 1, projection='polar')
1149: (4)               assert ax is not None
1150: (4)               assert ax1 is not ax
1151: (4)               plt.close()
1152: (4)               fig = plt.figure()
1153: (4)               ax = fig.add_subplot(1, 1, 1, projection='polar')
1154: (4)               ax1 = fig.add_subplot(1, 1, 1)
1155: (4)               assert ax is not None
1156: (4)               assert ax1.name == 'rectilinear'
1157: (4)               assert ax1 is not ax
1158: (4)               plt.close()
1159: (0)           def test_add_axes_kwargs():
1160: (4)               fig = plt.figure()
1161: (4)               ax = fig.add_axes([0, 0, 1, 1])
1162: (4)               ax1 = fig.add_axes([0, 0, 1, 1])
1163: (4)               assert ax is not None
1164: (4)               assert ax1 is not ax
1165: (4)               plt.close()
1166: (4)               fig = plt.figure()
1167: (4)               ax = fig.add_axes([0, 0, 1, 1], projection='polar')
1168: (4)               ax1 = fig.add_axes([0, 0, 1, 1], projection='polar')
1169: (4)               assert ax is not None
1170: (4)               assert ax1 is not ax
1171: (4)               plt.close()
1172: (4)               fig = plt.figure()
1173: (4)               ax = fig.add_axes([0, 0, 1, 1], projection='polar')
1174: (4)               ax1 = fig.add_axes([0, 0, 1, 1])
1175: (4)               assert ax is not None
1176: (4)               assert ax1.name == 'rectilinear'
1177: (4)               assert ax1 is not ax
1178: (4)               plt.close()
1179: (0)           def test_ginput(recwarn):  # recwarn undoes warn filters at exit.
1180: (4)               warnings.filterwarnings("ignore", "cannot show the figure")
1181: (4)               fig, ax = plt.subplots()
1182: (4)               trans = ax.transData.transform
1183: (4)               def single_press():
1184: (8)                   MouseEvent("button_press_event", fig.canvas, *trans((.1, .2)),
1).._process()
1185: (4)               Timer(.1, single_press).start()
1186: (4)               assert fig.ginput() == [(.1, .2)]
1187: (4)               def multi_presses():
1188: (8)                   MouseEvent("button_press_event", fig.canvas, *trans((.1, .2)),
1).._process()
1189: (8)                   KeyEvent("key_press_event", fig.canvas, "backspace")._process()
1190: (8)                   MouseEvent("button_press_event", fig.canvas, *trans((.3, .4)),
1).._process()
1191: (8)                   MouseEvent("button_press_event", fig.canvas, *trans((.5, .6)),
```

```
1)._process()
1192: (8)                   MouseEvent("button_press_event", fig.canvas, *trans((0, 0)),
2)._process()
1193: (4)               Timer(.1, multi_presses).start()
1194: (4)               np.testing.assert_allclose(fig.ginput(3), [(.3, .4), (.5, .6)])
1195: (0)           def test_waitforbuttonpress(recwarn):  # recwarn undoes warn filters at exit.
1196: (4)               warnings.filterwarnings("ignore", "cannot show the figure")
1197: (4)               fig = plt.figure()
1198: (4)               assert fig.waitforbuttonpress(timeout=.1) is None
1199: (4)               Timer(.1, KeyEvent("key_press_event", fig.canvas, "z")._process).start()
1200: (4)               assert fig.waitforbuttonpress() is True
1201: (4)               Timer(.1, MouseEvent("button_press_event", fig.canvas, 0, 0,
1)._process).start()
1202: (4)               assert fig.waitforbuttonpress() is False
1203: (0)           def test_kwargs_pass():
1204: (4)               fig = Figure(label='whole Figure')
1205: (4)               sub_fig = fig.subfigures(1, 1, label='sub figure')
1206: (4)               assert fig.get_label() == 'whole Figure'
1207: (4)               assert sub_fig.get_label() == 'sub figure'
1208: (0)           @check_figures_equal(extensions=["png"])
1209: (0)           def test_rcparams(fig_test, fig_ref):
1210: (4)               fig_ref.supxlabel("xlabel", weight='bold', size=15)
1211: (4)               fig_ref.supylabel("ylabel", weight='bold', size=15)
1212: (4)               fig_ref.suptitle("Title", weight='light', size=20)
1213: (4)               with mpl.rc_context({'figure.labelweight': 'bold',
1214: (25)                                   'figure.labelsize': 15,
1215: (25)                                   'figure.titleweight': 'light',
1216: (25)                                   'figure.titlesize': 20}):
1217: (8)                   fig_test.supxlabel("xlabel")
1218: (8)                   fig_test.supylabel("ylabel")
1219: (8)                   fig_test.suptitle("Title")
1220: (0)           def test_deepcopy():
1221: (4)               fig1, ax = plt.subplots()
1222: (4)               ax.plot([0, 1], [2, 3])
1223: (4)               ax.set_yscale('log')
1224: (4)               fig2 = copy.deepcopy(fig1)
1225: (4)               assert fig2.axes[0] is not ax
1226: (4)               assert fig2.axes[0].get_yscale() == 'log'
1227: (4)               fig2.axes[0].set_yscale('linear')
1228: (4)               assert ax.get_yscale() == 'log'
1229: (4)               ax.set_xlim(1e-1, 1e2)
1230: (4)               fig1.draw_without_rendering()
1231: (4)               fig2.draw_without_rendering()
1232: (4)               assert ax.get_xlim() == (1e-1, 1e2)
1233: (4)               assert fig2.axes[0].get_xlim() == (0, 1)
1234: (0)           def test_unpickle_with_device_pixel_ratio():
1235: (4)               fig = Figure(dpi=42)
1236: (4)               fig.canvas._set_device_pixel_ratio(7)
1237: (4)               assert fig.dpi == 42*7
1238: (4)               fig2 = pickle.loads(pickle.dumps(fig))
1239: (4)               assert fig2.dpi == 42
1240: (0)           def test_gridspec_no_mutate_input():
1241: (4)               gs = {'left': .1}
1242: (4)               gs_orig = dict(gs)
1243: (4)               plt.subplots(1, 2, width_ratios=[1, 2], gridspec_kw=gs)
1244: (4)               assert gs == gs_orig
1245: (4)               plt.subplot_mosaic('AB', width_ratios=[1, 2], gridspec_kw=gs)
1246: (0)           @pytest.mark.parametrize('fmt', ['eps', 'pdf', 'png', 'ps', 'svg', 'svgz'])
1247: (0)           def test_savefig_metadata(fmt):
1248: (4)               Figure().savefig(io.BytesIO(), format=fmt, metadata={})
1249: (0)           @pytest.mark.parametrize('fmt', ['jpeg', 'jpg', 'tif', 'tiff', 'webp', "raw",
"rgba"])
1250: (0)           def test_savefig_metadata_error(fmt):
1251: (4)               with pytest.raises(ValueError, match="metadata not supported"):
1252: (8)                   Figure().savefig(io.BytesIO(), format=fmt, metadata={})
1253: (0)           def test_get_constrained_layout_pads():
1254: (4)               params = {'w_pad': 0.01, 'h_pad': 0.02, 'wspace': 0.03, 'hspace': 0.04}
1255: (4)               expected = tuple([*params.values()])
1256: (4)               fig =
```

```
plt.figure(layout=mpl.layout_engine.ConstrainedLayoutEngine(**params))
1257: (4)               with pytest.warns(PendingDeprecationWarning, match="will be deprecated"):
1258: (8)                   assert fig.get_constrained_layout_pads() == expected
1259: (0)           def test_not_visible_figure():
1260: (4)               fig = Figure()
1261: (4)               buf = io.StringIO()
1262: (4)               fig.savefig(buf, format='svg')
1263: (4)               buf.seek(0)
1264: (4)               assert '<g ' in buf.read()
1265: (4)               fig.set_visible(False)
1266: (4)               buf = io.StringIO()
1267: (4)               fig.savefig(buf, format='svg')
1268: (4)               buf.seek(0)
1269: (4)               assert '<g ' not in buf.read()
1270: (0)           def test_warn_colorbar_mismatch():
1271: (4)               fig1, ax1 = plt.subplots()
1272: (4)               fig2, (ax2_1, ax2_2) = plt.subplots(2)
1273: (4)               im = ax1.imshow([[1, 2], [3, 4]])
1274: (4)               fig1.colorbar(im)  # should not warn
1275: (4)               with pytest.warns(UserWarning, match="different Figure"):
1276: (8)                   fig2.colorbar(im)
1277: (4)               with pytest.warns(UserWarning, match="different Figure"):
1278: (8)                   fig2.colorbar(im, ax=ax1)
1279: (4)               with pytest.warns(UserWarning, match="different Figure"):
1280: (8)                   fig2.colorbar(im, ax=ax2_1)
1281: (4)               with pytest.warns(UserWarning, match="different Figure"):
1282: (8)                   fig2.colorbar(im, cax=ax2_2)
1283: (4)               fig3 = plt.figure()
1284: (4)               fig4 = plt.figure()
1285: (4)               subfig3_1 = fig3.subfigures()
1286: (4)               subfig3_2 = fig3.subfigures()
1287: (4)               subfig4_1 = fig4.subfigures()
1288: (4)               ax3_1 = subfig3_1.subplots()
1289: (4)               ax3_2 = subfig3_1.subplots()
1290: (4)               ax4_1 = subfig4_1.subplots()
1291: (4)               im3_1 = ax3_1.imshow([[1, 2], [3, 4]])
1292: (4)               im3_2 = ax3_2.imshow([[1, 2], [3, 4]])
1293: (4)               im4_1 = ax4_1.imshow([[1, 2], [3, 4]])
1294: (4)               fig3.colorbar(im3_1)   # should not warn
1295: (4)               subfig3_1.colorbar(im3_1)   # should not warn
1296: (4)               subfig3_1.colorbar(im3_2)   # should not warn
1297: (4)               with pytest.warns(UserWarning, match="different Figure"):
1298: (8)                   subfig3_1.colorbar(im4_1)
1299: (0)           def test_subfigure_stale_propagation():
1300: (4)               fig = plt.figure()
1301: (4)               fig.draw_without_rendering()
1302: (4)               assert not fig.stale
1303: (4)               sfig1 = fig.subfigures()
1304: (4)               assert fig.stale
1305: (4)               fig.draw_without_rendering()
1306: (4)               assert not fig.stale
1307: (4)               assert not sfig1.stale
1308: (4)               sfig2 = sfig1.subfigures()
1309: (4)               assert fig.stale
1310: (4)               fig.draw_without_rendering()
1311: (4)               assert not fig.stale
1312: (4)               assert not sfig2.stale
1313: (4)               sfig2.stale = True
1314: (4)               assert fig.stale

----------------------------------------

File 185 - test_fontconfig_pattern.py:

1: (0)              import pytest
2: (0)              from matplotlib.font_manager import FontProperties
3: (0)              keys = [
4: (4)                  "get_family",
5: (4)                  "get_style",
```

```
6: (4)                  "get_variant",
7: (4)                  "get_weight",
8: (4)                  "get_size",
9: (4)                  ]
10: (0)             def test_fontconfig_pattern():
11: (4)                 """Test converting a FontProperties to string then back."""
12: (4)                 test = "defaults "
13: (4)                 f1 = FontProperties()
14: (4)                 s = str(f1)
15: (4)                 f2 = FontProperties(s)
16: (4)                 for k in keys:
17: (8)                     assert getattr(f1, k)() == getattr(f2, k)(), test + k
18: (4)                 test = "basic "
19: (4)                 f1 = FontProperties(family="serif", size=20, style="italic")
20: (4)                 s = str(f1)
21: (4)                 f2 = FontProperties(s)
22: (4)                 for k in keys:
23: (8)                     assert getattr(f1, k)() == getattr(f2, k)(), test + k
24: (4)                 test = "full "
25: (4)                 f1 = FontProperties(family="sans-serif", size=24, weight="bold",
26: (24)                                    style="oblique", variant="small-caps",
27: (24)                                    stretch="expanded")
28: (4)                 s = str(f1)
29: (4)                 f2 = FontProperties(s)
30: (4)                 for k in keys:
31: (8)                     assert getattr(f1, k)() == getattr(f2, k)(), test + k
32: (0)             def test_fontconfig_str():
33: (4)                 """Test FontProperties string conversions for correctness."""
34: (4)                 test = "defaults "
35: (4)                 s = ("sans\\-serif:style=normal:variant=normal:weight=normal"
36: (9)                      ":stretch=normal:size=12.0")
37: (4)                 font = FontProperties(s)
38: (4)                 right = FontProperties()
39: (4)                 for k in keys:
40: (8)                     assert getattr(font, k)() == getattr(right, k)(), test + k
41: (4)                 test = "full "
42: (4)                 s = ("serif-24:style=oblique:variant=small-caps:weight=bold"
43: (9)                      ":stretch=expanded")
44: (4)                 font = FontProperties(s)
45: (4)                 right = FontProperties(family="serif", size=24, weight="bold",
46: (27)                                       style="oblique", variant="small-caps",
47: (27)                                       stretch="expanded")
48: (4)                 for k in keys:
49: (8)                     assert getattr(font, k)() == getattr(right, k)(), test + k
50: (0)             def test_fontconfig_unknown_constant():
51: (4)                 with pytest.raises(ValueError, match="ParseException"):
52: (8)                     FontProperties(":unknown")
```

File 186 - test_font_manager.py:

```
1: (0)              from io import BytesIO, StringIO
2: (0)              import gc
3: (0)              import multiprocessing
4: (0)              import os
5: (0)              from pathlib import Path
6: (0)              from PIL import Image
7: (0)              import shutil
8: (0)              import sys
9: (0)              import warnings
10: (0)             import numpy as np
11: (0)             import pytest
12: (0)             from matplotlib.font_manager import (
13: (4)                 findfont, findSystemFonts, FontEntry, FontProperties, fontManager,
14: (4)                 json_dump, json_load, get_font, is_opentype_cff_font,
15: (4)                 MSUserFontDirectories, _get_fontconfig_fonts, ttfFontProperty)
16: (0)             from matplotlib import cbook, ft2font, pyplot as plt, rc_context, figure as
mfigure
```

```
 17: (0)             from matplotlib.testing import subprocess_run_helper,
 subprocess_run_for_testing
 18: (0)             has_fclist = shutil.which('fc-list') is not None
 19: (0)             def test_font_priority():
 20: (4)                 with rc_context(rc={
 21: (12)                        'font.sans-serif':
 22: (12)                        ['cmmi10', 'Bitstream Vera Sans']}):
 23: (8)                     fontfile = findfont(FontProperties(family=["sans-serif"]))
 24: (4)                 assert Path(fontfile).name == 'cmmi10.ttf'
 25: (4)                 font = get_font(fontfile)
 26: (4)                 cmap = font.get_charmap()
 27: (4)                 assert len(cmap) == 131
 28: (4)                 assert cmap[8729] == 30
 29: (0)             def test_score_weight():
 30: (4)                 assert 0 == fontManager.score_weight("regular", "regular")
 31: (4)                 assert 0 == fontManager.score_weight("bold", "bold")
 32: (4)                 assert (0 < fontManager.score_weight(400, 400) <
 33: (12)                        fontManager.score_weight("normal", "bold"))
 34: (4)                 assert (0 < fontManager.score_weight("normal", "regular") <
 35: (12)                        fontManager.score_weight("normal", "bold"))
 36: (4)                 assert (fontManager.score_weight("normal", "regular") ==
 37: (12)                        fontManager.score_weight(400, 400))
 38: (0)             def test_json_serialization(tmp_path):
 39: (4)                 json_dump(fontManager, tmp_path / "fontlist.json")
 40: (4)                 copy = json_load(tmp_path / "fontlist.json")
 41: (4)                 with warnings.catch_warnings():
 42: (8)                     warnings.filterwarnings('ignore', 'findfont: Font family.*not found')
 43: (8)                     for prop in ({'family': 'STIXGeneral'},
 44: (21)                                 {'family': 'Bitstream Vera Sans', 'weight': 700},
 45: (21)                                 {'family': 'no such font family'}):
 46: (12)                        fp = FontProperties(**prop)
 47: (12)                        assert (fontManager.findfont(fp, rebuild_if_missing=False) ==
 48: (20)                                copy.findfont(fp, rebuild_if_missing=False))
 49: (0)             def test_otf():
 50: (4)                 fname = '/usr/share/fonts/opentype/freefont/FreeMono.otf'
 51: (4)                 if Path(fname).exists():
 52: (8)                     assert is_opentype_cff_font(fname)
 53: (4)                 for f in fontManager.ttflist:
 54: (8)                     if 'otf' in f.fname:
 55: (12)                        with open(f.fname, 'rb') as fd:
 56: (16)                            res = fd.read(4) == b'OTTO'
 57: (12)                        assert res == is_opentype_cff_font(f.fname)
 58: (0)             @pytest.mark.skipif(sys.platform == "win32" or not has_fclist,
 59: (20)                                reason='no fontconfig installed')
 60: (0)             def test_get_fontconfig_fonts():
 61: (4)                 assert len(_get_fontconfig_fonts()) > 1
 62: (0)             @pytest.mark.parametrize('factor', [2, 4, 6, 8])
 63: (0)             def test_hinting_factor(factor):
 64: (4)                 font = findfont(FontProperties(family=["sans-serif"]))
 65: (4)                 font1 = get_font(font, hinting_factor=1)
 66: (4)                 font1.clear()
 67: (4)                 font1.set_size(12, 100)
 68: (4)                 font1.set_text('abc')
 69: (4)                 expected = font1.get_width_height()
 70: (4)                 hinted_font = get_font(font, hinting_factor=factor)
 71: (4)                 hinted_font.clear()
 72: (4)                 hinted_font.set_size(12, 100)
 73: (4)                 hinted_font.set_text('abc')
 74: (4)                 np.testing.assert_allclose(hinted_font.get_width_height(), expected,
 75: (31)                                           rtol=0.1)
 76: (0)             def test_utf16m_sfnt():
 77: (4)                 try:
 78: (8)                     entry = next(entry for entry in fontManager.ttflist
 79: (21)                                 if Path(entry.fname).name == "seguisbi.ttf")
 80: (4)                 except StopIteration:
 81: (8)                     pytest.skip("Couldn't find seguisbi.ttf font to test against.")
 82: (4)                 else:
 83: (8)                     assert entry.weight == 600
 84: (0)             def test_find_ttc():
```

```
 85: (4)                 fp = FontProperties(family=["WenQuanYi Zen Hei"])
 86: (4)                 if Path(findfont(fp)).name != "wqy-zenhei.ttc":
 87: (8)                     pytest.skip("Font wqy-zenhei.ttc may be missing")
 88: (4)                 fig, ax = plt.subplots()
 89: (4)                 ax.text(.5, .5, "\N{KANGXI RADICAL DRAGON}", fontproperties=fp)
 90: (4)                 for fmt in ["raw", "svg", "pdf", "ps"]:
 91: (8)                     fig.savefig(BytesIO(), format=fmt)
 92: (0)             def test_find_noto():
 93: (4)                 fp = FontProperties(family=["Noto Sans CJK SC", "Noto Sans CJK JP"])
 94: (4)                 name = Path(findfont(fp)).name
 95: (4)                 if name not in ("NotoSansCJKsc-Regular.otf", "NotoSansCJK-Regular.ttc"):
 96: (8)                     pytest.skip(f"Noto Sans CJK SC font may be missing (found {name})")
 97: (4)                 fig, ax = plt.subplots()
 98: (4)                 ax.text(0.5, 0.5, 'Hello, 你好', fontproperties=fp)
 99: (4)                 for fmt in ["raw", "svg", "pdf", "ps"]:
100: (8)                     fig.savefig(BytesIO(), format=fmt)
101: (0)             def test_find_invalid(tmp_path):
102: (4)                 with pytest.raises(FileNotFoundError):
103: (8)                     get_font(tmp_path / 'non-existent-font-name.ttf')
104: (4)                 with pytest.raises(FileNotFoundError):
105: (8)                     get_font(str(tmp_path / 'non-existent-font-name.ttf'))
106: (4)                 with pytest.raises(FileNotFoundError):
107: (8)                     get_font(bytes(tmp_path / 'non-existent-font-name.ttf'))
108: (4)                 from matplotlib.ft2font import FT2Font
109: (4)                 with pytest.raises(TypeError, match='font file or a binary-mode file'):
110: (8)                     FT2Font(StringIO())  # type: ignore[arg-type]
111: (0)             @pytest.mark.skipif(sys.platform != 'linux' or not has_fclist,
112: (20)                                 reason='only Linux with fontconfig installed')
113: (0)             def test_user_fonts_linux(tmpdir, monkeypatch):
114: (4)                 font_test_file = 'mpltest.ttf'
115: (4)                 fonts = findSystemFonts()
116: (4)                 if any(font_test_file in font for font in fonts):
117: (8)                     pytest.skip(f'{font_test_file} already exists in system fonts')
118: (4)                 user_fonts_dir = tmpdir.join('fonts')
119: (4)                 user_fonts_dir.ensure(dir=True)
120: (4)                 shutil.copyfile(Path(__file__).parent / font_test_file,
121: (20)                                user_fonts_dir.join(font_test_file))
122: (4)                 with monkeypatch.context() as m:
123: (8)                     m.setenv('XDG_DATA_HOME', str(tmpdir))
124: (8)                     _get_fontconfig_fonts.cache_clear()
125: (8)                     fonts = findSystemFonts()
126: (8)                     assert any(font_test_file in font for font in fonts)
127: (4)                 _get_fontconfig_fonts.cache_clear()
128: (0)             def test_addfont_as_path():
129: (4)                 """Smoke test that addfont() accepts pathlib.Path."""
130: (4)                 font_test_file = 'mpltest.ttf'
131: (4)                 path = Path(__file__).parent / font_test_file
132: (4)                 try:
133: (8)                     fontManager.addfont(path)
134: (8)                     added, = [font for font in fontManager.ttflist
135: (18)                              if font.fname.endswith(font_test_file)]
136: (8)                     fontManager.ttflist.remove(added)
137: (4)                 finally:
138: (8)                     to_remove = [font for font in fontManager.ttflist
139: (21)                                 if font.fname.endswith(font_test_file)]
140: (8)                     for font in to_remove:
141: (12)                        fontManager.ttflist.remove(font)
142: (0)             @pytest.mark.skipif(sys.platform != 'win32', reason='Windows only')
143: (0)             def test_user_fonts_win32():
144: (4)                 if not (os.environ.get('APPVEYOR') or os.environ.get('TF_BUILD')):
145: (8)                     pytest.xfail("This test should only run on CI (appveyor or azure) "
146: (21)                                 "as the developer's font directory should remain "
147: (21)                                 "unchanged.")
148: (4)                 pytest.xfail("We need to update the registry for this test to work")
149: (4)                 font_test_file = 'mpltest.ttf'
150: (4)                 fonts = findSystemFonts()
151: (4)                 if any(font_test_file in font for font in fonts):
152: (8)                     pytest.skip(f'{font_test_file} already exists in system fonts')
153: (4)                 user_fonts_dir = MSUserFontDirectories[0]
```

```
154: (4)                os.makedirs(user_fonts_dir)
155: (4)                shutil.copy(Path(__file__).parent / font_test_file, user_fonts_dir)
156: (4)                fonts = findSystemFonts()
157: (4)                assert any(font_test_file in font for font in fonts)
158: (0)            def _model_handler(_):
159: (4)                fig, ax = plt.subplots()
160: (4)                fig.savefig(BytesIO(), format="pdf")
161: (4)                plt.close()
162: (0)            @pytest.mark.skipif(not hasattr(os, "register_at_fork"),
163: (20)                               reason="Cannot register at_fork handlers")
164: (0)            def test_fork():
165: (4)                _model_handler(0)  # Make sure the font cache is filled.
166: (4)                ctx = multiprocessing.get_context("fork")
167: (4)                with ctx.Pool(processes=2) as pool:
168: (8)                    pool.map(_model_handler, range(2))
169: (0)            def test_missing_family(caplog):
170: (4)                plt.rcParams["font.sans-serif"] = ["this-font-does-not-exist"]
171: (4)                with caplog.at_level("WARNING"):
172: (8)                    findfont("sans")
173: (4)                assert [rec.getMessage() for rec in caplog.records] == [
174: (8)                    "findfont: Font family ['sans'] not found. "
175: (8)                    "Falling back to DejaVu Sans.",
176: (8)                    "findfont: Generic family 'sans' not found because none of the "
177: (8)                    "following families were found: this-font-does-not-exist",
178: (4)                ]
179: (0)            def _test_threading():
180: (4)                import threading
181: (4)                from matplotlib.ft2font import LOAD_NO_HINTING
182: (4)                import matplotlib.font_manager as fm
183: (4)                N = 10
184: (4)                b = threading.Barrier(N)
185: (4)                def bad_idea(n):
186: (8)                    b.wait()
187: (8)                    for j in range(100):
188: (12)                       font = fm.get_font(fm.findfont("DejaVu Sans"))
189: (12)                       font.set_text(str(n), 0.0, flags=LOAD_NO_HINTING)
190: (4)                threads = [
191: (8)                    threading.Thread(target=bad_idea, name=f"bad_thread_{j}", args=(j,))
192: (8)                    for j in range(N)
193: (4)                ]
194: (4)                for t in threads:
195: (8)                    t.start()
196: (4)                for t in threads:
197: (8)                    t.join()
198: (0)            def test_fontcache_thread_safe():
199: (4)                pytest.importorskip('threading')
200: (4)                subprocess_run_helper(_test_threading, timeout=10)
201: (0)            def test_lockfilefailure(tmp_path):
202: (4)                proc = subprocess_run_for_testing(
203: (8)                    [
204: (12)                       sys.executable,
205: (12)                       "-c",
206: (12)                       "import matplotlib;"
207: (12)                       "import os;"
208: (12)                       "p = matplotlib.get_cachedir();"
209: (12)                       "os.chmod(p, 0o555);"
210: (12)                       "import matplotlib.font_manager;"
211: (8)                    ],
212: (8)                    env={**os.environ, 'MPLCONFIGDIR': str(tmp_path)},
213: (8)                    check=True
214: (4)                )
215: (0)            def test_fontentry_dataclass():
216: (4)                fontent = FontEntry(name='font-name')
217: (4)                png = fontent._repr_png_()
218: (4)                img = Image.open(BytesIO(png))
219: (4)                assert img.width > 0
220: (4)                assert img.height > 0
221: (4)                html = fontent._repr_html_()
222: (4)                assert html.startswith("<img src=\"data:image/png;base64")
```

```
223: (0)            def test_fontentry_dataclass_invalid_path():
224: (4)                with pytest.raises(FileNotFoundError):
225: (8)                    fontent = FontEntry(fname='/random', name='font-name')
226: (8)                    fontent._repr_html_()
227: (0)            @pytest.mark.skipif(sys.platform == 'win32', reason='Linux or OS only')
228: (0)            def test_get_font_names():
229: (4)                paths_mpl = [cbook._get_data_path('fonts', subdir) for subdir in ['ttf']]
230: (4)                fonts_mpl = findSystemFonts(paths_mpl, fontext='ttf')
231: (4)                fonts_system = findSystemFonts(fontext='ttf')
232: (4)                ttf_fonts = []
233: (4)                for path in fonts_mpl + fonts_system:
234: (8)                    try:
235: (12)                       font = ft2font.FT2Font(path)
236: (12)                       prop = ttfFontProperty(font)
237: (12)                       ttf_fonts.append(prop.name)
238: (8)                    except Exception:
239: (12)                       pass
240: (4)                available_fonts = sorted(list(set(ttf_fonts)))
241: (4)                mpl_font_names = sorted(fontManager.get_font_names())
242: (4)                assert set(available_fonts) == set(mpl_font_names)
243: (4)                assert len(available_fonts) == len(mpl_font_names)
244: (4)                assert available_fonts == mpl_font_names
245: (0)            def test_donot_cache_tracebacks():
246: (4)                class SomeObject:
247: (8)                    pass
248: (4)                def inner():
249: (8)                    x = SomeObject()
250: (8)                    fig = mfigure.Figure()
251: (8)                    ax = fig.subplots()
252: (8)                    fig.text(.5, .5, 'aardvark', family='doesnotexist')
253: (8)                    with BytesIO() as out:
254: (12)                       with warnings.catch_warnings():
255: (16)                           warnings.filterwarnings('ignore')
256: (16)                           fig.savefig(out, format='raw')
257: (4)                inner()
258: (4)                for obj in gc.get_objects():
259: (8)                    if isinstance(obj, SomeObject):
260: (12)                       pytest.fail("object from inner stack still alive")
```

File 187 - test_ft2font.py:

```
1: (0)              from pathlib import Path
2: (0)              import io
3: (0)              import pytest
4: (0)              from matplotlib import ft2font
5: (0)              from matplotlib.testing.decorators import check_figures_equal
6: (0)              import matplotlib.font_manager as fm
7: (0)              import matplotlib.pyplot as plt
8: (0)              def test_fallback_errors():
9: (4)                  file_name = fm.findfont('DejaVu Sans')
10: (4)                 with pytest.raises(TypeError, match="Fallback list must be a list"):
11: (8)                     ft2font.FT2Font(file_name, _fallback_list=(0,))  # type: ignore[arg-
type]
12: (4)                 with pytest.raises(
13: (12)                        TypeError, match="Fallback fonts must be FT2Font objects."
14: (4)                 ):
15: (8)                     ft2font.FT2Font(file_name, _fallback_list=[0])  # type: ignore[list-
item]
16: (0)             def test_ft2font_positive_hinting_factor():
17: (4)                 file_name = fm.findfont('DejaVu Sans')
18: (4)                 with pytest.raises(
19: (12)                        ValueError, match="hinting_factor must be greater than 0"
20: (4)                 ):
21: (8)                     ft2font.FT2Font(file_name, 0)
22: (0)             @pytest.mark.parametrize('family_name, file_name',
23: (26)                                     [("WenQuanYi Zen Hei",  "wqy-zenhei.ttc"),
24: (27)                                      ("Noto Sans CJK JP", "NotoSansCJK.ttc"),
```

```
 25: (27)                                    ("Noto Sans TC", "NotoSansTC-Regular.otf")]
 26: (25)                                 )
 27: (0)             def test_fallback_smoke(family_name, file_name):
 28: (4)                 fp = fm.FontProperties(family=[family_name])
 29: (4)                 if Path(fm.findfont(fp)).name != file_name:
 30: (8)                     pytest.skip(f"Font {family_name} ({file_name}) is missing")
 31: (4)                 plt.rcParams['font.size'] = 20
 32: (4)                 fig = plt.figure(figsize=(4.75, 1.85))
 33: (4)                 fig.text(0.05, 0.45, "There are 几个汉字 in between!",
 34: (13)                         family=['DejaVu Sans', family_name])
 35: (4)                 fig.text(0.05, 0.85, "There are 几个汉字 in between!",
 36: (13)                         family=[family_name])
 37: (4)                 for fmt in ['png', 'raw']:  # ["svg", "pdf", "ps"]:
 38: (8)                     fig.savefig(io.BytesIO(), format=fmt)
 39: (0)             @pytest.mark.parametrize('family_name, file_name',
 40: (25)                                 [("WenQuanYi Zen Hei",  "wqy-zenhei"),
 41: (26)                                  ("Noto Sans CJK JP", "NotoSansCJK"),
 42: (26)                                  ("Noto Sans TC", "NotoSansTC-Regular.otf")]
 43: (25)                                 )
 44: (0)             @check_figures_equal(extensions=["png", "pdf", "eps", "svg"])
 45: (0)             def test_font_fallback_chinese(fig_test, fig_ref, family_name, file_name):
 46: (4)                 fp = fm.FontProperties(family=[family_name])
 47: (4)                 if file_name not in Path(fm.findfont(fp)).name:
 48: (8)                     pytest.skip(f"Font {family_name} ({file_name}) is missing")
 49: (4)                 text = ["There are", "几个汉字", "in between!"]
 50: (4)                 plt.rcParams["font.size"] = 20
 51: (4)                 test_fonts = [["DejaVu Sans", family_name]] * 3
 52: (4)                 ref_fonts = [["DejaVu Sans"], [family_name], ["DejaVu Sans"]]
 53: (4)                 for j, (txt, test_font, ref_font) in enumerate(
 54: (12)                        zip(text, test_fonts, ref_fonts)
 55: (4)                 ):
 56: (8)                     fig_ref.text(0.05, .85 - 0.15*j, txt, family=ref_font)
 57: (8)                     fig_test.text(0.05, .85 - 0.15*j, txt, family=test_font)
 58: (0)             @pytest.mark.parametrize("font_list",
 59: (26)                                  [['DejaVu Serif', 'DejaVu Sans'],
 60: (27)                                   ['DejaVu Sans Mono']],
 61: (25)                                 ids=["two fonts", "one font"])
 62: (0)             def test_fallback_missing(recwarn, font_list):
 63: (4)                 fig = plt.figure()
 64: (4)                 fig.text(.5, .5, "Hello 🙃 World!", family=font_list)
 65: (4)                 fig.canvas.draw()
 66: (4)                 assert all(isinstance(warn.message, UserWarning) for warn in recwarn)
 67: (4)                 assert recwarn[0].message.args[0].startswith(
 68: (11)                       "Glyph 128579 (\\N{UPSIDE-DOWN FACE}) missing from font(s)")
 69: (4)                 assert all([font in recwarn[0].message.args[0] for font in font_list])
 70: (0)             @pytest.mark.parametrize(
 71: (4)                 "family_name, file_name",
 72: (4)                 [
 73: (8)                     ("WenQuanYi Zen Hei", "wqy-zenhei"),
 74: (8)                     ("Noto Sans CJK JP", "NotoSansCJK"),
 75: (8)                     ("Noto Sans TC", "NotoSansTC-Regular.otf")
 76: (4)                 ],
 77: (0)             )
 78: (0)             def test__get_fontmap(family_name, file_name):
 79: (4)                 fp = fm.FontProperties(family=[family_name])
 80: (4)                 found_file_name = Path(fm.findfont(fp)).name
 81: (4)                 if file_name not in found_file_name:
 82: (8)                     pytest.skip(f"Font {family_name} ({file_name}) is missing")
 83: (4)                 text = "There are 几个汉字 in between!"
 84: (4)                 ft = fm.get_font(
 85: (8)                     fm.fontManager._find_fonts_by_props(
 86: (12)                        fm.FontProperties(family=["DejaVu Sans", family_name])
 87: (8)                     )
 88: (4)                 )
 89: (4)                 fontmap = ft._get_fontmap(text)
 90: (4)                 for char, font in fontmap.items():
 91: (8)                     if ord(char) > 127:
 92: (12)                        assert Path(font.fname).name == found_file_name
 93: (8)                     else:
```

```
94: (12)                        assert Path(font.fname).name == "DejaVuSans.ttf"

----------------------------------------

File 188 - test_getattr.py:

1: (0)              from importlib import import_module
2: (0)              from pkgutil import walk_packages
3: (0)              import matplotlib
4: (0)              import pytest
5: (0)              module_names = [
6: (4)                  m.name
7: (4)                  for m in walk_packages(
8: (8)                      path=matplotlib.__path__, prefix=f'{matplotlib.__name__}.'
9: (4)                  )
10: (4)                 if not m.name.startswith(__package__)
11: (4)                 and not any(x.startswith('_') for x in m.name.split('.'))
12: (0)             ]
13: (0)             @pytest.mark.parametrize('module_name', module_names)
14: (0)             @pytest.mark.filterwarnings('ignore::DeprecationWarning')
15: (0)             @pytest.mark.filterwarnings('ignore::ImportWarning')
16: (0)             def test_getattr(module_name):
17: (4)                 """
18: (4)                 Test that __getattr__ methods raise AttributeError for unknown keys.
19: (4)                 See #20822, #20855.
20: (4)                 """
21: (4)                 try:
22: (8)                     module = import_module(module_name)
23: (4)                 except (ImportError, RuntimeError, OSError) as e:
24: (8)                     pytest.skip(f'Cannot import {module_name} due to {e}')
25: (4)                 key = 'THIS_SYMBOL_SHOULD_NOT_EXIST'
26: (4)                 if hasattr(module, key):
27: (8)                     delattr(module, key)

----------------------------------------

File 189 - test_gridspec.py:

1: (0)              import matplotlib.gridspec as gridspec
2: (0)              import matplotlib.pyplot as plt
3: (0)              import pytest
4: (0)              def test_equal():
5: (4)                  gs = gridspec.GridSpec(2, 1)
6: (4)                  assert gs[0, 0] == gs[0, 0]
7: (4)                  assert gs[:, 0] == gs[:, 0]
8: (0)              def test_width_ratios():
9: (4)                  """
10: (4)                 Addresses issue #5835.
11: (4)                 See at https://github.com/matplotlib/matplotlib/issues/5835.
12: (4)                 """
13: (4)                 with pytest.raises(ValueError):
14: (8)                     gridspec.GridSpec(1, 1, width_ratios=[2, 1, 3])
15: (0)             def test_height_ratios():
16: (4)                 """
17: (4)                 Addresses issue #5835.
18: (4)                 See at https://github.com/matplotlib/matplotlib/issues/5835.
19: (4)                 """
20: (4)                 with pytest.raises(ValueError):
21: (8)                     gridspec.GridSpec(1, 1, height_ratios=[2, 1, 3])
22: (0)             def test_repr():
23: (4)                 ss = gridspec.GridSpec(3, 3)[2, 1:3]
24: (4)                 assert repr(ss) == "GridSpec(3, 3)[2:3, 1:3]"
25: (4)                 ss = gridspec.GridSpec(2, 2,
26: (27)                                       height_ratios=(3, 1),
27: (27)                                       width_ratios=(1, 3))
28: (4)                 assert repr(ss) == \
29: (8)                     "GridSpec(2, 2, height_ratios=(3, 1), width_ratios=(1, 3))"
30: (0)             def test_subplotspec_args():
31: (4)                 fig, axs = plt.subplots(1, 2)
```

```
32: (4)                 gs = gridspec.GridSpecFromSubplotSpec(2, 1,
33: (42)
subplot_spec=axs[0].get_subplotspec())
34: (4)                 assert gs.get_topmost_subplotspec() == axs[0].get_subplotspec()
35: (4)                 with pytest.raises(TypeError, match="subplot_spec must be type
SubplotSpec"):
36: (8)                     gs = gridspec.GridSpecFromSubplotSpec(2, 1, subplot_spec=axs[0])
37: (4)                 with pytest.raises(TypeError, match="subplot_spec must be type
SubplotSpec"):
38: (8)                     gs = gridspec.GridSpecFromSubplotSpec(2, 1, subplot_spec=axs)

----------------------------------------

File 190 - test_image.py:

1: (0)              from contextlib import ExitStack
2: (0)              from copy import copy
3: (0)              import functools
4: (0)              import io
5: (0)              import os
6: (0)              from pathlib import Path
7: (0)              import platform
8: (0)              import sys
9: (0)              import urllib.request
10: (0)             import numpy as np
11: (0)             from numpy.testing import assert_array_equal
12: (0)             from PIL import Image
13: (0)             import matplotlib as mpl
14: (0)             from matplotlib import (
15: (4)                 colors, image as mimage, patches, pyplot as plt, style, rcParams)
16: (0)             from matplotlib.image import (AxesImage, BboxImage, FigureImage,
17: (30)                                          NonUniformImage, PcolorImage)
18: (0)             from matplotlib.testing.decorators import check_figures_equal,
image_comparison
19: (0)             from matplotlib.transforms import Bbox, Affine2D, TransformedBbox
20: (0)             import matplotlib.ticker as mticker
21: (0)             import pytest
22: (0)             @image_comparison(['image_interps'], style='mpl20')
23: (0)             def test_image_interps():
24: (4)                 """Make the basic nearest, bilinear and bicubic interps."""
25: (4)                 plt.rcParams['text.kerning_factor'] = 6
26: (4)                 X = np.arange(100).reshape(5, 20)
27: (4)                 fig, (ax1, ax2, ax3) = plt.subplots(3)
28: (4)                 ax1.imshow(X, interpolation='nearest')
29: (4)                 ax1.set_title('three interpolations')
30: (4)                 ax1.set_ylabel('nearest')
31: (4)                 ax2.imshow(X, interpolation='bilinear')
32: (4)                 ax2.set_ylabel('bilinear')
33: (4)                 ax3.imshow(X, interpolation='bicubic')
34: (4)                 ax3.set_ylabel('bicubic')
35: (0)             @image_comparison(['interp_alpha.png'], remove_text=True)
36: (0)             def test_alpha_interp():
37: (4)                 """Test the interpolation of the alpha channel on RGBA images"""
38: (4)                 fig, (axl, axr) = plt.subplots(1, 2)
39: (4)                 img = np.zeros((5, 5, 4))
40: (4)                 img[..., 1] = np.ones((5, 5))
41: (4)                 img[..., 3] = np.tril(np.ones((5, 5), dtype=np.uint8))
42: (4)                 axl.imshow(img, interpolation="none")
43: (4)                 axr.imshow(img, interpolation="bilinear")
44: (0)             @image_comparison(['interp_nearest_vs_none'],
45: (18)                              extensions=['pdf', 'svg'], remove_text=True)
46: (0)             def test_interp_nearest_vs_none():
47: (4)                 """Test the effect of "nearest" and "none" interpolation"""
48: (4)                 rcParams['savefig.dpi'] = 3
49: (4)                 X = np.array([[[218, 165, 32], [122, 103, 238]],
50: (18)                              [[127, 255, 0], [255, 99, 71]]], dtype=np.uint8)
51: (4)                 fig, (ax1, ax2) = plt.subplots(1, 2)
52: (4)                 ax1.imshow(X, interpolation='none')
53: (4)                 ax1.set_title('interpolation none')
```

```
 54: (4)                 ax2.imshow(X, interpolation='nearest')
 55: (4)                 ax2.set_title('interpolation nearest')
 56: (0)             @pytest.mark.parametrize('suppressComposite', [False, True])
 57: (0)             @image_comparison(['figimage'], extensions=['png', 'pdf'])
 58: (0)             def test_figimage(suppressComposite):
 59: (4)                 fig = plt.figure(figsize=(2, 2), dpi=100)
 60: (4)                 fig.suppressComposite = suppressComposite
 61: (4)                 x, y = np.ix_(np.arange(100) / 100.0, np.arange(100) / 100)
 62: (4)                 z = np.sin(x**2 + y**2 - x*y)
 63: (4)                 c = np.sin(20*x**2 + 50*y**2)
 64: (4)                 img = z + c/5
 65: (4)                 fig.figimage(img, xo=0, yo=0, origin='lower')
 66: (4)                 fig.figimage(img[::-1, :], xo=0, yo=100, origin='lower')
 67: (4)                 fig.figimage(img[:, ::-1], xo=100, yo=0, origin='lower')
 68: (4)                 fig.figimage(img[::-1, ::-1], xo=100, yo=100, origin='lower')
 69: (0)             def test_image_python_io():
 70: (4)                 fig, ax = plt.subplots()
 71: (4)                 ax.plot([1, 2, 3])
 72: (4)                 buffer = io.BytesIO()
 73: (4)                 fig.savefig(buffer)
 74: (4)                 buffer.seek(0)
 75: (4)                 plt.imread(buffer)
 76: (0)             @pytest.mark.parametrize(
 77: (4)                 "img_size, fig_size, interpolation",
 78: (4)                 [(5, 2, "hanning"),  # data larger than figure.
 79: (5)                  (5, 5, "nearest"),  # exact resample.
 80: (5)                  (5, 10, "nearest"),  # double sample.
 81: (5)                  (3, 2.9, "hanning"),  # <3 upsample.
 82: (5)                  (3, 9.1, "nearest"),  # >3 upsample.
 83: (5)                  ])
 84: (0)             @check_figures_equal(extensions=['png'])
 85: (0)             def test_imshow_antialiased(fig_test, fig_ref,
 86: (28)                                        img_size, fig_size, interpolation):
 87: (4)                 np.random.seed(19680801)
 88: (4)                 dpi = plt.rcParams["savefig.dpi"]
 89: (4)                 A = np.random.rand(int(dpi * img_size), int(dpi * img_size))
 90: (4)                 for fig in [fig_test, fig_ref]:
 91: (8)                     fig.set_size_inches(fig_size, fig_size)
 92: (4)                 ax = fig_test.subplots()
 93: (4)                 ax.set_position([0, 0, 1, 1])
 94: (4)                 ax.imshow(A, interpolation='antialiased')
 95: (4)                 ax = fig_ref.subplots()
 96: (4)                 ax.set_position([0, 0, 1, 1])
 97: (4)                 ax.imshow(A, interpolation=interpolation)
 98: (0)             @check_figures_equal(extensions=['png'])
 99: (0)             def test_imshow_zoom(fig_test, fig_ref):
100: (4)                 np.random.seed(19680801)
101: (4)                 dpi = plt.rcParams["savefig.dpi"]
102: (4)                 A = np.random.rand(int(dpi * 3), int(dpi * 3))
103: (4)                 for fig in [fig_test, fig_ref]:
104: (8)                     fig.set_size_inches(2.9, 2.9)
105: (4)                 ax = fig_test.subplots()
106: (4)                 ax.imshow(A, interpolation='antialiased')
107: (4)                 ax.set_xlim([10, 20])
108: (4)                 ax.set_ylim([10, 20])
109: (4)                 ax = fig_ref.subplots()
110: (4)                 ax.imshow(A, interpolation='nearest')
111: (4)                 ax.set_xlim([10, 20])
112: (4)                 ax.set_ylim([10, 20])
113: (0)             @check_figures_equal()
114: (0)             def test_imshow_pil(fig_test, fig_ref):
115: (4)                 style.use("default")
116: (4)                 png_path = Path(__file__).parent / "baseline_images/pngsuite/basn3p04.png"
117: (4)                 tiff_path = Path(__file__).parent /
"baseline_images/test_image/uint16.tif"
118: (4)                 axs = fig_test.subplots(2)
119: (4)                 axs[0].imshow(Image.open(png_path))
120: (4)                 axs[1].imshow(Image.open(tiff_path))
121: (4)                 axs = fig_ref.subplots(2)
```

```
(4)                 axs[0].imshow(plt.imread(png_path))
(4)                 axs[1].imshow(plt.imread(tiff_path))
(0)             def test_imread_pil_uint16():
(4)                 img = plt.imread(os.path.join(os.path.dirname(__file__),
(21)                                 'baseline_images', 'test_image', 'uint16.tif'))
(4)                 assert img.dtype == np.uint16
(4)                 assert np.sum(img) == 134184960
(0)             def test_imread_fspath():
(4)                 img = plt.imread(
(8)                     Path(__file__).parent / 'baseline_images/test_image/uint16.tif')
(4)                 assert img.dtype == np.uint16
(4)                 assert np.sum(img) == 134184960
(0)             @pytest.mark.parametrize("fmt", ["png", "jpg", "jpeg", "tiff"])
(0)             def test_imsave(fmt):
(4)                 has_alpha = fmt not in ["jpg", "jpeg"]
(4)                 np.random.seed(1)
(4)                 data = np.random.rand(1856, 2)
(4)                 buff_dpi1 = io.BytesIO()
(4)                 plt.imsave(buff_dpi1, data, format=fmt, dpi=1)
(4)                 buff_dpi100 = io.BytesIO()
(4)                 plt.imsave(buff_dpi100, data, format=fmt, dpi=100)
(4)                 buff_dpi1.seek(0)
(4)                 arr_dpi1 = plt.imread(buff_dpi1, format=fmt)
(4)                 buff_dpi100.seek(0)
(4)                 arr_dpi100 = plt.imread(buff_dpi100, format=fmt)
(4)                 assert arr_dpi1.shape == (1856, 2, 3 + has_alpha)
(4)                 assert arr_dpi100.shape == (1856, 2, 3 + has_alpha)
(4)                 assert_array_equal(arr_dpi1, arr_dpi100)
(0)             @pytest.mark.parametrize("origin", ["upper", "lower"])
(0)             def test_imsave_rgba_origin(origin):
(4)                 buf = io.BytesIO()
(4)                 result = np.zeros((10, 10, 4), dtype='uint8')
(4)                 mimage.imsave(buf, arr=result, format="png", origin=origin)
(0)             @pytest.mark.parametrize("fmt", ["png", "pdf", "ps", "eps", "svg"])
(0)             def test_imsave_fspath(fmt):
(4)                 plt.imsave(Path(os.devnull), np.array([[0, 1]]), format=fmt)
(0)             def test_imsave_color_alpha():
(4)                 np.random.seed(1)
(4)                 for origin in ['lower', 'upper']:
(8)                     data = np.random.rand(16, 16, 4)
(8)                     buff = io.BytesIO()
(8)                     plt.imsave(buff, data, origin=origin, format="png")
(8)                     buff.seek(0)
(8)                     arr_buf = plt.imread(buff)
(8)                     data = (255*data).astype('uint8')
(8)                     if origin == 'lower':
(12)                        data = data[::-1]
(8)                     arr_buf = (255*arr_buf).astype('uint8')
(8)                     assert_array_equal(data, arr_buf)
(0)             def test_imsave_pil_kwargs_png():
(4)                 from PIL.PngImagePlugin import PngInfo
(4)                 buf = io.BytesIO()
(4)                 pnginfo = PngInfo()
(4)                 pnginfo.add_text("Software", "test")
(4)                 plt.imsave(buf, [[0, 1], [2, 3]],
(15)                           format="png", pil_kwargs={"pnginfo": pnginfo})
(4)                 im = Image.open(buf)
(4)                 assert im.info["Software"] == "test"
(0)             def test_imsave_pil_kwargs_tiff():
(4)                 from PIL.TiffTags import TAGS_V2 as TAGS
(4)                 buf = io.BytesIO()
(4)                 pil_kwargs = {"description": "test image"}
(4)                 plt.imsave(buf, [[0, 1], [2, 3]], format="tiff", pil_kwargs=pil_kwargs)
(4)                 assert len(pil_kwargs) == 1
(4)                 im = Image.open(buf)
(4)                 tags = {TAGS[k].name: v for k, v in im.tag_v2.items()}
(4)                 assert tags["ImageDescription"] == "test image"
(0)             @image_comparison(['image_alpha'], remove_text=True)
(0)             def test_image_alpha():
```

```
(4)                 np.random.seed(0)
(4)                 Z = np.random.rand(6, 6)
(4)                 fig, (ax1, ax2, ax3) = plt.subplots(1, 3)
(4)                 ax1.imshow(Z, alpha=1.0, interpolation='none')
(4)                 ax2.imshow(Z, alpha=0.5, interpolation='none')
(4)                 ax3.imshow(Z, alpha=0.5, interpolation='nearest')
(0)             @mpl.style.context('mpl20')
(0)             @check_figures_equal(extensions=['png'])
(0)             def test_imshow_alpha(fig_test, fig_ref):
(4)                 np.random.seed(19680801)
(4)                 rgbf = np.random.rand(6, 6, 3)
(4)                 rgbu = np.uint8(rgbf * 255)
(4)                 ((ax0, ax1), (ax2, ax3)) = fig_test.subplots(2, 2)
(4)                 ax0.imshow(rgbf, alpha=0.5)
(4)                 ax1.imshow(rgbf, alpha=0.75)
(4)                 ax2.imshow(rgbu, alpha=0.5)
(4)                 ax3.imshow(rgbu, alpha=0.75)
(4)                 rgbaf = np.concatenate((rgbf, np.ones((6, 6, 1))), axis=2)
(4)                 rgbau = np.concatenate((rgbu, np.full((6, 6, 1), 255, np.uint8)), axis=2)
(4)                 ((ax0, ax1), (ax2, ax3)) = fig_ref.subplots(2, 2)
(4)                 rgbaf[:, :, 3] = 0.5
(4)                 ax0.imshow(rgbaf)
(4)                 rgbaf[:, :, 3] = 0.75
(4)                 ax1.imshow(rgbaf)
(4)                 rgbau[:, :, 3] = 127
(4)                 ax2.imshow(rgbau)
(4)                 rgbau[:, :, 3] = 191
(4)                 ax3.imshow(rgbau)
(0)             def test_cursor_data():
(4)                 from matplotlib.backend_bases import MouseEvent
(4)                 fig, ax = plt.subplots()
(4)                 im = ax.imshow(np.arange(100).reshape(10, 10), origin='upper')
(4)                 x, y = 4, 4
(4)                 xdisp, ydisp = ax.transData.transform([x, y])
(4)                 event = MouseEvent('motion_notify_event', fig.canvas, xdisp, ydisp)
(4)                 assert im.get_cursor_data(event) == 44
(4)                 x, y = 10.1, 4
(4)                 xdisp, ydisp = ax.transData.transform([x, y])
(4)                 event = MouseEvent('motion_notify_event', fig.canvas, xdisp, ydisp)
(4)                 assert im.get_cursor_data(event) is None
(4)                 ax.clear()
(4)                 im = ax.imshow(np.arange(100).reshape(10, 10), origin='lower')
(4)                 x, y = 4, 4
(4)                 xdisp, ydisp = ax.transData.transform([x, y])
(4)                 event = MouseEvent('motion_notify_event', fig.canvas, xdisp, ydisp)
(4)                 assert im.get_cursor_data(event) == 44
(4)                 fig, ax = plt.subplots()
(4)                 im = ax.imshow(np.arange(100).reshape(10, 10), extent=[0, 0.5, 0, 0.5])
(4)                 x, y = 0.25, 0.25
(4)                 xdisp, ydisp = ax.transData.transform([x, y])
(4)                 event = MouseEvent('motion_notify_event', fig.canvas, xdisp, ydisp)
(4)                 assert im.get_cursor_data(event) == 55
(4)                 x, y = 0.75, 0.25
(4)                 xdisp, ydisp = ax.transData.transform([x, y])
(4)                 event = MouseEvent('motion_notify_event', fig.canvas, xdisp, ydisp)
(4)                 assert im.get_cursor_data(event) is None
(4)                 x, y = 0.01, -0.01
(4)                 xdisp, ydisp = ax.transData.transform([x, y])
(4)                 event = MouseEvent('motion_notify_event', fig.canvas, xdisp, ydisp)
(4)                 assert im.get_cursor_data(event) is None
(4)                 trans = Affine2D().scale(2).rotate(0.5)
(4)                 im = ax.imshow(np.arange(100).reshape(10, 10),
(19)                               transform=trans + ax.transData)
(4)                 x, y = 3, 10
(4)                 xdisp, ydisp = ax.transData.transform([x, y])
(4)                 event = MouseEvent('motion_notify_event', fig.canvas, xdisp, ydisp)
(4)                 assert im.get_cursor_data(event) == 44
(0)             @pytest.mark.parametrize("xy, data", [
(4)                 [[0.5, 0.5], 0 + 0],
```

```
(4)                 [[0.5, 1.5], 0 + 1],
(4)                 [[4.5, 0.5], 16 + 0],
(4)                 [[8.5, 0.5], 16 + 0],
(4)                 [[9.5, 2.5], 81 + 4],
(4)                 [[-1, 0.5], None],
(4)                 [[0.5, -1], None],
(4)                 ]
(0)             )
(0)             def test_cursor_data_nonuniform(xy, data):
(4)                 from matplotlib.backend_bases import MouseEvent
(4)                 x = np.array([0, 1, 4, 9, 16])
(4)                 y = np.array([0, 1, 2, 3, 4])
(4)                 z = x[np.newaxis, :]**2 + y[:, np.newaxis]**2
(4)                 fig, ax = plt.subplots()
(4)                 im = NonUniformImage(ax, extent=(x.min(), x.max(), y.min(), y.max()))
(4)                 im.set_data(x, y, z)
(4)                 ax.add_image(im)
(4)                 ax.set_xlim(x.min() - 2, x.max())
(4)                 ax.set_ylim(y.min() - 2, y.max())
(4)                 xdisp, ydisp = ax.transData.transform(xy)
(4)                 event = MouseEvent('motion_notify_event', fig.canvas, xdisp, ydisp)
(4)                 assert im.get_cursor_data(event) == data, (im.get_cursor_data(event),
data)
(0)             @pytest.mark.parametrize(
(4)                 "data, text", [
(8)                     ([[10001, 10000]], "[10001.000]"),
(8)                     ([[.123, .987]], "[0.123]"),
(8)                     ([[np.nan, 1, 2]], "[]"),
(8)                     ([[1, 1+1e-15]], "[1.0000000000000000]"),
(8)                     ([[-1, -1]], "[-1.0]"),
(8)                     ([[0, 0]], "[0.00]"),
(4)                 ])
(0)             def test_format_cursor_data(data, text):
(4)                 from matplotlib.backend_bases import MouseEvent
(4)                 fig, ax = plt.subplots()
(4)                 im = ax.imshow(data)
(4)                 xdisp, ydisp = ax.transData.transform([0, 0])
(4)                 event = MouseEvent('motion_notify_event', fig.canvas, xdisp, ydisp)
(4)                 assert im.format_cursor_data(im.get_cursor_data(event)) == text
(0)             @image_comparison(['image_clip'], style='mpl20')
(0)             def test_image_clip():
(4)                 d = [[1, 2], [3, 4]]
(4)                 fig, ax = plt.subplots()
(4)                 im = ax.imshow(d)
(4)                 patch = patches.Circle((0, 0), radius=1, transform=ax.transData)
(4)                 im.set_clip_path(patch)
(0)             @image_comparison(['image_cliprect'], style='mpl20')
(0)             def test_image_cliprect():
(4)                 fig, ax = plt.subplots()
(4)                 d = [[1, 2], [3, 4]]
(4)                 im = ax.imshow(d, extent=(0, 5, 0, 5))
(4)                 rect = patches.Rectangle(
(8)                     xy=(1, 1), width=2, height=2, transform=im.axes.transData)
(4)                 im.set_clip_path(rect)
(0)             @image_comparison(['imshow'], remove_text=True, style='mpl20')
(0)             def test_imshow():
(4)                 fig, ax = plt.subplots()
(4)                 arr = np.arange(100).reshape((10, 10))
(4)                 ax.imshow(arr, interpolation="bilinear", extent=(1, 2, 1, 2))
(4)                 ax.set_xlim(0, 3)
(4)                 ax.set_ylim(0, 3)
(0)             @check_figures_equal(extensions=['png'])
(0)             def test_imshow_10_10_1(fig_test, fig_ref):
(4)                 arr = np.arange(100).reshape((10, 10, 1))
(4)                 ax = fig_ref.subplots()
(4)                 ax.imshow(arr[:, :, 0], interpolation="bilinear", extent=(1, 2, 1, 2))
(4)                 ax.set_xlim(0, 3)
(4)                 ax.set_ylim(0, 3)
(4)                 ax = fig_test.subplots()
```

```
(4)                 ax.imshow(arr, interpolation="bilinear", extent=(1, 2, 1, 2))
(4)                 ax.set_xlim(0, 3)
(4)                 ax.set_ylim(0, 3)
(0)             def test_imshow_10_10_2():
(4)                 fig, ax = plt.subplots()
(4)                 arr = np.arange(200).reshape((10, 10, 2))
(4)                 with pytest.raises(TypeError):
(8)                     ax.imshow(arr)
(0)             def test_imshow_10_10_5():
(4)                 fig, ax = plt.subplots()
(4)                 arr = np.arange(500).reshape((10, 10, 5))
(4)                 with pytest.raises(TypeError):
(8)                     ax.imshow(arr)
(0)             @image_comparison(['no_interpolation_origin'], remove_text=True)
(0)             def test_no_interpolation_origin():
(4)                 fig, axs = plt.subplots(2)
(4)                 axs[0].imshow(np.arange(100).reshape((2, 50)), origin="lower",
(18)                              interpolation='none')
(4)                 axs[1].imshow(np.arange(100).reshape((2, 50)), interpolation='none')
(0)             @image_comparison(['image_shift'], remove_text=True, extensions=['pdf',
'svg'])
(0)             def test_image_shift():
(4)                 imgData = [[1 / x + 1 / y for x in range(1, 100)] for y in range(1, 100)]
(4)                 tMin = 734717.945208
(4)                 tMax = 734717.946366
(4)                 fig, ax = plt.subplots()
(4)                 ax.imshow(imgData, norm=colors.LogNorm(), interpolation='none',
(14)                          extent=(tMin, tMax, 1, 100))
(4)                 ax.set_aspect('auto')
(0)             def test_image_edges():
(4)                 fig = plt.figure(figsize=[1, 1])
(4)                 ax = fig.add_axes([0, 0, 1, 1], frameon=False)
(4)                 data = np.tile(np.arange(12), 15).reshape(20, 9)
(4)                 im = ax.imshow(data, origin='upper', extent=[-10, 10, -10, 10],
(19)                               interpolation='none', cmap='gray')
(4)                 x = y = 2
(4)                 ax.set_xlim([-x, x])
(4)                 ax.set_ylim([-y, y])
(4)                 ax.set_xticks([])
(4)                 ax.set_yticks([])
(4)                 buf = io.BytesIO()
(4)                 fig.savefig(buf, facecolor=(0, 1, 0))
(4)                 buf.seek(0)
(4)                 im = plt.imread(buf)
(4)                 r, g, b, a = sum(im[:, 0])
(4)                 r, g, b, a = sum(im[:, -1])
(4)                 assert g != 100, 'Expected a non-green edge - but sadly, it was.'
(0)             @image_comparison(['image_composite_background'],
(18)                              remove_text=True, style='mpl20')
(0)             def test_image_composite_background():
(4)                 fig, ax = plt.subplots()
(4)                 arr = np.arange(12).reshape(4, 3)
(4)                 ax.imshow(arr, extent=[0, 2, 15, 0])
(4)                 ax.imshow(arr, extent=[4, 6, 15, 0])
(4)                 ax.set_facecolor((1, 0, 0, 0.5))
(4)                 ax.set_xlim([0, 12])
(0)             @image_comparison(['image_composite_alpha'], remove_text=True)
(0)             def test_image_composite_alpha():
(4)                 """
(4)                 Tests that the alpha value is recognized and correctly applied in the
(4)                 process of compositing images together.
(4)                 """
(4)                 fig, ax = plt.subplots()
(4)                 arr = np.zeros((11, 21, 4))
(4)                 arr[:, :, 0] = 1
(4)                 arr[:, :, 3] = np.concatenate(
(8)                     (np.arange(0, 1.1, 0.1), np.arange(0, 1, 0.1)[::-1]))
(4)                 arr2 = np.zeros((21, 11, 4))
(4)                 arr2[:, :, 0] = 1
```

```
(4)                 arr2[:, :, 1] = 1
(4)                 arr2[:, :, 3] = np.concatenate(
(8)                     (np.arange(0, 1.1, 0.1), np.arange(0, 1, 0.1)[::-1]))[:, np.newaxis]
(4)                 ax.imshow(arr, extent=[1, 2, 5, 0], alpha=0.3)
(4)                 ax.imshow(arr, extent=[2, 3, 5, 0], alpha=0.6)
(4)                 ax.imshow(arr, extent=[3, 4, 5, 0])
(4)                 ax.imshow(arr2, extent=[0, 5, 1, 2])
(4)                 ax.imshow(arr2, extent=[0, 5, 2, 3], alpha=0.6)
(4)                 ax.imshow(arr2, extent=[0, 5, 3, 4], alpha=0.3)
(4)                 ax.set_facecolor((0, 0.5, 0, 1))
(4)                 ax.set_xlim([0, 5])
(4)                 ax.set_ylim([5, 0])
(0)             @check_figures_equal(extensions=["pdf"])
(0)             def test_clip_path_disables_compositing(fig_test, fig_ref):
(4)                 t = np.arange(9).reshape((3, 3))
(4)                 for fig in [fig_test, fig_ref]:
(8)                     ax = fig.add_subplot()
(8)                     ax.imshow(t, clip_path=(mpl.path.Path([(0, 0), (0, 1), (1, 0)]),
(32)                                             ax.transData))
(8)                     ax.imshow(t, clip_path=(mpl.path.Path([(1, 1), (1, 2), (2, 1)]),
(32)                                             ax.transData))
(4)                 fig_ref.suppressComposite = True
(0)             @image_comparison(['rasterize_10dpi'],
(18)                              extensions=['pdf', 'svg'], remove_text=True, style='mpl20')
(0)             def test_rasterize_dpi():
(4)                 img = np.asarray([[1, 2], [3, 4]])
(4)                 fig, axs = plt.subplots(1, 3, figsize=(3, 1))
(4)                 axs[0].imshow(img)
(4)                 axs[1].plot([0, 1], [0, 1], linewidth=20., rasterized=True)
(4)                 axs[1].set(xlim=(0, 1), ylim=(-1, 2))
(4)                 axs[2].plot([0, 1], [0, 1], linewidth=20.)
(4)                 axs[2].set(xlim=(0, 1), ylim=(-1, 2))
(4)                 for ax in axs:
(8)                     ax.set_xticks([])
(8)                     ax.set_yticks([])
(8)                     ax.spines[:].set_visible(False)
(4)                 rcParams['savefig.dpi'] = 10
(0)             @image_comparison(['bbox_image_inverted'], remove_text=True, style='mpl20')
(0)             def test_bbox_image_inverted():
(4)                 image = np.arange(100).reshape((10, 10))
(4)                 fig, ax = plt.subplots()
(4)                 bbox_im = BboxImage(
(8)                     TransformedBbox(Bbox([[100, 100], [0, 0]]), ax.transData),
(8)                     interpolation='nearest')
(4)                 bbox_im.set_data(image)
(4)                 bbox_im.set_clip_on(False)
(4)                 ax.set_xlim(0, 100)
(4)                 ax.set_ylim(0, 100)
(4)                 ax.add_artist(bbox_im)
(4)                 image = np.identity(10)
(4)                 bbox_im = BboxImage(TransformedBbox(Bbox([[0.1, 0.2], [0.3, 0.25]]),
(40)                                                 ax.figure.transFigure),
(24)                                 interpolation='nearest')
(4)                 bbox_im.set_data(image)
(4)                 bbox_im.set_clip_on(False)
(4)                 ax.add_artist(bbox_im)
(0)             def test_get_window_extent_for_AxisImage():
(4)                 im = np.array([[0.25, 0.75, 1.0, 0.75], [0.1, 0.65, 0.5, 0.4],
(19)                               [0.6, 0.3, 0.0, 0.2], [0.7, 0.9, 0.4, 0.6]])
(4)                 fig, ax = plt.subplots(figsize=(10, 10), dpi=100)
(4)                 ax.set_position([0, 0, 1, 1])
(4)                 ax.set_xlim(0, 1)
(4)                 ax.set_ylim(0, 1)
(4)                 im_obj = ax.imshow(
(8)                     im, extent=[0.4, 0.7, 0.2, 0.9], interpolation='nearest')
(4)                 fig.canvas.draw()
(4)                 renderer = fig.canvas.renderer
(4)                 im_bbox = im_obj.get_window_extent(renderer)
(4)                 assert_array_equal(im_bbox.get_points(), [[400, 200], [700, 900]])
```

```
465: (4)                fig, ax = plt.subplots(figsize=(10, 10), dpi=100)
466: (4)                ax.set_position([0, 0, 1, 1])
467: (4)                ax.set_xlim(1, 2)
468: (4)                ax.set_ylim(0, 1)
469: (4)                im_obj = ax.imshow(
470: (8)                    im, extent=[0.4, 0.7, 0.2, 0.9], interpolation='nearest',
471: (8)                    transform=ax.transAxes)
472: (4)                fig.canvas.draw()
473: (4)                renderer = fig.canvas.renderer
474: (4)                im_bbox = im_obj.get_window_extent(renderer)
475: (4)                assert_array_equal(im_bbox.get_points(), [[400, 200], [700, 900]])
476: (0)            @image_comparison(['zoom_and_clip_upper_origin.png'],
477: (18)                              remove_text=True, style='mpl20')
478: (0)            def test_zoom_and_clip_upper_origin():
479: (4)                image = np.arange(100)
480: (4)                image = image.reshape((10, 10))
481: (4)                fig, ax = plt.subplots()
482: (4)                ax.imshow(image)
483: (4)                ax.set_ylim(2.0, -0.5)
484: (4)                ax.set_xlim(-0.5, 2.0)
485: (0)            def test_nonuniformimage_setcmap():
486: (4)                ax = plt.gca()
487: (4)                im = NonUniformImage(ax)
488: (4)                im.set_cmap('Blues')
489: (0)            def test_nonuniformimage_setnorm():
490: (4)                ax = plt.gca()
491: (4)                im = NonUniformImage(ax)
492: (4)                im.set_norm(plt.Normalize())
493: (0)            def test_jpeg_2d():
494: (4)                imd = np.ones((10, 10), dtype='uint8')
495: (4)                for i in range(10):
496: (8)                    imd[i, :] = np.linspace(0.0, 1.0, 10) * 255
497: (4)                im = Image.new('L', (10, 10))
498: (4)                im.putdata(imd.flatten())
499: (4)                fig, ax = plt.subplots()
500: (4)                ax.imshow(im)
501: (0)            def test_jpeg_alpha():
502: (4)                plt.figure(figsize=(1, 1), dpi=300)
503: (4)                im = np.zeros((300, 300, 4), dtype=float)
504: (4)                im[..., 3] = np.linspace(0.0, 1.0, 300)
505: (4)                plt.figimage(im)
506: (4)                buff = io.BytesIO()
507: (4)                plt.savefig(buff, facecolor="red", format='jpg', dpi=300)
508: (4)                buff.seek(0)
509: (4)                image = Image.open(buff)
510: (4)                num_colors = len(image.getcolors(256))
511: (4)                assert 175 <= num_colors <= 210
512: (4)                corner_pixel = image.getpixel((0, 0))
513: (4)                assert corner_pixel == (254, 0, 0)
514: (0)            def test_axesimage_setdata():
515: (4)                ax = plt.gca()
516: (4)                im = AxesImage(ax)
517: (4)                z = np.arange(12, dtype=float).reshape((4, 3))
518: (4)                im.set_data(z)
519: (4)                z[0, 0] = 9.9
520: (4)                assert im._A[0, 0] == 0, 'value changed'
521: (0)            def test_figureimage_setdata():
522: (4)                fig = plt.gcf()
523: (4)                im = FigureImage(fig)
524: (4)                z = np.arange(12, dtype=float).reshape((4, 3))
525: (4)                im.set_data(z)
526: (4)                z[0, 0] = 9.9
527: (4)                assert im._A[0, 0] == 0, 'value changed'
528: (0)            @pytest.mark.parametrize(
529: (4)                "image_cls,x,y,a", [
530: (8)                    (NonUniformImage,
531: (9)                     np.arange(3.), np.arange(4.), np.arange(12.).reshape((4, 3))),
532: (8)                    (PcolorImage,
533: (9)                     np.arange(3.), np.arange(4.), np.arange(6.).reshape((3, 2))),
```

```
534: (4)                ])
535: (0)            def test_setdata_xya(image_cls, x, y, a):
536: (4)                ax = plt.gca()
537: (4)                im = image_cls(ax)
538: (4)                im.set_data(x, y, a)
539: (4)                x[0] = y[0] = a[0, 0] = 9.9
540: (4)                assert im._A[0, 0] == im._Ax[0] == im._Ay[0] == 0, 'value changed'
541: (4)                im.set_data(x, y, a.reshape((*a.shape, -1)))  # Just a smoketest.
542: (0)            def test_minimized_rasterized():
543: (4)                from xml.etree import ElementTree
544: (4)                np.random.seed(0)
545: (4)                data = np.random.rand(10, 10)
546: (4)                fig, ax = plt.subplots(1, 2)
547: (4)                p1 = ax[0].pcolormesh(data)
548: (4)                p2 = ax[1].pcolormesh(data)
549: (4)                plt.colorbar(p1, ax=ax[0])
550: (4)                plt.colorbar(p2, ax=ax[1])
551: (4)                buff = io.BytesIO()
552: (4)                plt.savefig(buff, format='svg')
553: (4)                buff = io.BytesIO(buff.getvalue())
554: (4)                tree = ElementTree.parse(buff)
555: (4)                width = None
556: (4)                for image in tree.iter('image'):
557: (8)                    if width is None:
558: (12)                       width = image['width']
559: (8)                    else:
560: (12)                       if image['width'] != width:
561: (16)                           assert False
562: (0)            def test_load_from_url():
563: (4)                path = Path(__file__).parent / "baseline_images/pngsuite/basn3p04.png"
564: (4)                url = ('file:'
565: (11)                      + ('///' if sys.platform == 'win32' else '')
566: (11)                      + path.resolve().as_posix())
567: (4)                with pytest.raises(ValueError, match="Please open the URL"):
568: (8)                    plt.imread(url)
569: (4)                with urllib.request.urlopen(url) as file:
570: (8)                    plt.imread(file)
571: (0)            @image_comparison(['log_scale_image'], remove_text=True)
572: (0)            def test_log_scale_image():
573: (4)                Z = np.zeros((10, 10))
574: (4)                Z[::2] = 1
575: (4)                fig, ax = plt.subplots()
576: (4)                ax.imshow(Z, extent=[1, 100, 1, 100], cmap='viridis', vmax=1, vmin=-1,
577: (14)                         aspect='auto')
578: (4)                ax.set(yscale='log')
579: (0)            @image_comparison(['rotate_image'], remove_text=True)
580: (0)            def test_rotate_image():
581: (4)                delta = 0.25
582: (4)                x = y = np.arange(-3.0, 3.0, delta)
583: (4)                X, Y = np.meshgrid(x, y)
584: (4)                Z1 = np.exp(-(X**2 + Y**2) / 2) / (2 * np.pi)
585: (4)                Z2 = (np.exp(-(((X - 1) / 1.5)**2 + ((Y - 1) / 0.5)**2) / 2) /
586: (10)                     (2 * np.pi * 0.5 * 1.5))
587: (4)                Z = Z2 - Z1  # difference of Gaussians
588: (4)                fig, ax1 = plt.subplots(1, 1)
589: (4)                im1 = ax1.imshow(Z, interpolation='none', cmap='viridis',
590: (21)                                origin='lower',
591: (21)                                extent=[-2, 4, -3, 2], clip_on=True)
592: (4)                trans_data2 = Affine2D().rotate_deg(30) + ax1.transData
593: (4)                im1.set_transform(trans_data2)
594: (4)                x1, x2, y1, y2 = im1.get_extent()
595: (4)                ax1.plot([x1, x2, x2, x1, x1], [y1, y1, y2, y2, y1], "r--", lw=3,
596: (13)                         transform=trans_data2)
597: (4)                ax1.set_xlim(2, 5)
598: (4)                ax1.set_ylim(0, 4)
599: (0)            def test_image_preserve_size():
600: (4)                buff = io.BytesIO()
601: (4)                im = np.zeros((481, 321))
602: (4)                plt.imsave(buff, im, format="png")
```

```
603: (4)                buff.seek(0)
604: (4)                img = plt.imread(buff)
605: (4)                assert img.shape[:2] == im.shape
606: (0)            def test_image_preserve_size2():
607: (4)                n = 7
608: (4)                data = np.identity(n, float)
609: (4)                fig = plt.figure(figsize=(n, n), frameon=False)
610: (4)                ax = fig.add_axes((0.0, 0.0, 1.0, 1.0))
611: (4)                ax.set_axis_off()
612: (4)                ax.imshow(data, interpolation='nearest', origin='lower', aspect='auto')
613: (4)                buff = io.BytesIO()
614: (4)                fig.savefig(buff, dpi=1)
615: (4)                buff.seek(0)
616: (4)                img = plt.imread(buff)
617: (4)                assert img.shape == (7, 7, 4)
618: (4)                assert_array_equal(np.asarray(img[:, :, 0], bool),
619: (23)                                   np.identity(n, bool)[::-1])
620: (0)            @image_comparison(['mask_image_over_under.png'], remove_text=True, tol=1.0)
621: (0)            def test_mask_image_over_under():
622: (4)                plt.rcParams['pcolormesh.snap'] = False
623: (4)                delta = 0.025
624: (4)                x = y = np.arange(-3.0, 3.0, delta)
625: (4)                X, Y = np.meshgrid(x, y)
626: (4)                Z1 = np.exp(-(X**2 + Y**2) / 2) / (2 * np.pi)
627: (4)                Z2 = (np.exp(-(((X - 1) / 1.5)**2 + ((Y - 1) / 0.5)**2) / 2) /
628: (10)                     (2 * np.pi * 0.5 * 1.5))
629: (4)                Z = 10*(Z2 - Z1)  # difference of Gaussians
630: (4)                palette = plt.cm.gray.with_extremes(over='r', under='g', bad='b')
631: (4)                Zm = np.ma.masked_where(Z > 1.2, Z)
632: (4)                fig, (ax1, ax2) = plt.subplots(1, 2)
633: (4)                im = ax1.imshow(Zm, interpolation='bilinear',
634: (20)                               cmap=palette,
635: (20)                               norm=colors.Normalize(vmin=-1.0, vmax=1.0, clip=False),
636: (20)                               origin='lower', extent=[-3, 3, -3, 3])
637: (4)                ax1.set_title('Green=low, Red=high, Blue=bad')
638: (4)                fig.colorbar(im, extend='both', orientation='horizontal',
639: (17)                            ax=ax1, aspect=10)
640: (4)                im = ax2.imshow(Zm, interpolation='nearest',
641: (20)                               cmap=palette,
642: (20)                               norm=colors.BoundaryNorm([-1, -0.5, -0.2, 0, 0.2, 0.5, 1],
643: (45)                                                        ncolors=256, clip=False),
644: (20)                               origin='lower', extent=[-3, 3, -3, 3])
645: (4)                ax2.set_title('With BoundaryNorm')
646: (4)                fig.colorbar(im, extend='both', spacing='proportional',
647: (17)                            orientation='horizontal', ax=ax2, aspect=10)
648: (0)            @image_comparison(['mask_image'], remove_text=True)
649: (0)            def test_mask_image():
650: (4)                fig, (ax1, ax2) = plt.subplots(1, 2)
651: (4)                A = np.ones((5, 5))
652: (4)                A[1:2, 1:2] = np.nan
653: (4)                ax1.imshow(A, interpolation='nearest')
654: (4)                A = np.zeros((5, 5), dtype=bool)
655: (4)                A[1:2, 1:2] = True
656: (4)                A = np.ma.masked_array(np.ones((5, 5), dtype=np.uint16), A)
657: (4)                ax2.imshow(A, interpolation='nearest')
658: (0)            def test_mask_image_all():
659: (4)                data = np.full((2, 2), np.nan)
660: (4)                fig, ax = plt.subplots()
661: (4)                ax.imshow(data)
662: (4)                fig.canvas.draw_idle()  # would emit a warning
663: (0)            @image_comparison(['imshow_endianess.png'], remove_text=True)
664: (0)            def test_imshow_endianess():
665: (4)                x = np.arange(10)
666: (4)                X, Y = np.meshgrid(x, x)
667: (4)                Z = np.hypot(X - 5, Y - 5)
668: (4)                fig, (ax1, ax2) = plt.subplots(1, 2)
669: (4)                kwargs = dict(origin="lower", interpolation='nearest', cmap='viridis')
670: (4)                ax1.imshow(Z.astype('<f8'), **kwargs)
671: (4)                ax2.imshow(Z.astype('>f8'), **kwargs)
```

```
(0)            @image_comparison(['imshow_masked_interpolation'],
(18)                             tol=0 if platform.machine() == 'x86_64' else 0.01,
(18)                             remove_text=True, style='mpl20')
(0)            def test_imshow_masked_interpolation():
(4)                cmap = mpl.colormaps['viridis'].with_extremes(over='r', under='b',
bad='k')
(4)                N = 20
(4)                n = colors.Normalize(vmin=0, vmax=N*N-1)
(4)                data = np.arange(N*N, dtype=float).reshape(N, N)
(4)                data[5, 5] = -1
(4)                data[15, 5] = 1e5
(4)                data[15, 15] = np.inf
(4)                mask = np.zeros_like(data).astype('bool')
(4)                mask[5, 15] = True
(4)                data = np.ma.masked_array(data, mask)
(4)                fig, ax_grid = plt.subplots(3, 6)
(4)                interps = sorted(mimage._interpd_)
(4)                interps.remove('antialiased')
(4)                for interp, ax in zip(interps, ax_grid.ravel()):
(8)                    ax.set_title(interp)
(8)                    ax.imshow(data, norm=n, cmap=cmap, interpolation=interp)
(8)                    ax.axis('off')
(0)            def test_imshow_no_warn_invalid():
(4)                plt.imshow([[1, 2], [3, np.nan]])  # Check that no warning is emitted.
(0)            @pytest.mark.parametrize(
(4)                'dtype', [np.dtype(s) for s in 'u2 u4 i2 i4 i8 f4 f8'.split()])
(0)            def test_imshow_clips_rgb_to_valid_range(dtype):
(4)                arr = np.arange(300, dtype=dtype).reshape((10, 10, 3))
(4)                if dtype.kind != 'u':
(8)                    arr -= 10
(4)                too_low = arr < 0
(4)                too_high = arr > 255
(4)                if dtype.kind == 'f':
(8)                    arr = arr / 255
(4)                _, ax = plt.subplots()
(4)                out = ax.imshow(arr).get_array()
(4)                assert (out[too_low] == 0).all()
(4)                if dtype.kind == 'f':
(8)                    assert (out[too_high] == 1).all()
(8)                    assert out.dtype.kind == 'f'
(4)                else:
(8)                    assert (out[too_high] == 255).all()
(8)                    assert out.dtype == np.uint8
(0)            @image_comparison(['imshow_flatfield.png'], remove_text=True, style='mpl20')
(0)            def test_imshow_flatfield():
(4)                fig, ax = plt.subplots()
(4)                im = ax.imshow(np.ones((5, 5)), interpolation='nearest')
(4)                im.set_clim(.5, 1.5)
(0)            @image_comparison(['imshow_bignumbers.png'], remove_text=True, style='mpl20')
(0)            def test_imshow_bignumbers():
(4)                rcParams['image.interpolation'] = 'nearest'
(4)                fig, ax = plt.subplots()
(4)                img = np.array([[1, 2, 1e12], [3, 1, 4]], dtype=np.uint64)
(4)                pc = ax.imshow(img)
(4)                pc.set_clim(0, 5)
(0)            @image_comparison(['imshow_bignumbers_real.png'],
(18)                             remove_text=True, style='mpl20')
(0)            def test_imshow_bignumbers_real():
(4)                rcParams['image.interpolation'] = 'nearest'
(4)                fig, ax = plt.subplots()
(4)                img = np.array([[2., 1., 1.e22], [4., 1., 3.]])
(4)                pc = ax.imshow(img)
(4)                pc.set_clim(0, 5)
(0)            @pytest.mark.parametrize(
(4)                "make_norm",
(4)                [colors.Normalize,
(5)                 colors.LogNorm,
(5)                 lambda: colors.SymLogNorm(1),
(5)                 lambda: colors.PowerNorm(1)])
```

```
(0)             def test_empty_imshow(make_norm):
(4)                 fig, ax = plt.subplots()
(4)                 with pytest.warns(UserWarning,
(22)                                  match="Attempting to set identical low and high xlims"):
(8)                     im = ax.imshow([[]], norm=make_norm())
(4)                 im.set_extent([-5, 5, -5, 5])
(4)                 fig.canvas.draw()
(4)                 with pytest.raises(RuntimeError):
(8)                     im.make_image(fig.canvas.get_renderer())
(0)             def test_imshow_float16():
(4)                 fig, ax = plt.subplots()
(4)                 ax.imshow(np.zeros((3, 3), dtype=np.float16))
(4)                 fig.canvas.draw()
(0)             def test_imshow_float128():
(4)                 fig, ax = plt.subplots()
(4)                 ax.imshow(np.zeros((3, 3), dtype=np.longdouble))
(4)                 with (ExitStack() if np.can_cast(np.longdouble, np.float64, "equiv")
(10)                      else pytest.warns(UserWarning)):
(8)                     fig.canvas.draw()
(0)             def test_imshow_bool():
(4)                 fig, ax = plt.subplots()
(4)                 ax.imshow(np.array([[True, False], [False, True]], dtype=bool))
(0)             def test_full_invalid():
(4)                 fig, ax = plt.subplots()
(4)                 ax.imshow(np.full((10, 10), np.nan))
(4)                 fig.canvas.draw()
(0)             @pytest.mark.parametrize("fmt,counted",
(25)                                     [("ps", b" colorimage"), ("svg", b"<image")])
(0)             @pytest.mark.parametrize("composite_image,count", [(True, 1), (False, 2)])
(0)             def test_composite(fmt, counted, composite_image, count):
(4)                 X, Y = np.meshgrid(np.arange(-5, 5, 1), np.arange(-5, 5, 1))
(4)                 Z = np.sin(Y ** 2)
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_xlim(0, 3)
(4)                 ax.imshow(Z, extent=[0, 1, 0, 1])
(4)                 ax.imshow(Z[::-1], extent=[2, 3, 0, 1])
(4)                 plt.rcParams['image.composite_image'] = composite_image
(4)                 buf = io.BytesIO()
(4)                 fig.savefig(buf, format=fmt)
(4)                 assert buf.getvalue().count(counted) == count
(0)             def test_relim():
(4)                 fig, ax = plt.subplots()
(4)                 ax.imshow([[0]], extent=(0, 1, 0, 1))
(4)                 ax.relim()
(4)                 ax.autoscale()
(4)                 assert ax.get_xlim() == ax.get_ylim() == (0, 1)
(0)             def test_unclipped():
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_axis_off()
(4)                 im = ax.imshow([[0, 0], [0, 0]], aspect="auto", extent=(-10, 10, -10, 10),
(19)                               cmap='gray', clip_on=False)
(4)                 ax.set(xlim=(0, 1), ylim=(0, 1))
(4)                 fig.canvas.draw()
(4)                 assert (np.array(fig.canvas.buffer_rgba())[..., :3] == 0).all()
(0)             def test_respects_bbox():
(4)                 fig, axs = plt.subplots(2)
(4)                 for ax in axs:
(8)                     ax.set_axis_off()
(4)                 im = axs[1].imshow([[0, 1], [2, 3]], aspect="auto", extent=(0, 1, 0, 1))
(4)                 im.set_clip_path(None)
(4)                 im.set_clip_box(axs[0].bbox)
(4)                 buf_before = io.BytesIO()
(4)                 fig.savefig(buf_before, format="rgba")
(4)                 assert {*buf_before.getvalue()} == {0xff}  # All white.
(4)                 axs[1].set(ylim=(-1, 0))
(4)                 buf_after = io.BytesIO()
(4)                 fig.savefig(buf_after, format="rgba")
(4)                 assert buf_before.getvalue() != buf_after.getvalue()  # Not all white.
(0)             def test_image_cursor_formatting():
```

```
809: (4)                fig, ax = plt.subplots()
810: (4)                im = ax.imshow(np.zeros((4, 4)))
811: (4)                data = np.ma.masked_array([0], mask=[True])
812: (4)                assert im.format_cursor_data(data) == '[]'
813: (4)                data = np.ma.masked_array([0], mask=[False])
814: (4)                assert im.format_cursor_data(data) == '[0]'
815: (4)                data = np.nan
816: (4)                assert im.format_cursor_data(data) == '[nan]'
817: (0)            @check_figures_equal()
818: (0)            def test_image_array_alpha(fig_test, fig_ref):
819: (4)                """Per-pixel alpha channel test."""
820: (4)                x = np.linspace(0, 1)
821: (4)                xx, yy = np.meshgrid(x, x)
822: (4)                zz = np.exp(- 3 * ((xx - 0.5) ** 2) + (yy - 0.7 ** 2))
823: (4)                alpha = zz / zz.max()
824: (4)                cmap = mpl.colormaps['viridis']
825: (4)                ax = fig_test.add_subplot()
826: (4)                ax.imshow(zz, alpha=alpha, cmap=cmap, interpolation='nearest')
827: (4)                ax = fig_ref.add_subplot()
828: (4)                rgba = cmap(colors.Normalize()(zz))
829: (4)                rgba[..., -1] = alpha
830: (4)                ax.imshow(rgba, interpolation='nearest')
831: (0)            def test_image_array_alpha_validation():
832: (4)                with pytest.raises(TypeError, match="alpha must be a float, two-d"):
833: (8)                    plt.imshow(np.zeros((2, 2)), alpha=[1, 1])
834: (0)            @mpl.style.context('mpl20')
835: (0)            def test_exact_vmin():
836: (4)                cmap = copy(mpl.colormaps["autumn_r"])
837: (4)                cmap.set_under(color="lightgrey")
838: (4)                fig = plt.figure(figsize=(1.9, 0.1), dpi=100)
839: (4)                ax = fig.add_axes([0, 0, 1, 1])
840: (4)                data = np.array(
841: (8)                    [[-1, -1, -1, 0, 0, 0, 0, 43, 79, 95, 66, 1, -1, -1, -1, 0, 0, 0,
34]],
842: (8)                    dtype=float,
843: (4)                )
844: (4)                im = ax.imshow(data, aspect="auto", cmap=cmap, vmin=0, vmax=100)
845: (4)                ax.axis("off")
846: (4)                fig.canvas.draw()
847: (4)                from_image = im.make_image(fig.canvas.renderer)[0][0]
848: (4)                direct_computation = (
849: (8)                    im.cmap(im.norm((data * ([[1]] * 10)).T.ravel())) * 255
850: (4)                ).astype(int)
851: (4)                assert np.all(from_image == direct_computation)
852: (0)            @image_comparison(['image_placement'], extensions=['svg', 'pdf'],
853: (18)                             remove_text=True, style='mpl20')
854: (0)            def test_image_placement():
855: (4)                """
856: (4)                The red box should line up exactly with the outside of the image.
857: (4)                """
858: (4)                fig, ax = plt.subplots()
859: (4)                ax.plot([0, 0, 1, 1, 0], [0, 1, 1, 0, 0], color='r', lw=0.1)
860: (4)                np.random.seed(19680801)
861: (4)                ax.imshow(np.random.randn(16, 16), cmap='Blues', extent=(0, 1, 0, 1),
862: (14)                         interpolation='none', vmin=-1, vmax=1)
863: (4)                ax.set_xlim(-0.1, 1+0.1)
864: (4)                ax.set_ylim(-0.1, 1+0.1)
865: (0)            class QuantityND(np.ndarray):
866: (4)                def __new__(cls, input_array, units):
867: (8)                    obj = np.asarray(input_array).view(cls)
868: (8)                    obj.units = units
869: (8)                    return obj
870: (4)                def __array_finalize__(self, obj):
871: (8)                    self.units = getattr(obj, "units", None)
872: (4)                def __getitem__(self, item):
873: (8)                    units = getattr(self, "units", None)
874: (8)                    ret = super().__getitem__(item)
875: (8)                    if isinstance(ret, QuantityND) or units is not None:
876: (12)                       ret = QuantityND(ret, units)
```

```
(8)                 return ret
(4)             def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
(8)                 func = getattr(ufunc, method)
(8)                 if "out" in kwargs:
(12)                    return NotImplemented
(8)                 if len(inputs) == 1:
(12)                    i0 = inputs[0]
(12)                    unit = getattr(i0, "units", "dimensionless")
(12)                    out_arr = func(np.asarray(i0), **kwargs)
(8)                 elif len(inputs) == 2:
(12)                    i0 = inputs[0]
(12)                    i1 = inputs[1]
(12)                    u0 = getattr(i0, "units", "dimensionless")
(12)                    u1 = getattr(i1, "units", "dimensionless")
(12)                    u0 = u1 if u0 is None else u0
(12)                    u1 = u0 if u1 is None else u1
(12)                    if ufunc in [np.add, np.subtract]:
(16)                        if u0 != u1:
(20)                            raise ValueError
(16)                        unit = u0
(12)                    elif ufunc == np.multiply:
(16)                        unit = f"{u0}*{u1}"
(12)                    elif ufunc == np.divide:
(16)                        unit = f"{u0}/({u1})"
(12)                    elif ufunc in (np.greater, np.greater_equal,
(27)                                   np.equal, np.not_equal,
(27)                                   np.less, np.less_equal):
(16)                        unit = None
(12)                    else:
(16)                        return NotImplemented
(12)                    out_arr = func(i0.view(np.ndarray), i1.view(np.ndarray), **kwargs)
(8)                 else:
(12)                    return NotImplemented
(8)                 if unit is None:
(12)                    out_arr = np.array(out_arr)
(8)                 else:
(12)                    out_arr = QuantityND(out_arr, unit)
(8)                 return out_arr
(4)             @property
(4)             def v(self):
(8)                 return self.view(np.ndarray)
(0)         def test_quantitynd():
(4)             q = QuantityND([1, 2], "m")
(4)             q0, q1 = q[:]
(4)             assert np.all(q.v == np.asarray([1, 2]))
(4)             assert q.units == "m"
(4)             assert np.all((q0 + q1).v == np.asarray([3]))
(4)             assert (q0 * q1).units == "m*m"
(4)             assert (q1 / q0).units == "m/(m)"
(4)             with pytest.raises(ValueError):
(8)                 q0 + QuantityND(1, "s")
(0)         def test_imshow_quantitynd():
(4)             arr = QuantityND(np.ones((2, 2)), "m")
(4)             fig, ax = plt.subplots()
(4)             ax.imshow(arr)
(4)             fig.canvas.draw()
(0)         @check_figures_equal(extensions=['png'])
(0)         def test_norm_change(fig_test, fig_ref):
(4)             data = np.full((5, 5), 1, dtype=np.float64)
(4)             data[0:2, :] = -1
(4)             masked_data = np.ma.array(data, mask=False)
(4)             masked_data.mask[0:2, 0:2] = True
(4)             cmap = mpl.colormaps['viridis'].with_extremes(under='w')
(4)             ax = fig_test.subplots()
(4)             im = ax.imshow(data, norm=colors.LogNorm(vmin=0.5, vmax=1),
(19)                           extent=(0, 5, 0, 5), interpolation='nearest', cmap=cmap)
(4)             im.set_norm(colors.Normalize(vmin=-2, vmax=2))
(4)             im = ax.imshow(masked_data, norm=colors.LogNorm(vmin=0.5, vmax=1),
(19)                           extent=(5, 10, 5, 10), interpolation='nearest', cmap=cmap)
```

```
(4)                 im.set_norm(colors.Normalize(vmin=-2, vmax=2))
(4)                 ax.set(xlim=(0, 10), ylim=(0, 10))
(4)                 ax = fig_ref.subplots()
(4)                 ax.imshow(data, norm=colors.Normalize(vmin=-2, vmax=2),
(14)                          extent=(0, 5, 0, 5), interpolation='nearest', cmap=cmap)
(4)                 ax.imshow(masked_data, norm=colors.Normalize(vmin=-2, vmax=2),
(14)                          extent=(5, 10, 5, 10), interpolation='nearest', cmap=cmap)
(4)                 ax.set(xlim=(0, 10), ylim=(0, 10))
(0)             @pytest.mark.parametrize('x', [-1, 1])
(0)             @check_figures_equal(extensions=['png'])
(0)             def test_huge_range_log(fig_test, fig_ref, x):
(4)                 data = np.full((5, 5), x, dtype=np.float64)
(4)                 data[0:2, :] = 1E20
(4)                 ax = fig_test.subplots()
(4)                 ax.imshow(data, norm=colors.LogNorm(vmin=1, vmax=data.max()),
(14)                          interpolation='nearest', cmap='viridis')
(4)                 data = np.full((5, 5), x, dtype=np.float64)
(4)                 data[0:2, :] = 1000
(4)                 ax = fig_ref.subplots()
(4)                 cmap = mpl.colormaps['viridis'].with_extremes(under='w')
(4)                 ax.imshow(data, norm=colors.Normalize(vmin=1, vmax=data.max()),
(14)                          interpolation='nearest', cmap=cmap)
(0)             @check_figures_equal()
(0)             def test_spy_box(fig_test, fig_ref):
(4)                 ax_test = fig_test.subplots(1, 3)
(4)                 ax_ref = fig_ref.subplots(1, 3)
(4)                 plot_data = (
(8)                     [[1, 1], [1, 1]],
(8)                     [[0, 0], [0, 0]],
(8)                     [[0, 1], [1, 0]],
(4)                 )
(4)                 plot_titles = ["ones", "zeros", "mixed"]
(4)                 for i, (z, title) in enumerate(zip(plot_data, plot_titles)):
(8)                     ax_test[i].set_title(title)
(8)                     ax_test[i].spy(z)
(8)                     ax_ref[i].set_title(title)
(8)                     ax_ref[i].imshow(z, interpolation='nearest',
(28)                                     aspect='equal', origin='upper', cmap='Greys',
(28)                                     vmin=0, vmax=1)
(8)                     ax_ref[i].set_xlim(-0.5, 1.5)
(8)                     ax_ref[i].set_ylim(1.5, -0.5)
(8)                     ax_ref[i].xaxis.tick_top()
(8)                     ax_ref[i].title.set_y(1.05)
(8)                     ax_ref[i].xaxis.set_ticks_position('both')
(8)                     ax_ref[i].xaxis.set_major_locator(
(12)                        mticker.MaxNLocator(nbins=9, steps=[1, 2, 5, 10], integer=True)
(8)                     )
(8)                     ax_ref[i].yaxis.set_major_locator(
(12)                        mticker.MaxNLocator(nbins=9, steps=[1, 2, 5, 10], integer=True)
(8)                     )
(0)             @image_comparison(["nonuniform_and_pcolor.png"], style="mpl20")
(0)             def test_nonuniform_and_pcolor():
(4)                 axs = plt.figure(figsize=(3, 3)).subplots(3, sharex=True, sharey=True)
(4)                 for ax, interpolation in zip(axs, ["nearest", "bilinear"]):
(8)                     im = NonUniformImage(ax, interpolation=interpolation)
(8)                     im.set_data(np.arange(3) ** 2, np.arange(3) ** 2,
(20)                                np.arange(9).reshape((3, 3)))
(8)                     ax.add_image(im)
(4)                 axs[2].pcolorfast(  # PcolorImage
(8)                     np.arange(4) ** 2, np.arange(4) ** 2, np.arange(9).reshape((3, 3)))
(4)                 for ax in axs:
(8)                     ax.set_axis_off()
(8)                     ax.set(xlim=(0, 10))
(0)             @image_comparison(
(4)                 ['rgba_antialias.png'], style='mpl20', remove_text=True,
(4)                 tol=0 if platform.machine() == 'x86_64' else 0.007)
(0)             def test_rgba_antialias():
(4)                 fig, axs = plt.subplots(2, 2, figsize=(3.5, 3.5), sharex=False,
(28)                                        sharey=False, constrained_layout=True)
```

```
(4)             N = 250
(4)             aa = np.ones((N, N))
(4)             aa[::2, :] = -1
(4)             x = np.arange(N) / N - 0.5
(4)             y = np.arange(N) / N - 0.5
(4)             X, Y = np.meshgrid(x, y)
(4)             R = np.sqrt(X**2 + Y**2)
(4)             f0 = 10
(4)             k = 75
(4)             a = np.sin(np.pi * 2 * (f0 * R + k * R**2 / 2))
(4)             a[:int(N/2), :][R[:int(N/2), :] < 0.4] = -1
(4)             a[:int(N/2), :][R[:int(N/2), :] < 0.3] = 1
(4)             aa[:, int(N/2):] = a[:, int(N/2):]
(4)             aa[20:50, 20:50] = np.nan
(4)             aa[70:90, 70:90] = 1e6
(4)             aa[70:90, 20:30] = -1e6
(4)             aa[70:90, 195:215] = 1e6
(4)             aa[20:30, 195:215] = -1e6
(4)             cmap = copy(plt.cm.RdBu_r)
(4)             cmap.set_over('yellow')
(4)             cmap.set_under('cyan')
(4)             axs = axs.flatten()
(4)             axs[0].imshow(aa, interpolation='nearest', cmap=cmap, vmin=-1.2, vmax=1.2)
(4)             axs[0].set_xlim([N/2-25, N/2+25])
(4)             axs[0].set_ylim([N/2+50, N/2-10])
(4)             axs[1].imshow(aa, interpolation='nearest', cmap=cmap, vmin=-1.2, vmax=1.2)
(4)             axs[2].imshow(aa, interpolation='antialiased', interpolation_stage='data',
(18)                          cmap=cmap, vmin=-1.2, vmax=1.2)
(4)             axs[3].imshow(aa, interpolation='antialiased', interpolation_stage='rgba',
(18)                          cmap=cmap, vmin=-1.2, vmax=1.2)
(0)         def test_rc_interpolation_stage():
(4)             for val in ["data", "rgba"]:
(8)                 with mpl.rc_context({"image.interpolation_stage": val}):
(12)                    assert plt.imshow([[1, 2]]).get_interpolation_stage() == val
(4)             for val in ["DATA", "foo", None]:
(8)                 with pytest.raises(ValueError):
(12)                    mpl.rcParams["image.interpolation_stage"] = val
(0)         @pytest.mark.filterwarnings(r'ignore:Data with more than .* '
(28)                                    'cannot be accurately displayed')
(0)         @pytest.mark.parametrize('origin', ['upper', 'lower'])
(0)         @pytest.mark.parametrize(
(4)             'dim, size, msg', [['row', 2**23, r'2\*\*23 columns'],
(23)                               ['col', 2**24, r'2\*\*24 rows']])
(0)         @check_figures_equal(extensions=('png', ))
(0)         def test_large_image(fig_test, fig_ref, dim, size, msg, origin):
(4)             ax_test = fig_test.subplots()
(4)             ax_ref = fig_ref.subplots()
(4)             array = np.zeros((1, size + 2))
(4)             array[:, array.size // 2:] = 1
(4)             if dim == 'col':
(8)                 array = array.T
(4)             im = ax_test.imshow(array, vmin=0, vmax=1,
(24)                                aspect='auto', extent=(0, 1, 0, 1),
(24)                                interpolation='none',
(24)                                origin=origin)
(4)             with pytest.warns(UserWarning,
(22)                              match=f'Data with more than {msg} cannot be '
(22)                              'accurately displayed.'):
(8)                 fig_test.canvas.draw()
(4)             array = np.zeros((1, 2))
(4)             array[:, 1] = 1
(4)             if dim == 'col':
(8)                 array = array.T
(4)             im = ax_ref.imshow(array, vmin=0, vmax=1, aspect='auto',
(23)                               extent=(0, 1, 0, 1),
(23)                               interpolation='none',
(23)                               origin=origin)
(0)         @check_figures_equal(extensions=["png"])
(0)         def test_str_norms(fig_test, fig_ref):
```

```
1084: (4)               t = np.random.rand(10, 10) * .8 + .1  # between 0 and 1
1085: (4)               axts = fig_test.subplots(1, 5)
1086: (4)               axts[0].imshow(t, norm="log")
1087: (4)               axts[1].imshow(t, norm="log", vmin=.2)
1088: (4)               axts[2].imshow(t, norm="symlog")
1089: (4)               axts[3].imshow(t, norm="symlog", vmin=.3, vmax=.7)
1090: (4)               axts[4].imshow(t, norm="logit", vmin=.3, vmax=.7)
1091: (4)               axrs = fig_ref.subplots(1, 5)
1092: (4)               axrs[0].imshow(t, norm=colors.LogNorm())
1093: (4)               axrs[1].imshow(t, norm=colors.LogNorm(vmin=.2))
1094: (4)               axrs[2].imshow(t, norm=colors.SymLogNorm(linthresh=2))
1095: (4)               axrs[3].imshow(t, norm=colors.SymLogNorm(linthresh=2, vmin=.3, vmax=.7))
1096: (4)               axrs[4].imshow(t, norm="logit", clim=(.3, .7))
1097: (4)               assert type(axts[0].images[0].norm) is colors.LogNorm  # Exactly that
class
1098: (4)               with pytest.raises(ValueError):
1099: (8)                   axts[0].imshow(t, norm="foobar")
1100: (0)           def test__resample_valid_output():
1101: (4)               resample = functools.partial(mpl._image.resample, transform=Affine2D())
1102: (4)               with pytest.raises(TypeError, match="incompatible function arguments"):
1103: (8)                   resample(np.zeros((9, 9)), None)
1104: (4)               with pytest.raises(ValueError, match="different dimensionalities"):
1105: (8)                   resample(np.zeros((9, 9)), np.zeros((9, 9, 4)))
1106: (4)               with pytest.raises(ValueError, match="different dimensionalities"):
1107: (8)                   resample(np.zeros((9, 9, 4)), np.zeros((9, 9)))
1108: (4)               with pytest.raises(ValueError, match="3D input array must be RGBA"):
1109: (8)                   resample(np.zeros((9, 9, 3)), np.zeros((9, 9, 4)))
1110: (4)               with pytest.raises(ValueError, match="3D output array must be RGBA"):
1111: (8)                   resample(np.zeros((9, 9, 4)), np.zeros((9, 9, 3)))
1112: (4)               with pytest.raises(ValueError, match="mismatched types"):
1113: (8)                   resample(np.zeros((9, 9), np.uint8), np.zeros((9, 9)))
1114: (4)               with pytest.raises(ValueError, match="must be C-contiguous"):
1115: (8)                   resample(np.zeros((9, 9)), np.zeros((9, 9)).T)
1116: (4)               out = np.zeros((9, 9))
1117: (4)               out.flags.writeable = False
1118: (4)               with pytest.raises(ValueError, match="Output array must be writeable"):
1119: (8)                   resample(np.zeros((9, 9)), out)
1120: (0)           def test_axesimage_get_shape():
1121: (4)               ax = plt.gca()
1122: (4)               im = AxesImage(ax)
1123: (4)               with pytest.raises(RuntimeError, match="You must first set the image
array"):
1124: (8)                   im.get_shape()
1125: (4)               z = np.arange(12, dtype=float).reshape((4, 3))
1126: (4)               im.set_data(z)
1127: (4)               assert im.get_shape() == (4, 3)
1128: (4)               assert im.get_size() == im.get_shape()
1129: (0)           def test_non_transdata_image_does_not_touch_aspect():
1130: (4)               ax = plt.figure().add_subplot()
1131: (4)               im = np.arange(4).reshape((2, 2))
1132: (4)               ax.imshow(im, transform=ax.transAxes)
1133: (4)               assert ax.get_aspect() == "auto"
1134: (4)               ax.imshow(im, transform=Affine2D().scale(2) + ax.transData)
1135: (4)               assert ax.get_aspect() == 1
1136: (4)               ax.imshow(im, transform=ax.transAxes, aspect=2)
1137: (4)               assert ax.get_aspect() == 2
1138: (0)           @pytest.mark.parametrize(
1139: (4)               'dtype',
1140: (4)               ('float64', 'float32', 'int16', 'uint16', 'int8', 'uint8'),
1141: (0)           )
1142: (0)           @pytest.mark.parametrize('ndim', (2, 3))
1143: (0)           def test_resample_dtypes(dtype, ndim):
1144: (4)               rng = np.random.default_rng(4181)
1145: (4)               shape = (2, 2) if ndim == 2 else (2, 2, 3)
1146: (4)               data = rng.uniform(size=shape).astype(np.dtype(dtype, copy=True))
1147: (4)               fig, ax = plt.subplots()
1148: (4)               axes_image = ax.imshow(data)
1149: (4)               axes_image.make_image(None)[0]
```

```
----------------------------------------

File 191 - test_legend.py:

1: (0)              import collections
2: (0)              import itertools
3: (0)              import platform
4: (0)              import time
5: (0)              from unittest import mock
6: (0)              import warnings
7: (0)              import numpy as np
8: (0)              from numpy.testing import assert_allclose
9: (0)              import pytest
10: (0)             from matplotlib.testing.decorators import check_figures_equal,
image_comparison
11: (0)             from matplotlib.testing._markers import needs_usetex
12: (0)             import matplotlib.pyplot as plt
13: (0)             import matplotlib as mpl
14: (0)             import matplotlib.patches as mpatches
15: (0)             import matplotlib.transforms as mtransforms
16: (0)             import matplotlib.collections as mcollections
17: (0)             import matplotlib.lines as mlines
18: (0)             from matplotlib.legend_handler import HandlerTuple
19: (0)             import matplotlib.legend as mlegend
20: (0)             from matplotlib import _api, rc_context
21: (0)             from matplotlib.font_manager import FontProperties
22: (0)             def test_legend_ordereddict():
23: (4)                 X = np.random.randn(10)
24: (4)                 Y = np.random.randn(10)
25: (4)                 labels = ['a'] * 5 + ['b'] * 5
26: (4)                 colors = ['r'] * 5 + ['g'] * 5
27: (4)                 fig, ax = plt.subplots()
28: (4)                 for x, y, label, color in zip(X, Y, labels, colors):
29: (8)                     ax.scatter(x, y, label=label, c=color)
30: (4)                 handles, labels = ax.get_legend_handles_labels()
31: (4)                 legend = collections.OrderedDict(zip(labels, handles))
32: (4)                 ax.legend(legend.values(), legend.keys(),
33: (14)                          loc='center left', bbox_to_anchor=(1, .5))
34: (0)             @image_comparison(['legend_auto1'], remove_text=True)
35: (0)             def test_legend_auto1():
36: (4)                 """Test automatic legend placement"""
37: (4)                 fig, ax = plt.subplots()
38: (4)                 x = np.arange(100)
39: (4)                 ax.plot(x, 50 - x, 'o', label='y=1')
40: (4)                 ax.plot(x, x - 50, 'o', label='y=-1')
41: (4)                 ax.legend(loc='best')
42: (0)             @image_comparison(['legend_auto2'], remove_text=True)
43: (0)             def test_legend_auto2():
44: (4)                 """Test automatic legend placement"""
45: (4)                 fig, ax = plt.subplots()
46: (4)                 x = np.arange(100)
47: (4)                 b1 = ax.bar(x, x, align='edge', color='m')
48: (4)                 b2 = ax.bar(x, x[::-1], align='edge', color='g')
49: (4)                 ax.legend([b1[0], b2[0]], ['up', 'down'], loc='best')
50: (0)             @image_comparison(['legend_auto3'])
51: (0)             def test_legend_auto3():
52: (4)                 """Test automatic legend placement"""
53: (4)                 fig, ax = plt.subplots()
54: (4)                 x = [0.9, 0.1, 0.1, 0.9, 0.9, 0.5]
55: (4)                 y = [0.95, 0.95, 0.05, 0.05, 0.5, 0.5]
56: (4)                 ax.plot(x, y, 'o-', label='line')
57: (4)                 ax.set_xlim(0.0, 1.0)
58: (4)                 ax.set_ylim(0.0, 1.0)
59: (4)                 ax.legend(loc='best')
60: (0)             def test_legend_auto4():
61: (4)                 """
62: (4)                 Check that the legend location with automatic placement is the same,
63: (4)                 whatever the histogram type is. Related to issue #9580.
64: (4)                 """
```

```
65: (4)                 fig, axs = plt.subplots(ncols=3, figsize=(6.4, 2.4))
66: (4)                 leg_bboxes = []
67: (4)                 for ax, ht in zip(axs.flat, ('bar', 'step', 'stepfilled')):
68: (8)                     ax.set_title(ht)
69: (8)                     ax.hist([0] + 5*[9], bins=range(10), label="Legend", histtype=ht)
70: (8)                     leg = ax.legend(loc="best")
71: (8)                     fig.canvas.draw()
72: (8)                     leg_bboxes.append(
73: (12)                        leg.get_window_extent().transformed(ax.transAxes.inverted()))
74: (4)                 assert_allclose(leg_bboxes[1].bounds, leg_bboxes[0].bounds)
75: (4)                 assert_allclose(leg_bboxes[2].bounds, leg_bboxes[0].bounds)
76: (0)             def test_legend_auto5():
77: (4)                 """
78: (4)                 Check that the automatic placement handle a rather complex
79: (4)                 case with non rectangular patch. Related to issue #9580.
80: (4)                 """
81: (4)                 fig, axs = plt.subplots(ncols=2, figsize=(9.6, 4.8))
82: (4)                 leg_bboxes = []
83: (4)                 for ax, loc in zip(axs.flat, ("center", "best")):
84: (8)                     for _patch in [
85: (16)                            mpatches.Ellipse(
86: (20)                                xy=(0.5, 0.9), width=0.8, height=0.2, fc="C1"),
87: (16)                            mpatches.Polygon(np.array([
88: (20)                                [0, 1], [0, 0], [1, 0], [1, 1], [0.9, 1.0], [0.9, 0.1],
89: (20)                                [0.1, 0.1], [0.1, 1.0], [0.1, 1.0]]), fc="C1"),
90: (16)                            mpatches.Wedge((0.5, 0.5), 0.5, 0, 360, width=0.05, fc="C0")
91: (16)                            ]:
92: (12)                        ax.add_patch(_patch)
93: (8)                     ax.plot([0.1, 0.9], [0.9, 0.9], label="A segment")  # sthg to label
94: (8)                     leg = ax.legend(loc=loc)
95: (8)                     fig.canvas.draw()
96: (8)                     leg_bboxes.append(
97: (12)                        leg.get_window_extent().transformed(ax.transAxes.inverted()))
98: (4)                 assert_allclose(leg_bboxes[1].bounds, leg_bboxes[0].bounds)
99: (0)             @image_comparison(['legend_various_labels'], remove_text=True)
100: (0)            def test_various_labels():
101: (4)                fig = plt.figure()
102: (4)                ax = fig.add_subplot(121)
103: (4)                ax.plot(np.arange(4), 'o', label=1)
104: (4)                ax.plot(np.linspace(4, 4.1), 'o', label='Développés')
105: (4)                ax.plot(np.arange(4, 1, -1), 'o', label='__nolegend__')
106: (4)                ax.legend(numpoints=1, loc='best')
107: (0)            def test_legend_label_with_leading_underscore():
108: (4)                """
109: (4)                Test that artists with labels starting with an underscore are not added to
110: (4)                the legend, and that a warning is issued if one tries to add them
111: (4)                explicitly.
112: (4)                """
113: (4)                fig, ax = plt.subplots()
114: (4)                line, = ax.plot([0, 1], label='_foo')
115: (4)                with pytest.warns(_api.MatplotlibDeprecationWarning, match="with an
underscore"):
116: (8)                    legend = ax.legend(handles=[line])
117: (4)                assert len(legend.legend_handles) == 0
118: (0)            @image_comparison(['legend_labels_first.png'], remove_text=True,
119: (18)                           tol=0.013 if platform.machine() == 'arm64' else 0)
120: (0)            def test_labels_first():
121: (4)                fig, ax = plt.subplots()
122: (4)                ax.plot(np.arange(10), '-o', label=1)
123: (4)                ax.plot(np.ones(10)*5, ':x', label="x")
124: (4)                ax.plot(np.arange(20, 10, -1), 'd', label="diamond")
125: (4)                ax.legend(loc='best', markerfirst=False)
126: (0)            @image_comparison(['legend_multiple_keys.png'], remove_text=True,
127: (18)                           tol=0.013 if platform.machine() == 'arm64' else 0)
128: (0)            def test_multiple_keys():
129: (4)                fig, ax = plt.subplots()
130: (4)                p1, = ax.plot([1, 2, 3], '-o')
131: (4)                p2, = ax.plot([2, 3, 4], '-x')
132: (4)                p3, = ax.plot([3, 4, 5], '-d')
```

```
(4)                 ax.legend([(p1, p2), (p2, p1), p3], ['two keys', 'pad=0', 'one key'],
(14)                          numpoints=1,
(14)                          handler_map={(p1, p2): HandlerTuple(ndivide=None),
(27)                                       (p2, p1): HandlerTuple(ndivide=None, pad=0)})
(0)             @image_comparison(['rgba_alpha.png'], remove_text=True,
(18)                              tol=0 if platform.machine() == 'x86_64' else 0.03)
(0)             def test_alpha_rgba():
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(range(10), lw=5)
(4)                 leg = plt.legend(['Longlabel that will go away'], loc='center')
(4)                 leg.legendPatch.set_facecolor([1, 0, 0, 0.5])
(0)             @image_comparison(['rcparam_alpha.png'], remove_text=True,
(18)                              tol=0 if platform.machine() == 'x86_64' else 0.03)
(0)             def test_alpha_rcparam():
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(range(10), lw=5)
(4)                 with mpl.rc_context(rc={'legend.framealpha': .75}):
(8)                     leg = plt.legend(['Longlabel that will go away'], loc='center')
(8)                     leg.legendPatch.set_facecolor([1, 0, 0, 0.5])
(0)             @image_comparison(['fancy'], remove_text=True, tol=0.05)
(0)             def test_fancy():
(4)                 plt.subplot(121)
(4)                 plt.plot([5] * 10, 'o--', label='XX')
(4)                 plt.scatter(np.arange(10), np.arange(10, 0, -1), label='XX\nXX')
(4)                 plt.errorbar(np.arange(10), np.arange(10), xerr=0.5,
(17)                            yerr=0.5, label='XX')
(4)                 plt.legend(loc="center left", bbox_to_anchor=[1.0, 0.5],
(15)                           ncols=2, shadow=True, title="My legend", numpoints=1)
(0)             @image_comparison(['framealpha'], remove_text=True,
(18)                              tol=0 if platform.machine() == 'x86_64' else 0.024)
(0)             def test_framealpha():
(4)                 x = np.linspace(1, 100, 100)
(4)                 y = x
(4)                 plt.plot(x, y, label='mylabel', lw=10)
(4)                 plt.legend(framealpha=0.5)
(0)             @image_comparison(['scatter_rc3', 'scatter_rc1'], remove_text=True)
(0)             def test_rc():
(4)                 plt.figure()
(4)                 ax = plt.subplot(121)
(4)                 ax.scatter(np.arange(10), np.arange(10, 0, -1), label='three')
(4)                 ax.legend(loc="center left", bbox_to_anchor=[1.0, 0.5],
(14)                          title="My legend")
(4)                 mpl.rcParams['legend.scatterpoints'] = 1
(4)                 plt.figure()
(4)                 ax = plt.subplot(121)
(4)                 ax.scatter(np.arange(10), np.arange(10, 0, -1), label='one')
(4)                 ax.legend(loc="center left", bbox_to_anchor=[1.0, 0.5],
(14)                          title="My legend")
(0)             @image_comparison(['legend_expand'], remove_text=True)
(0)             def test_legend_expand():
(4)                 """Test expand mode"""
(4)                 legend_modes = [None, "expand"]
(4)                 fig, axs = plt.subplots(len(legend_modes), 1)
(4)                 x = np.arange(100)
(4)                 for ax, mode in zip(axs, legend_modes):
(8)                     ax.plot(x, 50 - x, 'o', label='y=1')
(8)                     l1 = ax.legend(loc='upper left', mode=mode)
(8)                     ax.add_artist(l1)
(8)                     ax.plot(x, x - 50, 'o', label='y=-1')
(8)                     l2 = ax.legend(loc='right', mode=mode)
(8)                     ax.add_artist(l2)
(8)                     ax.legend(loc='lower left', mode=mode, ncols=2)
(0)             @image_comparison(['hatching'], remove_text=True, style='default')
(0)             def test_hatching():
(4)                 plt.rcParams['text.kerning_factor'] = 6
(4)                 fig, ax = plt.subplots()
(4)                 patch = plt.Rectangle((0, 0), 0.3, 0.3, hatch='xx',
(26)                                      label='Patch\ndefault color\nfilled')
(4)                 ax.add_patch(patch)
```

```
(4)                 patch = plt.Rectangle((0.33, 0), 0.3, 0.3, hatch='||', edgecolor='C1',
(26)                                      label='Patch\nexplicit color\nfilled')
(4)                 ax.add_patch(patch)
(4)                 patch = plt.Rectangle((0, 0.4), 0.3, 0.3, hatch='xx', fill=False,
(26)                                      label='Patch\ndefault color\nunfilled')
(4)                 ax.add_patch(patch)
(4)                 patch = plt.Rectangle((0.33, 0.4), 0.3, 0.3, hatch='||', fill=False,
(26)                                      edgecolor='C1',
(26)                                      label='Patch\nexplicit color\nunfilled')
(4)                 ax.add_patch(patch)
(4)                 ax.fill_between([0, .15, .3], [.8, .8, .8], [.9, 1.0, .9],
(20)                                hatch='+', label='Path\ndefault color')
(4)                 ax.fill_between([.33, .48, .63], [.8, .8, .8], [.9, 1.0, .9],
(20)                                hatch='+', edgecolor='C2', label='Path\nexplicit color')
(4)                 ax.set_xlim(-0.01, 1.1)
(4)                 ax.set_ylim(-0.01, 1.1)
(4)                 ax.legend(handlelength=4, handleheight=4)
(0)             def test_legend_remove():
(4)                 fig, ax = plt.subplots()
(4)                 lines = ax.plot(range(10))
(4)                 leg = fig.legend(lines, "test")
(4)                 leg.remove()
(4)                 assert fig.legends == []
(4)                 leg = ax.legend("test")
(4)                 leg.remove()
(4)                 assert ax.get_legend() is None
(0)             def test_reverse_legend_handles_and_labels():
(4)                 """Check that the legend handles and labels are reversed."""
(4)                 fig, ax = plt.subplots()
(4)                 x = 1
(4)                 y = 1
(4)                 labels = ["First label", "Second label", "Third label"]
(4)                 markers = ['.', ',', 'o']
(4)                 ax.plot(x, y, markers[0], label=labels[0])
(4)                 ax.plot(x, y, markers[1], label=labels[1])
(4)                 ax.plot(x, y, markers[2], label=labels[2])
(4)                 leg = ax.legend(reverse=True)
(4)                 actual_labels = [t.get_text() for t in leg.get_texts()]
(4)                 actual_markers = [h.get_marker() for h in leg.legend_handles]
(4)                 assert actual_labels == list(reversed(labels))
(4)                 assert actual_markers == list(reversed(markers))
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_reverse_legend_display(fig_test, fig_ref):
(4)                 """Check that the rendered legend entries are reversed"""
(4)                 ax = fig_test.subplots()
(4)                 ax.plot([1], 'ro', label="first")
(4)                 ax.plot([2], 'bx', label="second")
(4)                 ax.legend(reverse=True)
(4)                 ax = fig_ref.subplots()
(4)                 ax.plot([2], 'bx', label="second")
(4)                 ax.plot([1], 'ro', label="first")
(4)                 ax.legend()
(0)             class TestLegendFunction:
(4)                 def test_legend_no_args(self):
(8)                     lines = plt.plot(range(10), label='hello world')
(8)                     with mock.patch('matplotlib.legend.Legend') as Legend:
(12)                        plt.legend()
(8)                     Legend.assert_called_with(plt.gca(), lines, ['hello world'])
(4)                 def test_legend_positional_handles_labels(self):
(8)                     lines = plt.plot(range(10))
(8)                     with mock.patch('matplotlib.legend.Legend') as Legend:
(12)                        plt.legend(lines, ['hello world'])
(8)                     Legend.assert_called_with(plt.gca(), lines, ['hello world'])
(4)                 def test_legend_positional_handles_only(self):
(8)                     lines = plt.plot(range(10))
(8)                     with pytest.raises(TypeError, match='but found an Artist'):
(12)                        plt.legend(lines)
(4)                 def test_legend_positional_labels_only(self):
(8)                     lines = plt.plot(range(10), label='hello world')
```

```
(8)                     with mock.patch('matplotlib.legend.Legend') as Legend:
(12)                        plt.legend(['foobar'])
(8)                     Legend.assert_called_with(plt.gca(), lines, ['foobar'])
(4)                 def test_legend_three_args(self):
(8)                     lines = plt.plot(range(10), label='hello world')
(8)                     with mock.patch('matplotlib.legend.Legend') as Legend:
(12)                        plt.legend(lines, ['foobar'], loc='right')
(8)                     Legend.assert_called_with(plt.gca(), lines, ['foobar'], loc='right')
(4)                 def test_legend_handler_map(self):
(8)                     lines = plt.plot(range(10), label='hello world')
(8)                     with mock.patch('matplotlib.legend.'
(24)                                    '_get_legend_handles_labels') as handles_labels:
(12)                        handles_labels.return_value = lines, ['hello world']
(12)                        plt.legend(handler_map={'1': 2})
(8)                     handles_labels.assert_called_with([plt.gca()], {'1': 2})
(4)                 def test_legend_kwargs_handles_only(self):
(8)                     fig, ax = plt.subplots()
(8)                     x = np.linspace(0, 1, 11)
(8)                     ln1, = ax.plot(x, x, label='x')
(8)                     ln2, = ax.plot(x, 2*x, label='2x')
(8)                     ln3, = ax.plot(x, 3*x, label='3x')
(8)                     with mock.patch('matplotlib.legend.Legend') as Legend:
(12)                        ax.legend(handles=[ln3, ln2])  # reversed and not ln1
(8)                     Legend.assert_called_with(ax, [ln3, ln2], ['3x', '2x'])
(4)                 def test_legend_kwargs_labels_only(self):
(8)                     fig, ax = plt.subplots()
(8)                     x = np.linspace(0, 1, 11)
(8)                     ln1, = ax.plot(x, x)
(8)                     ln2, = ax.plot(x, 2*x)
(8)                     with mock.patch('matplotlib.legend.Legend') as Legend:
(12)                        ax.legend(labels=['x', '2x'])
(8)                     Legend.assert_called_with(ax, [ln1, ln2], ['x', '2x'])
(4)                 def test_legend_kwargs_handles_labels(self):
(8)                     fig, ax = plt.subplots()
(8)                     th = np.linspace(0, 2*np.pi, 1024)
(8)                     lns, = ax.plot(th, np.sin(th), label='sin')
(8)                     lnc, = ax.plot(th, np.cos(th), label='cos')
(8)                     with mock.patch('matplotlib.legend.Legend') as Legend:
(12)                        ax.legend(labels=('a', 'b'), handles=(lnc, lns))
(8)                     Legend.assert_called_with(ax, (lnc, lns), ('a', 'b'))
(4)                 def test_warn_mixed_args_and_kwargs(self):
(8)                     fig, ax = plt.subplots()
(8)                     th = np.linspace(0, 2*np.pi, 1024)
(8)                     lns, = ax.plot(th, np.sin(th), label='sin')
(8)                     lnc, = ax.plot(th, np.cos(th), label='cos')
(8)                     with pytest.warns(DeprecationWarning) as record:
(12)                        ax.legend((lnc, lns), labels=('a', 'b'))
(8)                     assert len(record) == 1
(8)                     assert str(record[0].message).startswith(
(12)                        "You have mixed positional and keyword arguments, some input may "
(12)                        "be discarded.")
(4)                 def test_parasite(self):
(8)                     from mpl_toolkits.axes_grid1 import host_subplot  # type: ignore
(8)                     host = host_subplot(111)
(8)                     par = host.twinx()
(8)                     p1, = host.plot([0, 1, 2], [0, 1, 2], label="Density")
(8)                     p2, = par.plot([0, 1, 2], [0, 3, 2], label="Temperature")
(8)                     with mock.patch('matplotlib.legend.Legend') as Legend:
(12)                        plt.legend()
(8)                     Legend.assert_called_with(host, [p1, p2], ['Density', 'Temperature'])
(0)             class TestLegendFigureFunction:
(4)                 def test_legend_handle_label(self):
(8)                     fig, ax = plt.subplots()
(8)                     lines = ax.plot(range(10))
(8)                     with mock.patch('matplotlib.legend.Legend') as Legend:
(12)                        fig.legend(lines, ['hello world'])
(8)                     Legend.assert_called_with(fig, lines, ['hello world'],
(34)                                              bbox_transform=fig.transFigure)
(4)                 def test_legend_no_args(self):
```

```
340: (8)                    fig, ax = plt.subplots()
341: (8)                    lines = ax.plot(range(10), label='hello world')
342: (8)                    with mock.patch('matplotlib.legend.Legend') as Legend:
343: (12)                       fig.legend()
344: (8)                    Legend.assert_called_with(fig, lines, ['hello world'],
345: (34)                                             bbox_transform=fig.transFigure)
346: (4)                def test_legend_label_arg(self):
347: (8)                    fig, ax = plt.subplots()
348: (8)                    lines = ax.plot(range(10))
349: (8)                    with mock.patch('matplotlib.legend.Legend') as Legend:
350: (12)                       fig.legend(['foobar'])
351: (8)                    Legend.assert_called_with(fig, lines, ['foobar'],
352: (34)                                             bbox_transform=fig.transFigure)
353: (4)                def test_legend_label_three_args(self):
354: (8)                    fig, ax = plt.subplots()
355: (8)                    lines = ax.plot(range(10))
356: (8)                    with pytest.raises(TypeError, match="0-2"):
357: (12)                       fig.legend(lines, ['foobar'], 'right')
358: (8)                    with pytest.raises(TypeError, match="0-2"):
359: (12)                       fig.legend(lines, ['foobar'], 'right', loc='left')
360: (4)                def test_legend_kw_args(self):
361: (8)                    fig, axs = plt.subplots(1, 2)
362: (8)                    lines = axs[0].plot(range(10))
363: (8)                    lines2 = axs[1].plot(np.arange(10) * 2.)
364: (8)                    with mock.patch('matplotlib.legend.Legend') as Legend:
365: (12)                       fig.legend(loc='right', labels=('a', 'b'), handles=(lines,
lines2))
366: (8)                    Legend.assert_called_with(
367: (12)                       fig, (lines, lines2), ('a', 'b'), loc='right',
368: (12)                       bbox_transform=fig.transFigure)
369: (4)                def test_warn_args_kwargs(self):
370: (8)                    fig, axs = plt.subplots(1, 2)
371: (8)                    lines = axs[0].plot(range(10))
372: (8)                    lines2 = axs[1].plot(np.arange(10) * 2.)
373: (8)                    with pytest.warns(DeprecationWarning) as record:
374: (12)                       fig.legend((lines, lines2), labels=('a', 'b'))
375: (8)                    assert len(record) == 1
376: (8)                    assert str(record[0].message).startswith(
377: (12)                       "You have mixed positional and keyword arguments, some input may "
378: (12)                       "be discarded.")
379: (0)            def test_figure_legend_outside():
380: (4)                todos = ['upper ' + pos for pos in ['left', 'center', 'right']]
381: (4)                todos += ['lower ' + pos for pos in ['left', 'center', 'right']]
382: (4)                todos += ['left ' + pos for pos in ['lower', 'center', 'upper']]
383: (4)                todos += ['right ' + pos for pos in ['lower', 'center', 'upper']]
384: (4)                upperext = [20.347556,  27.722556, 790.583, 545.499]
385: (4)                lowerext = [20.347556,  71.056556, 790.583, 588.833]
386: (4)                leftext = [151.681556, 27.722556, 790.583, 588.833]
387: (4)                rightext = [20.347556,  27.722556, 659.249, 588.833]
388: (4)                axbb = [upperext, upperext, upperext,
389: (12)                       lowerext, lowerext, lowerext,
390: (12)                       leftext, leftext, leftext,
391: (12)                       rightext, rightext, rightext]
392: (4)                legbb = [[10., 555., 133., 590.],     # upper left
393: (13)                        [338.5, 555., 461.5, 590.],  # upper center
394: (13)                        [667, 555., 790.,  590.],    # upper right
395: (13)                        [10., 10., 133.,  45.],      # lower left
396: (13)                        [338.5, 10., 461.5,  45.],   # lower center
397: (13)                        [667., 10., 790.,  45.],     # lower right
398: (13)                        [10., 10., 133., 45.],       # left lower
399: (13)                        [10., 282.5, 133., 317.5],   # left center
400: (13)                        [10., 555., 133., 590.],     # left upper
401: (13)                        [667, 10., 790., 45.],       # right lower
402: (13)                        [667., 282.5, 790., 317.5],  # right center
403: (13)                        [667., 555., 790., 590.]]    # right upper
404: (4)                for nn, todo in enumerate(todos):
405: (8)                    print(todo)
406: (8)                    fig, axs = plt.subplots(constrained_layout=True, dpi=100)
407: (8)                    axs.plot(range(10), label='Boo1')
```

```
408: (8)                    leg = fig.legend(loc='outside ' + todo)
409: (8)                    fig.draw_without_rendering()
410: (8)                    assert_allclose(axs.get_window_extent().extents,
411: (24)                                   axbb[nn])
412: (8)                    assert_allclose(leg.get_window_extent().extents,
413: (24)                                   legbb[nn])
414: (0)            @image_comparison(['legend_stackplot.png'],
415: (18)                                 tol=0.031 if platform.machine() == 'arm64' else 0)
416: (0)            def test_legend_stackplot():
417: (4)                """Test legend for PolyCollection using stackplot."""
418: (4)                fig, ax = plt.subplots()
419: (4)                x = np.linspace(0, 10, 10)
420: (4)                y1 = 1.0 * x
421: (4)                y2 = 2.0 * x + 1
422: (4)                y3 = 3.0 * x + 2
423: (4)                ax.stackplot(x, y1, y2, y3, labels=['y1', 'y2', 'y3'])
424: (4)                ax.set_xlim((0, 10))
425: (4)                ax.set_ylim((0, 70))
426: (4)                ax.legend(loc='best')
427: (0)            def test_cross_figure_patch_legend():
428: (4)                fig, ax = plt.subplots()
429: (4)                fig2, ax2 = plt.subplots()
430: (4)                brs = ax.bar(range(3), range(3))
431: (4)                fig2.legend(brs, 'foo')
432: (0)            def test_nanscatter():
433: (4)                fig, ax = plt.subplots()
434: (4)                h = ax.scatter([np.nan], [np.nan], marker="o",
435: (19)                               facecolor="r", edgecolor="r", s=3)
436: (4)                ax.legend([h], ["scatter"])
437: (4)                fig, ax = plt.subplots()
438: (4)                for color in ['red', 'green', 'blue']:
439: (8)                    n = 750
440: (8)                    x, y = np.random.rand(2, n)
441: (8)                    scale = 200.0 * np.random.rand(n)
442: (8)                    ax.scatter(x, y, c=color, s=scale, label=color,
443: (19)                               alpha=0.3, edgecolors='none')
444: (4)                ax.legend()
445: (4)                ax.grid(True)
446: (0)            def test_legend_repeatcheckok():
447: (4)                fig, ax = plt.subplots()
448: (4)                ax.scatter(0.0, 1.0, color='k', marker='o', label='test')
449: (4)                ax.scatter(0.5, 0.0, color='r', marker='v', label='test')
450: (4)                ax.legend()
451: (4)                hand, lab = mlegend._get_legend_handles_labels([ax])
452: (4)                assert len(lab) == 2
453: (4)                fig, ax = plt.subplots()
454: (4)                ax.scatter(0.0, 1.0, color='k', marker='o', label='test')
455: (4)                ax.scatter(0.5, 0.0, color='k', marker='v', label='test')
456: (4)                ax.legend()
457: (4)                hand, lab = mlegend._get_legend_handles_labels([ax])
458: (4)                assert len(lab) == 2
459: (0)            @image_comparison(['not_covering_scatter.png'])
460: (0)            def test_not_covering_scatter():
461: (4)                colors = ['b', 'g', 'r']
462: (4)                for n in range(3):
463: (8)                    plt.scatter([n], [n], color=colors[n])
464: (4)                plt.legend(['foo', 'foo', 'foo'], loc='best')
465: (4)                plt.gca().set_xlim(-0.5, 2.2)
466: (4)                plt.gca().set_ylim(-0.5, 2.2)
467: (0)            @image_comparison(['not_covering_scatter_transform.png'])
468: (0)            def test_not_covering_scatter_transform():
469: (4)                offset = mtransforms.Affine2D().translate(-20, 20)
470: (4)                x = np.linspace(0, 30, 1000)
471: (4)                plt.plot(x, x)
472: (4)                plt.scatter([20], [10], transform=offset + plt.gca().transData)
473: (4)                plt.legend(['foo', 'bar'], loc='best')
474: (0)            def test_linecollection_scaled_dashes():
475: (4)                lines1 = [[(0, .5), (.5, 1)], [(.3, .6), (.2, .2)]]
476: (4)                lines2 = [[[0.7, .2], [.8, .4]], [[.5, .7], [.6, .1]]]
```

```
477: (4)                lines3 = [[[0.6, .2], [.8, .4]], [[.5, .7], [.1, .1]]]
478: (4)                lc1 = mcollections.LineCollection(lines1, linestyles="--", lw=3)
479: (4)                lc2 = mcollections.LineCollection(lines2, linestyles="-.")
480: (4)                lc3 = mcollections.LineCollection(lines3, linestyles=":", lw=.5)
481: (4)                fig, ax = plt.subplots()
482: (4)                ax.add_collection(lc1)
483: (4)                ax.add_collection(lc2)
484: (4)                ax.add_collection(lc3)
485: (4)                leg = ax.legend([lc1, lc2, lc3], ["line1", "line2", 'line 3'])
486: (4)                h1, h2, h3 = leg.legend_handles
487: (4)                for oh, lh in zip((lc1, lc2, lc3), (h1, h2, h3)):
488: (8)                    assert oh.get_linestyles()[0] == lh._dash_pattern
489: (0)            def test_handler_numpoints():
490: (4)                """Test legend handler with numpoints <= 1."""
491: (4)                fig, ax = plt.subplots()
492: (4)                ax.plot(range(5), label='test')
493: (4)                ax.legend(numpoints=0.5)
494: (0)            def test_text_nohandler_warning():
495: (4)                """Test that Text artists with labels raise a warning"""
496: (4)                fig, ax = plt.subplots()
497: (4)                ax.plot([0], label="mock data")
498: (4)                ax.text(x=0, y=0, s="text", label="label")
499: (4)                with pytest.warns(UserWarning) as record:
500: (8)                    ax.legend()
501: (4)                assert len(record) == 1
502: (4)                f, ax = plt.subplots()
503: (4)                ax.pcolormesh(np.random.uniform(0, 1, (10, 10)))
504: (4)                with warnings.catch_warnings():
505: (8)                    warnings.simplefilter("error")
506: (8)                    ax.get_legend_handles_labels()
507: (0)            def test_empty_bar_chart_with_legend():
508: (4)                """Test legend when bar chart is empty with a label."""
509: (4)                plt.bar([], [], label='test')
510: (4)                plt.legend()
511: (0)            @image_comparison(['shadow_argument_types.png'], remove_text=True,
style='mpl20',
512: (18)                             tol=0.028 if platform.machine() == 'arm64' else 0)
513: (0)            def test_shadow_argument_types():
514: (4)                fig, ax = plt.subplots()
515: (4)                ax.plot([1, 2, 3], label='test')
516: (4)                legs = (ax.legend(loc='upper left', shadow=True),    # True
517: (12)                       ax.legend(loc='upper right', shadow=False),  # False
518: (12)                       ax.legend(loc='center left',                 # string
519: (22)                                 shadow={'color': 'red', 'alpha': 0.1}),
520: (12)                       ax.legend(loc='center right',                # tuple
521: (22)                                 shadow={'color': (0.1, 0.2, 0.5), 'oy': -5}),
522: (12)                       ax.legend(loc='lower left',                   # tab
523: (22)                                 shadow={'color': 'tab:cyan', 'ox': 10})
524: (12)                       )
525: (4)                for l in legs:
526: (8)                    ax.add_artist(l)
527: (4)                ax.legend(loc='lower right')  # default
528: (0)            def test_shadow_invalid_argument():
529: (4)                fig, ax = plt.subplots()
530: (4)                ax.plot([1, 2, 3], label='test')
531: (4)                with pytest.raises(ValueError, match="dict or bool"):
532: (8)                    ax.legend(loc="upper left", shadow="aardvark")  # Bad argument
533: (0)            def test_shadow_framealpha():
534: (4)                fig, ax = plt.subplots()
535: (4)                ax.plot(range(100), label="test")
536: (4)                leg = ax.legend(shadow=True, facecolor='w')
537: (4)                assert leg.get_frame().get_alpha() == 1
538: (0)            def test_legend_title_empty():
539: (4)                fig, ax = plt.subplots()
540: (4)                ax.plot(range(10), label="mock data")
541: (4)                leg = ax.legend()
542: (4)                assert leg.get_title().get_text() == ""
543: (4)                assert not leg.get_title().get_visible()
544: (0)            def test_legend_proper_window_extent():
```

```
(4)                 fig, ax = plt.subplots(dpi=100)
(4)                 ax.plot(range(10), label='Aardvark')
(4)                 leg = ax.legend()
(4)                 x01 = leg.get_window_extent(fig.canvas.get_renderer()).x0
(4)                 fig, ax = plt.subplots(dpi=200)
(4)                 ax.plot(range(10), label='Aardvark')
(4)                 leg = ax.legend()
(4)                 x02 = leg.get_window_extent(fig.canvas.get_renderer()).x0
(4)                 assert pytest.approx(x01*2, 0.1) == x02
(0)             def test_window_extent_cached_renderer():
(4)                 fig, ax = plt.subplots(dpi=100)
(4)                 ax.plot(range(10), label='Aardvark')
(4)                 leg = ax.legend()
(4)                 leg2 = fig.legend()
(4)                 fig.canvas.draw()
(4)                 leg.get_window_extent()
(4)                 leg2.get_window_extent()
(0)             def test_legend_title_fontprop_fontsize():
(4)                 plt.plot(range(10), label="mock data")
(4)                 with pytest.raises(ValueError):
(8)                     plt.legend(title='Aardvark', title_fontsize=22,
(19)                               title_fontproperties={'family': 'serif', 'size': 22})
(4)                 leg = plt.legend(title='Aardvark', title_fontproperties=FontProperties(
(39)                                                 family='serif', size=22))
(4)                 assert leg.get_title().get_size() == 22
(4)                 fig, axes = plt.subplots(2, 3, figsize=(10, 6))
(4)                 axes = axes.flat
(4)                 axes[0].plot(range(10), label="mock data")
(4)                 leg0 = axes[0].legend(title='Aardvark', title_fontsize=22)
(4)                 assert leg0.get_title().get_fontsize() == 22
(4)                 axes[1].plot(range(10), label="mock data")
(4)                 leg1 = axes[1].legend(title='Aardvark',
(26)                                      title_fontproperties={'family': 'serif', 'size':
22})
(4)                 assert leg1.get_title().get_fontsize() == 22
(4)                 axes[2].plot(range(10), label="mock data")
(4)                 mpl.rcParams['legend.title_fontsize'] = None
(4)                 leg2 = axes[2].legend(title='Aardvark',
(26)                                      title_fontproperties={'family': 'serif'})
(4)                 assert leg2.get_title().get_fontsize() == mpl.rcParams['font.size']
(4)                 axes[3].plot(range(10), label="mock data")
(4)                 leg3 = axes[3].legend(title='Aardvark')
(4)                 assert leg3.get_title().get_fontsize() == mpl.rcParams['font.size']
(4)                 axes[4].plot(range(10), label="mock data")
(4)                 mpl.rcParams['legend.title_fontsize'] = 20
(4)                 leg4 = axes[4].legend(title='Aardvark',
(26)                                      title_fontproperties={'family': 'serif'})
(4)                 assert leg4.get_title().get_fontsize() == 20
(4)                 axes[5].plot(range(10), label="mock data")
(4)                 leg5 = axes[5].legend(title='Aardvark')
(4)                 assert leg5.get_title().get_fontsize() == 20
(0)             @pytest.mark.parametrize('alignment', ('center', 'left', 'right'))
(0)             def test_legend_alignment(alignment):
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(range(10), label='test')
(4)                 leg = ax.legend(title="Aardvark", alignment=alignment)
(4)                 assert leg.get_children()[0].align == alignment
(4)                 assert leg.get_alignment() == alignment
(0)             @pytest.mark.parametrize('loc', ('center', 'best',))
(0)             def test_ax_legend_set_loc(loc):
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(range(10), label='test')
(4)                 leg = ax.legend()
(4)                 leg.set_loc(loc)
(4)                 assert leg._get_loc() == mlegend.Legend.codes[loc]
(0)             @pytest.mark.parametrize('loc', ('outside right', 'right',))
(0)             def test_fig_legend_set_loc(loc):
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(range(10), label='test')
```

```
(4)                 leg = fig.legend()
(4)                 leg.set_loc(loc)
(4)                 loc = loc.split()[1] if loc.startswith("outside") else loc
(4)                 assert leg._get_loc() == mlegend.Legend.codes[loc]
(0)             @pytest.mark.parametrize('alignment', ('center', 'left', 'right'))
(0)             def test_legend_set_alignment(alignment):
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(range(10), label='test')
(4)                 leg = ax.legend()
(4)                 leg.set_alignment(alignment)
(4)                 assert leg.get_children()[0].align == alignment
(4)                 assert leg.get_alignment() == alignment
(0)             @pytest.mark.parametrize('color', ('red', 'none', (.5, .5, .5)))
(0)             def test_legend_labelcolor_single(color):
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(np.arange(10), np.arange(10)*1, label='#1')
(4)                 ax.plot(np.arange(10), np.arange(10)*2, label='#2')
(4)                 ax.plot(np.arange(10), np.arange(10)*3, label='#3')
(4)                 leg = ax.legend(labelcolor=color)
(4)                 for text in leg.get_texts():
(8)                     assert mpl.colors.same_color(text.get_color(), color)
(0)             def test_legend_labelcolor_list():
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(np.arange(10), np.arange(10)*1, label='#1')
(4)                 ax.plot(np.arange(10), np.arange(10)*2, label='#2')
(4)                 ax.plot(np.arange(10), np.arange(10)*3, label='#3')
(4)                 leg = ax.legend(labelcolor=['r', 'g', 'b'])
(4)                 for text, color in zip(leg.get_texts(), ['r', 'g', 'b']):
(8)                     assert mpl.colors.same_color(text.get_color(), color)
(0)             def test_legend_labelcolor_linecolor():
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(np.arange(10), np.arange(10)*1, label='#1', color='r')
(4)                 ax.plot(np.arange(10), np.arange(10)*2, label='#2', color='g')
(4)                 ax.plot(np.arange(10), np.arange(10)*3, label='#3', color='b')
(4)                 leg = ax.legend(labelcolor='linecolor')
(4)                 for text, color in zip(leg.get_texts(), ['r', 'g', 'b']):
(8)                     assert mpl.colors.same_color(text.get_color(), color)
(0)             def test_legend_pathcollection_labelcolor_linecolor():
(4)                 fig, ax = plt.subplots()
(4)                 ax.scatter(np.arange(10), np.arange(10)*1, label='#1', c='r')
(4)                 ax.scatter(np.arange(10), np.arange(10)*2, label='#2', c='g')
(4)                 ax.scatter(np.arange(10), np.arange(10)*3, label='#3', c='b')
(4)                 leg = ax.legend(labelcolor='linecolor')
(4)                 for text, color in zip(leg.get_texts(), ['r', 'g', 'b']):
(8)                     assert mpl.colors.same_color(text.get_color(), color)
(0)             def test_legend_pathcollection_labelcolor_linecolor_iterable():
(4)                 fig, ax = plt.subplots()
(4)                 colors = np.random.default_rng().choice(['r', 'g', 'b'], 10)
(4)                 ax.scatter(np.arange(10), np.arange(10)*1, label='#1', c=colors)
(4)                 leg = ax.legend(labelcolor='linecolor')
(4)                 text, = leg.get_texts()
(4)                 assert mpl.colors.same_color(text.get_color(), 'black')
(0)             def test_legend_pathcollection_labelcolor_linecolor_cmap():
(4)                 fig, ax = plt.subplots()
(4)                 ax.scatter(np.arange(10), np.arange(10), c=np.arange(10), label='#1')
(4)                 leg = ax.legend(labelcolor='linecolor')
(4)                 text, = leg.get_texts()
(4)                 assert mpl.colors.same_color(text.get_color(), 'black')
(0)             def test_legend_labelcolor_markeredgecolor():
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(np.arange(10), np.arange(10)*1, label='#1', markeredgecolor='r')
(4)                 ax.plot(np.arange(10), np.arange(10)*2, label='#2', markeredgecolor='g')
(4)                 ax.plot(np.arange(10), np.arange(10)*3, label='#3', markeredgecolor='b')
(4)                 leg = ax.legend(labelcolor='markeredgecolor')
(4)                 for text, color in zip(leg.get_texts(), ['r', 'g', 'b']):
(8)                     assert mpl.colors.same_color(text.get_color(), color)
(0)             def test_legend_pathcollection_labelcolor_markeredgecolor():
(4)                 fig, ax = plt.subplots()
(4)                 ax.scatter(np.arange(10), np.arange(10)*1, label='#1', edgecolor='r')
```

```
(4)         ax.scatter(np.arange(10), np.arange(10)*2, label='#2', edgecolor='g')
(4)         ax.scatter(np.arange(10), np.arange(10)*3, label='#3', edgecolor='b')
(4)         leg = ax.legend(labelcolor='markeredgecolor')
(4)         for text, color in zip(leg.get_texts(), ['r', 'g', 'b']):
(8)             assert mpl.colors.same_color(text.get_color(), color)
(0)     def test_legend_pathcollection_labelcolor_markeredgecolor_iterable():
(4)         fig, ax = plt.subplots()
(4)         colors = np.random.default_rng().choice(['r', 'g', 'b'], 10)
(4)         ax.scatter(np.arange(10), np.arange(10)*1, label='#1', edgecolor=colors)
(4)         leg = ax.legend(labelcolor='markeredgecolor')
(4)         for text, color in zip(leg.get_texts(), ['k']):
(8)             assert mpl.colors.same_color(text.get_color(), color)
(0)     def test_legend_pathcollection_labelcolor_markeredgecolor_cmap():
(4)         fig, ax = plt.subplots()
(4)         edgecolors = mpl.cm.viridis(np.random.rand(10))
(4)         ax.scatter(
(8)             np.arange(10),
(8)             np.arange(10),
(8)             label='#1',
(8)             c=np.arange(10),
(8)             edgecolor=edgecolors,
(8)             cmap="Reds"
(4)         )
(4)         leg = ax.legend(labelcolor='markeredgecolor')
(4)         for text, color in zip(leg.get_texts(), ['k']):
(8)             assert mpl.colors.same_color(text.get_color(), color)
(0)     def test_legend_labelcolor_markerfacecolor():
(4)         fig, ax = plt.subplots()
(4)         ax.plot(np.arange(10), np.arange(10)*1, label='#1', markerfacecolor='r')
(4)         ax.plot(np.arange(10), np.arange(10)*2, label='#2', markerfacecolor='g')
(4)         ax.plot(np.arange(10), np.arange(10)*3, label='#3', markerfacecolor='b')
(4)         leg = ax.legend(labelcolor='markerfacecolor')
(4)         for text, color in zip(leg.get_texts(), ['r', 'g', 'b']):
(8)             assert mpl.colors.same_color(text.get_color(), color)
(0)     def test_legend_pathcollection_labelcolor_markerfacecolor():
(4)         fig, ax = plt.subplots()
(4)         ax.scatter(np.arange(10), np.arange(10)*1, label='#1', facecolor='r')
(4)         ax.scatter(np.arange(10), np.arange(10)*2, label='#2', facecolor='g')
(4)         ax.scatter(np.arange(10), np.arange(10)*3, label='#3', facecolor='b')
(4)         leg = ax.legend(labelcolor='markerfacecolor')
(4)         for text, color in zip(leg.get_texts(), ['r', 'g', 'b']):
(8)             assert mpl.colors.same_color(text.get_color(), color)
(0)     def test_legend_pathcollection_labelcolor_markerfacecolor_iterable():
(4)         fig, ax = plt.subplots()
(4)         colors = np.random.default_rng().choice(['r', 'g', 'b'], 10)
(4)         ax.scatter(np.arange(10), np.arange(10)*1, label='#1', facecolor=colors)
(4)         leg = ax.legend(labelcolor='markerfacecolor')
(4)         for text, color in zip(leg.get_texts(), ['k']):
(8)             assert mpl.colors.same_color(text.get_color(), color)
(0)     def test_legend_pathcollection_labelcolor_markfacecolor_cmap():
(4)         fig, ax = plt.subplots()
(4)         facecolors = mpl.cm.viridis(np.random.rand(10))
(4)         ax.scatter(
(8)             np.arange(10),
(8)             np.arange(10),
(8)             label='#1',
(8)             c=np.arange(10),
(8)             facecolor=facecolors
(4)         )
(4)         leg = ax.legend(labelcolor='markerfacecolor')
(4)         for text, color in zip(leg.get_texts(), ['k']):
(8)             assert mpl.colors.same_color(text.get_color(), color)
(0)     @pytest.mark.parametrize('color', ('red', 'none', (.5, .5, .5)))
(0)     def test_legend_labelcolor_rcparam_single(color):
(4)         fig, ax = plt.subplots()
(4)         ax.plot(np.arange(10), np.arange(10)*1, label='#1')
(4)         ax.plot(np.arange(10), np.arange(10)*2, label='#2')
(4)         ax.plot(np.arange(10), np.arange(10)*3, label='#3')
(4)         mpl.rcParams['legend.labelcolor'] = color
```

```
(4)                 leg = ax.legend()
(4)                 for text in leg.get_texts():
(8)                     assert mpl.colors.same_color(text.get_color(), color)
(0)             def test_legend_labelcolor_rcparam_linecolor():
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(np.arange(10), np.arange(10)*1, label='#1', color='r')
(4)                 ax.plot(np.arange(10), np.arange(10)*2, label='#2', color='g')
(4)                 ax.plot(np.arange(10), np.arange(10)*3, label='#3', color='b')
(4)                 mpl.rcParams['legend.labelcolor'] = 'linecolor'
(4)                 leg = ax.legend()
(4)                 for text, color in zip(leg.get_texts(), ['r', 'g', 'b']):
(8)                     assert mpl.colors.same_color(text.get_color(), color)
(0)             def test_legend_labelcolor_rcparam_markeredgecolor():
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(np.arange(10), np.arange(10)*1, label='#1', markeredgecolor='r')
(4)                 ax.plot(np.arange(10), np.arange(10)*2, label='#2', markeredgecolor='g')
(4)                 ax.plot(np.arange(10), np.arange(10)*3, label='#3', markeredgecolor='b')
(4)                 mpl.rcParams['legend.labelcolor'] = 'markeredgecolor'
(4)                 leg = ax.legend()
(4)                 for text, color in zip(leg.get_texts(), ['r', 'g', 'b']):
(8)                     assert mpl.colors.same_color(text.get_color(), color)
(0)             def test_legend_labelcolor_rcparam_markeredgecolor_short():
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(np.arange(10), np.arange(10)*1, label='#1', markeredgecolor='r')
(4)                 ax.plot(np.arange(10), np.arange(10)*2, label='#2', markeredgecolor='g')
(4)                 ax.plot(np.arange(10), np.arange(10)*3, label='#3', markeredgecolor='b')
(4)                 mpl.rcParams['legend.labelcolor'] = 'mec'
(4)                 leg = ax.legend()
(4)                 for text, color in zip(leg.get_texts(), ['r', 'g', 'b']):
(8)                     assert mpl.colors.same_color(text.get_color(), color)
(0)             def test_legend_labelcolor_rcparam_markerfacecolor():
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(np.arange(10), np.arange(10)*1, label='#1', markerfacecolor='r')
(4)                 ax.plot(np.arange(10), np.arange(10)*2, label='#2', markerfacecolor='g')
(4)                 ax.plot(np.arange(10), np.arange(10)*3, label='#3', markerfacecolor='b')
(4)                 mpl.rcParams['legend.labelcolor'] = 'markerfacecolor'
(4)                 leg = ax.legend()
(4)                 for text, color in zip(leg.get_texts(), ['r', 'g', 'b']):
(8)                     assert mpl.colors.same_color(text.get_color(), color)
(0)             def test_legend_labelcolor_rcparam_markerfacecolor_short():
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(np.arange(10), np.arange(10)*1, label='#1', markerfacecolor='r')
(4)                 ax.plot(np.arange(10), np.arange(10)*2, label='#2', markerfacecolor='g')
(4)                 ax.plot(np.arange(10), np.arange(10)*3, label='#3', markerfacecolor='b')
(4)                 mpl.rcParams['legend.labelcolor'] = 'mfc'
(4)                 leg = ax.legend()
(4)                 for text, color in zip(leg.get_texts(), ['r', 'g', 'b']):
(8)                     assert mpl.colors.same_color(text.get_color(), color)
(0)             @pytest.mark.filterwarnings("ignore:No artists with labels found to put in
legend")
(0)             def test_get_set_draggable():
(4)                 legend = plt.legend()
(4)                 assert not legend.get_draggable()
(4)                 legend.set_draggable(True)
(4)                 assert legend.get_draggable()
(4)                 legend.set_draggable(False)
(4)                 assert not legend.get_draggable()
(0)             @pytest.mark.parametrize('draggable', (True, False))
(0)             def test_legend_draggable(draggable):
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(range(10), label='shabnams')
(4)                 leg = ax.legend(draggable=draggable)
(4)                 assert leg.get_draggable() is draggable
(0)             def test_alpha_handles():
(4)                 x, n, hh = plt.hist([1, 2, 3], alpha=0.25, label='data', color='red')
(4)                 legend = plt.legend()
(4)                 for lh in legend.legend_handles:
(8)                     lh.set_alpha(1.0)
(4)                 assert lh.get_facecolor()[:-1] == hh[1].get_facecolor()[:-1]
```

```
819: (4)                assert lh.get_edgecolor()[:-1] == hh[1].get_edgecolor()[:-1]
820: (0)            @needs_usetex
821: (0)            def test_usetex_no_warn(caplog):
822: (4)                mpl.rcParams['font.family'] = 'serif'
823: (4)                mpl.rcParams['font.serif'] = 'Computer Modern'
824: (4)                mpl.rcParams['text.usetex'] = True
825: (4)                fig, ax = plt.subplots()
826: (4)                ax.plot(0, 0, label='input')
827: (4)                ax.legend(title="My legend")
828: (4)                fig.canvas.draw()
829: (4)                assert "Font family ['serif'] not found." not in caplog.text
830: (0)            def test_warn_big_data_best_loc(monkeypatch):
831: (4)                counter = itertools.count(0, step=1.5)
832: (4)                monkeypatch.setattr(time, 'perf_counter', lambda: next(counter))
833: (4)                fig, ax = plt.subplots()
834: (4)                fig.canvas.draw()  # So that we can call draw_artist later.
835: (4)                x = [0.9, 0.1, 0.1, 0.9, 0.9, 0.5]
836: (4)                y = [0.95, 0.95, 0.05, 0.05, 0.5, 0.5]
837: (4)                ax.plot(x, y, 'o-', label='line')
838: (4)                with rc_context({'legend.loc': 'best'}):
839: (8)                    legend = ax.legend()
840: (4)                with pytest.warns(UserWarning,
841: (22)                                  match='Creating legend with loc="best" can be slow with
large '
842: (22)                                  'amounts of data.') as records:
843: (8)                    fig.draw_artist(legend)  # Don't bother drawing the lines -- it's
slow.
844: (4)                assert len(records) == 2
845: (0)            def test_no_warn_big_data_when_loc_specified(monkeypatch):
846: (4)                counter = itertools.count(0, step=1.5)
847: (4)                monkeypatch.setattr(time, 'perf_counter', lambda: next(counter))
848: (4)                fig, ax = plt.subplots()
849: (4)                fig.canvas.draw()
850: (4)                x = [0.9, 0.1, 0.1, 0.9, 0.9, 0.5]
851: (4)                y = [0.95, 0.95, 0.05, 0.05, 0.5, 0.5]
852: (4)                ax.plot(x, y, 'o-', label='line')
853: (4)                legend = ax.legend('best')
854: (4)                fig.draw_artist(legend)  # Check that no warning is emitted.
855: (0)            @pytest.mark.parametrize('label_array', [['low', 'high'],
856: (41)                                                     ('low', 'high'),
857: (41)                                                     np.array(['low', 'high'])])
858: (0)            def test_plot_multiple_input_multiple_label(label_array):
859: (4)                x = [1, 2, 3]
860: (4)                y = [[1, 2],
861: (9)                     [2, 5],
862: (9)                     [4, 9]]
863: (4)                fig, ax = plt.subplots()
864: (4)                ax.plot(x, y, label=label_array)
865: (4)                leg = ax.legend()
866: (4)                legend_texts = [entry.get_text() for entry in leg.get_texts()]
867: (4)                assert legend_texts == ['low', 'high']
868: (0)            @pytest.mark.parametrize('label', ['one', 1, int])
869: (0)            def test_plot_multiple_input_single_label(label):
870: (4)                x = [1, 2, 3]
871: (4)                y = [[1, 2],
872: (9)                     [2, 5],
873: (9)                     [4, 9]]
874: (4)                fig, ax = plt.subplots()
875: (4)                ax.plot(x, y, label=label)
876: (4)                leg = ax.legend()
877: (4)                legend_texts = [entry.get_text() for entry in leg.get_texts()]
878: (4)                assert legend_texts == [str(label)] * 2
879: (0)            @pytest.mark.parametrize('label_array', [['low', 'high'],
880: (41)                                                     ('low', 'high'),
881: (41)                                                     np.array(['low', 'high'])])
882: (0)            def test_plot_single_input_multiple_label(label_array):
883: (4)                x = [1, 2, 3]
884: (4)                y = [2, 5, 6]
885: (4)                fig, ax = plt.subplots()
```

```
886: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning,
887: (22)                                  match='Passing label as a length 2 sequence'):
888: (8)                    ax.plot(x, y, label=label_array)
889: (4)                leg = ax.legend()
890: (4)                assert len(leg.get_texts()) == 1
891: (4)                assert leg.get_texts()[0].get_text() == str(label_array)
892: (0)            def test_plot_single_input_list_label():
893: (4)                fig, ax = plt.subplots()
894: (4)                line, = ax.plot([[0], [1]], label=['A'])
895: (4)                assert line.get_label() == 'A'
896: (0)            def test_plot_multiple_label_incorrect_length_exception():
897: (4)                with pytest.raises(ValueError):
898: (8)                    x = [1, 2, 3]
899: (8)                    y = [[1, 2],
900: (13)                        [2, 5],
901: (13)                        [4, 9]]
902: (8)                    label = ['high', 'low', 'medium']
903: (8)                    fig, ax = plt.subplots()
904: (8)                    ax.plot(x, y, label=label)
905: (0)            def test_legend_face_edgecolor():
906: (4)                fig, ax = plt.subplots()
907: (4)                ax.fill_between([0, 1, 2], [1, 2, 3], [2, 3, 4],
908: (20)                                facecolor='r', edgecolor='face', label='Fill')
909: (4)                ax.legend()
910: (0)            def test_legend_text_axes():
911: (4)                fig, ax = plt.subplots()
912: (4)                ax.plot([1, 2], [3, 4], label='line')
913: (4)                leg = ax.legend()
914: (4)                assert leg.axes is ax
915: (4)                assert leg.get_texts()[0].axes is ax
916: (0)            def test_handlerline2d():
917: (4)                fig, ax = plt.subplots()
918: (4)                ax.scatter([0, 1], [0, 1], marker="v")
919: (4)                handles = [mlines.Line2D([0], [0], marker="v")]
920: (4)                leg = ax.legend(handles, ["Aardvark"], numpoints=1)
921: (4)                assert handles[0].get_marker() == leg.legend_handles[0].get_marker()
922: (0)            def test_subfigure_legend():
923: (4)                subfig = plt.figure().subfigures()
924: (4)                ax = subfig.subplots()
925: (4)                ax.plot([0, 1], [0, 1], label="line")
926: (4)                leg = subfig.legend()
927: (4)                assert leg.figure is subfig
928: (0)            def test_setting_alpha_keeps_polycollection_color():
929: (4)                pc = plt.fill_between([0, 1], [2, 3], color='#123456', label='label')
930: (4)                patch = plt.legend().get_patches()[0]
931: (4)                patch.set_alpha(0.5)
932: (4)                assert patch.get_facecolor()[:3] == tuple(pc.get_facecolor()[0][:3])
933: (4)                assert patch.get_edgecolor()[:3] == tuple(pc.get_edgecolor()[0][:3])
934: (0)            def test_legend_markers_from_line2d():
935: (4)                _markers = ['.', '*', 'v']
936: (4)                fig, ax = plt.subplots()
937: (4)                lines = [mlines.Line2D([0], [0], ls='None', marker=mark)
938: (13)                         for mark in _markers]
939: (4)                labels = ["foo", "bar", "xyzzy"]
940: (4)                markers = [line.get_marker() for line in lines]
941: (4)                legend = ax.legend(lines, labels)
942: (4)                new_markers = [line.get_marker() for line in legend.get_lines()]
943: (4)                new_labels = [text.get_text() for text in legend.get_texts()]
944: (4)                assert markers == new_markers == _markers
945: (4)                assert labels == new_labels
946: (0)            @check_figures_equal()
947: (0)            def test_ncol_ncols(fig_test, fig_ref):
948: (4)                strings = ["a", "b", "c", "d", "e", "f"]
949: (4)                ncols = 3
950: (4)                fig_test.legend(strings, ncol=ncols)
951: (4)                fig_ref.legend(strings, ncols=ncols)
952: (0)            def test_loc_invalid_tuple_exception():
953: (4)                fig, ax = plt.subplots()
954: (4)                with pytest.raises(ValueError, match=('loc must be string, coordinate '
```

```
955: (23)                                 'tuple, or an integer 0-10, not \\(1.1,\\)')):
956: (8)                     ax.legend(loc=(1.1, ), labels=["mock data"])
957: (4)                 with pytest.raises(ValueError, match=('loc must be string, coordinate '
958: (23)                                 'tuple, or an integer 0-10, not \\(0.481, 0.4227,
0.4523\\)')):
959: (8)                     ax.legend(loc=(0.481, 0.4227, 0.4523), labels=["mock data"])
960: (4)                 with pytest.raises(ValueError, match=('loc must be string, coordinate '
961: (23)                                 'tuple, or an integer 0-10, not \\(0.481, \'go
blue\'\\)')):
962: (8)                     ax.legend(loc=(0.481, "go blue"), labels=["mock data"])
963: (0)             def test_loc_valid_tuple():
964: (4)                 fig, ax = plt.subplots()
965: (4)                 ax.legend(loc=(0.481, 0.442), labels=["mock data"])
966: (4)                 ax.legend(loc=(1, 2), labels=["mock data"])
967: (0)             def test_loc_valid_list():
968: (4)                 fig, ax = plt.subplots()
969: (4)                 ax.legend(loc=[0.481, 0.442], labels=["mock data"])
970: (4)                 ax.legend(loc=[1, 2], labels=["mock data"])
971: (0)             def test_loc_invalid_list_exception():
972: (4)                 fig, ax = plt.subplots()
973: (4)                 with pytest.raises(ValueError, match=('loc must be string, coordinate '
974: (23)                                 'tuple, or an integer 0-10, not \\[1.1, 2.2, 3.3\\]')):
975: (8)                     ax.legend(loc=[1.1, 2.2, 3.3], labels=["mock data"])
976: (0)             def test_loc_invalid_type():
977: (4)                 fig, ax = plt.subplots()
978: (4)                 with pytest.raises(ValueError, match=("loc must be string, coordinate "
979: (23)                                 "tuple, or an integer 0-10, not {'not': True}")):
980: (8)                     ax.legend(loc={'not': True}, labels=["mock data"])
981: (0)             def test_loc_validation_numeric_value():
982: (4)                 fig, ax = plt.subplots()
983: (4)                 ax.legend(loc=0, labels=["mock data"])
984: (4)                 ax.legend(loc=1, labels=["mock data"])
985: (4)                 ax.legend(loc=5, labels=["mock data"])
986: (4)                 ax.legend(loc=10, labels=["mock data"])
987: (4)                 with pytest.raises(ValueError, match=('loc must be string, coordinate '
988: (23)                                 'tuple, or an integer 0-10, not 11')):
989: (8)                     ax.legend(loc=11, labels=["mock data"])
990: (4)                 with pytest.raises(ValueError, match=('loc must be string, coordinate '
991: (23)                                 'tuple, or an integer 0-10, not -1')):
992: (8)                     ax.legend(loc=-1, labels=["mock data"])
993: (0)             def test_loc_validation_string_value():
994: (4)                 fig, ax = plt.subplots()
995: (4)                 labels = ["mock data"]
996: (4)                 ax.legend(loc='best', labels=labels)
997: (4)                 ax.legend(loc='upper right', labels=labels)
998: (4)                 ax.legend(loc='best', labels=labels)
999: (4)                 ax.legend(loc='upper right', labels=labels)
1000: (4)                ax.legend(loc='upper left', labels=labels)
1001: (4)                ax.legend(loc='lower left', labels=labels)
1002: (4)                ax.legend(loc='lower right', labels=labels)
1003: (4)                ax.legend(loc='right', labels=labels)
1004: (4)                ax.legend(loc='center left', labels=labels)
1005: (4)                ax.legend(loc='center right', labels=labels)
1006: (4)                ax.legend(loc='lower center', labels=labels)
1007: (4)                ax.legend(loc='upper center', labels=labels)
1008: (4)                with pytest.raises(ValueError, match="'wrong' is not a valid value for"):
1009: (8)                    ax.legend(loc='wrong', labels=labels)
1010: (0)            def test_legend_handle_label_mismatch():
1011: (4)                pl1, = plt.plot(range(10))
1012: (4)                pl2, = plt.plot(range(10))
1013: (4)                with pytest.warns(UserWarning, match="number of handles and labels"):
1014: (8)                    legend = plt.legend(handles=[pl1, pl2], labels=["pl1", "pl2", "pl3"])
1015: (8)                    assert len(legend.legend_handles) == 2
1016: (8)                    assert len(legend.get_texts()) == 2
1017: (0)            def test_legend_handle_label_mismatch_no_len():
1018: (4)                pl1, = plt.plot(range(10))
1019: (4)                pl2, = plt.plot(range(10))
1020: (4)                legend = plt.legend(handles=iter([pl1, pl2]),
1021: (24)                                   labels=iter(["pl1", "pl2", "pl3"]))
```

```
1022: (4)               assert len(legend.legend_handles) == 2
1023: (4)               assert len(legend.get_texts()) == 2
1024: (0)           def test_legend_nolabels_warning():
1025: (4)               plt.plot([1, 2, 3])
1026: (4)               with pytest.raises(UserWarning, match="No artists with labels found"):
1027: (8)                   plt.legend()
1028: (0)           @pytest.mark.filterwarnings("ignore:No artists with labels found to put in
legend")
1029: (0)           def test_legend_nolabels_draw():
1030: (4)               plt.plot([1, 2, 3])
1031: (4)               plt.legend()
1032: (4)               assert plt.gca().get_legend() is not None
1033: (0)           def test_legend_loc_polycollection():
1034: (4)               x = [3, 4, 5]
1035: (4)               y1 = [1, 1, 1]
1036: (4)               y2 = [5, 5, 5]
1037: (4)               leg_bboxes = []
1038: (4)               fig, axs = plt.subplots(ncols=2, figsize=(10, 5))
1039: (4)               for ax, loc in zip(axs.flat, ('best', 'lower left')):
1040: (8)                   ax.fill_between(x, y1, y2, color='gray', alpha=0.5, label='Shaded
Area')
1041: (8)                   ax.set_xlim(0, 6)
1042: (8)                   ax.set_ylim(-1, 5)
1043: (8)                   leg = ax.legend(loc=loc)
1044: (8)                   fig.canvas.draw()
1045: (8)                   leg_bboxes.append(
1046: (12)                      leg.get_window_extent().transformed(ax.transAxes.inverted()))
1047: (4)               assert_allclose(leg_bboxes[1].bounds, leg_bboxes[0].bounds)
1048: (0)           def test_legend_text():
1049: (4)               fig, axs = plt.subplots(ncols=2, figsize=(10, 5))
1050: (4)               leg_bboxes = []
1051: (4)               for ax, loc in zip(axs.flat, ('best', 'lower left')):
1052: (8)                   x = [1, 2]
1053: (8)                   y = [2, 1]
1054: (8)                   ax.plot(x, y, label='plot name')
1055: (8)                   ax.text(1.5, 2, 'some text blahblah', verticalalignment='top')
1056: (8)                   leg = ax.legend(loc=loc)
1057: (8)                   fig.canvas.draw()
1058: (8)                   leg_bboxes.append(
1059: (12)                      leg.get_window_extent().transformed(ax.transAxes.inverted()))
1060: (4)               assert_allclose(leg_bboxes[1].bounds, leg_bboxes[0].bounds)
1061: (0)           def test_boxplot_legend_labels():
1062: (4)               np.random.seed(19680801)
1063: (4)               data = np.random.random((10, 4))
1064: (4)               fig, axs = plt.subplots(nrows=1, ncols=4)
1065: (4)               legend_labels = ['box A', 'box B', 'box C', 'box D']
1066: (4)               bp1 = axs[0].boxplot(data, patch_artist=True, label=legend_labels)
1067: (4)               assert [v.get_label() for v in bp1['boxes']] == legend_labels
1068: (4)               handles, labels = axs[0].get_legend_handles_labels()
1069: (4)               assert labels == legend_labels
1070: (4)               assert all(isinstance(h, mpl.patches.PathPatch) for h in handles)
1071: (4)               bp2 = axs[1].boxplot(data, label=legend_labels, showbox=False)
1072: (4)               assert [v.get_label() for v in bp2['medians']] == legend_labels
1073: (4)               handles, labels = axs[1].get_legend_handles_labels()
1074: (4)               assert labels == legend_labels
1075: (4)               assert all(isinstance(h, mpl.lines.Line2D) for h in handles)
1076: (4)               with pytest.raises(ValueError, match='values must have same the length'):
1077: (8)                   bp3 = axs[2].boxplot(data, label=legend_labels[:-1])
1078: (4)               bp4 = axs[3].boxplot(data, label='box A')
1079: (4)               assert bp4['medians'][0].get_label() == 'box A'
1080: (4)               assert all(x.get_label().startswith("_") for x in bp4['medians'][1:])
```

File 192 - test_lines.py:

```
1: (0)              """
2: (0)              Tests specific to the lines module.
3: (0)              """
```

```
import itertools
import platform
import timeit
from types import SimpleNamespace
from cycler import cycler
import numpy as np
from numpy.testing import assert_array_equal
import pytest
import matplotlib
import matplotlib as mpl
from matplotlib import _path
import matplotlib.lines as mlines
from matplotlib.markers import MarkerStyle
from matplotlib.path import Path
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms
from matplotlib.testing.decorators import image_comparison, check_figures_equal
def test_segment_hits():
    """Test a problematic case."""
    cx, cy = 553, 902
    x, y = np.array([553., 553.]), np.array([95., 947.])
    radius = 6.94
    assert_array_equal(mlines.segment_hits(cx, cy, x, y, radius), [0])
@pytest.mark.flaky(reruns=3)
def test_invisible_Line_rendering():
    """
    GitHub issue #1256 identified a bug in Line.draw method
    Despite visibility attribute set to False, the draw method was not
    returning early enough and some pre-rendering code was executed
    though not necessary.
    Consequence was an excessive draw time for invisible Line instances
    holding a large number of points (Npts> 10**6)
    """
    N = 10**7
    x = np.linspace(0, 1, N)
    y = np.random.normal(size=N)
    fig = plt.figure()
    ax = plt.subplot()
    l = mlines.Line2D(x, y)
    l.set_visible(False)
    t_no_line = min(timeit.repeat(fig.canvas.draw, number=1, repeat=3))
    ax.add_line(l)
    t_invisible_line = min(timeit.repeat(fig.canvas.draw, number=1, repeat=3))
    slowdown_factor = t_invisible_line / t_no_line
    slowdown_threshold = 2  # trying to avoid false positive failures
    assert slowdown_factor < slowdown_threshold
def test_set_line_coll_dash():
    fig, ax = plt.subplots()
    np.random.seed(0)
    ax.contour(np.random.randn(20, 30), linestyles=[(0, (3, 3))])
def test_invalid_line_data():
    with pytest.raises(RuntimeError, match='xdata must be'):
        mlines.Line2D(0, [])
    with pytest.raises(RuntimeError, match='ydata must be'):
        mlines.Line2D([], 1)
    line = mlines.Line2D([], [])
    with pytest.raises(RuntimeError, match='x must be'):
        line.set_xdata(0)
    with pytest.raises(RuntimeError, match='y must be'):
        line.set_ydata(0)
@image_comparison(['line_dashes'], remove_text=True, tol=0.003)
def test_line_dashes():
    fig, ax = plt.subplots()
    ax.plot(range(10), linestyle=(0, (3, 3)), lw=5)
def test_line_colors():
    fig, ax = plt.subplots()
    ax.plot(range(10), color='none')
    ax.plot(range(10), color='r')
```

```
72: (4)                 ax.plot(range(10), color='.3')
73: (4)                 ax.plot(range(10), color=(1, 0, 0, 1))
74: (4)                 ax.plot(range(10), color=(1, 0, 0))
75: (4)                 fig.canvas.draw()
76: (0)             def test_valid_colors():
77: (4)                 line = mlines.Line2D([], [])
78: (4)                 with pytest.raises(ValueError):
79: (8)                     line.set_color("foobar")
80: (0)             def test_linestyle_variants():
81: (4)                 fig, ax = plt.subplots()
82: (4)                 for ls in ["-", "solid", "--", "dashed",
83: (15)                           "-.", "dashdot", ":", "dotted",
84: (15)                           (0, None), (0, ()), (0, []),  # gh-22930
85: (15)                           ]:
86: (8)                     ax.plot(range(10), linestyle=ls)
87: (4)                 fig.canvas.draw()
88: (0)             def test_valid_linestyles():
89: (4)                 line = mlines.Line2D([], [])
90: (4)                 with pytest.raises(ValueError):
91: (8)                     line.set_linestyle('aardvark')
92: (0)             @image_comparison(['drawstyle_variants.png'], remove_text=True,
93: (18)                              tol=0.03 if platform.machine() == 'arm64' else 0)
94: (0)             def test_drawstyle_variants():
95: (4)                 fig, axs = plt.subplots(6)
96: (4)                 dss = ["default", "steps-mid", "steps-pre", "steps-post", "steps", None]
97: (4)                 for ax, ds in zip(axs.flat, dss):
98: (8)                     ax.plot(range(2000), drawstyle=ds)
99: (8)                     ax.set(xlim=(0, 2), ylim=(0, 2))
100: (0)            @check_figures_equal(extensions=('png',))
101: (0)            def test_no_subslice_with_transform(fig_ref, fig_test):
102: (4)                ax = fig_ref.add_subplot()
103: (4)                x = np.arange(2000)
104: (4)                ax.plot(x + 2000, x)
105: (4)                ax = fig_test.add_subplot()
106: (4)                t = mtransforms.Affine2D().translate(2000.0, 0.0)
107: (4)                ax.plot(x, x, transform=t+ax.transData)
108: (0)            def test_valid_drawstyles():
109: (4)                line = mlines.Line2D([], [])
110: (4)                with pytest.raises(ValueError):
111: (8)                    line.set_drawstyle('foobar')
112: (0)            def test_set_drawstyle():
113: (4)                x = np.linspace(0, 2*np.pi, 10)
114: (4)                y = np.sin(x)
115: (4)                fig, ax = plt.subplots()
116: (4)                line, = ax.plot(x, y)
117: (4)                line.set_drawstyle("steps-pre")
118: (4)                assert len(line.get_path().vertices) == 2*len(x)-1
119: (4)                line.set_drawstyle("default")
120: (4)                assert len(line.get_path().vertices) == len(x)
121: (0)            @image_comparison(
122: (4)                ['line_collection_dashes'], remove_text=True, style='mpl20',
123: (4)                tol=0 if platform.machine() == 'x86_64' else 0.65)
124: (0)            def test_set_line_coll_dash_image():
125: (4)                fig, ax = plt.subplots()
126: (4)                np.random.seed(0)
127: (4)                ax.contour(np.random.randn(20, 30), linestyles=[(0, (3, 3))])
128: (0)            @image_comparison(['marker_fill_styles.png'], remove_text=True)
129: (0)            def test_marker_fill_styles():
130: (4)                colors = itertools.cycle([[0, 0, 1], 'g', '#ff0000', 'c', 'm', 'y',
131: (30)                                         np.array([0, 0, 0])])
132: (4)                altcolor = 'lightgreen'
133: (4)                y = np.array([1, 1])
134: (4)                x = np.array([0, 9])
135: (4)                fig, ax = plt.subplots()
136: (4)                for j, marker in enumerate("ov^<>8sp*hHDdPX"):
137: (8)                    for i, fs in enumerate(mlines.Line2D.fillStyles):
138: (12)                       color = next(colors)
139: (12)                       ax.plot(j * 10 + x, y + i + .5 * (j % 2),
140: (20)                               marker=marker,
```

```
141: (20)                                markersize=20,
142: (20)                                markerfacecoloralt=altcolor,
143: (20)                                fillstyle=fs,
144: (20)                                label=fs,
145: (20)                                linewidth=5,
146: (20)                                color=color,
147: (20)                                markeredgecolor=color,
148: (20)                                markeredgewidth=2)
149: (4)                ax.set_ylim([0, 7.5])
150: (4)                ax.set_xlim([-5, 155])
151: (0)            def test_markerfacecolor_fillstyle():
152: (4)                """Test that markerfacecolor does not override fillstyle='none'."""
153: (4)                l, = plt.plot([1, 3, 2], marker=MarkerStyle('o', fillstyle='none'),
154: (18)                              markerfacecolor='red')
155: (4)                assert l.get_fillstyle() == 'none'
156: (4)                assert l.get_markerfacecolor() == 'none'
157: (0)            @image_comparison(['scaled_lines'], style='default')
158: (0)            def test_lw_scaling():
159: (4)                th = np.linspace(0, 32)
160: (4)                fig, ax = plt.subplots()
161: (4)                lins_styles = ['dashed', 'dotted', 'dashdot']
162: (4)                cy = cycler(matplotlib.rcParams['axes.prop_cycle'])
163: (4)                for j, (ls, sty) in enumerate(zip(lins_styles, cy)):
164: (8)                    for lw in np.linspace(.5, 10, 10):
165: (12)                       ax.plot(th, j*np.ones(50) + .1 * lw, linestyle=ls, lw=lw, **sty)
166: (0)            def test_is_sorted_and_has_non_nan():
167: (4)                assert _path.is_sorted_and_has_non_nan(np.array([1, 2, 3]))
168: (4)                assert _path.is_sorted_and_has_non_nan(np.array([1, np.nan, 3]))
169: (4)                assert not _path.is_sorted_and_has_non_nan([3, 5] + [np.nan] * 100 + [0,
2])
170: (4)                assert not _path.is_sorted_and_has_non_nan(np.array([33554432, 65536],
">i4"))
171: (4)                n = 2 * mlines.Line2D._subslice_optim_min_size
172: (4)                plt.plot([np.nan] * n, range(n))
173: (0)            @check_figures_equal()
174: (0)            def test_step_markers(fig_test, fig_ref):
175: (4)                fig_test.subplots().step([0, 1], "-o")
176: (4)                fig_ref.subplots().plot([0, 0, 1], [0, 1, 1], "-o", markevery=[0, 2])
177: (0)            @pytest.mark.parametrize("parent", ["figure", "axes"])
178: (0)            @check_figures_equal(extensions=('png',))
179: (0)            def test_markevery(fig_test, fig_ref, parent):
180: (4)                np.random.seed(42)
181: (4)                x = np.linspace(0, 1, 14)
182: (4)                y = np.random.rand(len(x))
183: (4)                cases_test = [None, 4, (2, 5), [1, 5, 11],
184: (18)                              [0, -1], slice(5, 10, 2),
185: (18)                              np.arange(len(x))[y > 0.5],
186: (18)                              0.3, (0.3, 0.4)]
187: (4)                cases_ref = ["11111111111111", "10001000100010", "00100001000010",
188: (17)                            "01000100000100", "10000000000001", "00000101010000",
189: (17)                            "01110001110110", "11011011011110", "01010011011101"]
190: (4)                if parent == "figure":
191: (8)                    cases_test = cases_test[:-2]
192: (8)                    cases_ref = cases_ref[:-2]
193: (8)                    def add_test(x, y, *, markevery):
194: (12)                       fig_test.add_artist(
195: (16)                           mlines.Line2D(x, y, marker="o", markevery=markevery))
196: (8)                    def add_ref(x, y, *, markevery):
197: (12)                       fig_ref.add_artist(
198: (16)                           mlines.Line2D(x, y, marker="o", markevery=markevery))
199: (4)                elif parent == "axes":
200: (8)                    axs_test = iter(fig_test.subplots(3, 3).flat)
201: (8)                    axs_ref = iter(fig_ref.subplots(3, 3).flat)
202: (8)                    def add_test(x, y, *, markevery):
203: (12)                       next(axs_test).plot(x, y, "-gD", markevery=markevery)
204: (8)                    def add_ref(x, y, *, markevery):
205: (12)                       next(axs_ref).plot(x, y, "-gD", markevery=markevery)
206: (4)                for case in cases_test:
207: (8)                    add_test(x, y, markevery=case)
```

```
(4)     for case in cases_ref:
(8)         me = np.array(list(case)).astype(int).astype(bool)
(8)         add_ref(x, y, markevery=me)
(0) def test_markevery_figure_line_unsupported_relsize():
(4)     fig = plt.figure()
(4)     fig.add_artist(mlines.Line2D([0, 1], [0, 1], marker="o", markevery=.5))
(4)     with pytest.raises(ValueError):
(8)         fig.canvas.draw()
(0) def test_marker_as_markerstyle():
(4)     fig, ax = plt.subplots()
(4)     line, = ax.plot([2, 4, 3], marker=MarkerStyle("D"))
(4)     fig.canvas.draw()
(4)     assert line.get_marker() == "D"
(4)     line.set_marker("s")
(4)     fig.canvas.draw()
(4)     line.set_marker(MarkerStyle("o"))
(4)     fig.canvas.draw()
(4)     triangle1 = Path._create_closed([[-1, -1], [1, -1], [0, 2]])
(4)     line2, = ax.plot([1, 3, 2], marker=MarkerStyle(triangle1), ms=22)
(4)     line3, = ax.plot([0, 2, 1], marker=triangle1, ms=22)
(4)     assert_array_equal(line2.get_marker().vertices, triangle1.vertices)
(4)     assert_array_equal(line3.get_marker().vertices, triangle1.vertices)
(0) @image_comparison(['striped_line.png'], remove_text=True, style='mpl20')
(0) def test_striped_lines():
(4)     rng = np.random.default_rng(19680801)
(4)     _, ax = plt.subplots()
(4)     ax.plot(rng.uniform(size=12), color='orange', gapcolor='blue',
(12)            linestyle='--', lw=5, label=' ')
(4)     ax.plot(rng.uniform(size=12), color='red', gapcolor='black',
(12)            linestyle=(0, (2, 5, 4, 2)), lw=5, label=' ', alpha=0.5)
(4)     ax.legend(handlelength=5)
(0) @check_figures_equal()
(0) def test_odd_dashes(fig_test, fig_ref):
(4)     fig_test.add_subplot().plot([1, 2], dashes=[1, 2, 3])
(4)     fig_ref.add_subplot().plot([1, 2], dashes=[1, 2, 3, 1, 2, 3])
(0) def test_picking():
(4)     fig, ax = plt.subplots()
(4)     mouse_event = SimpleNamespace(x=fig.bbox.width // 2,
(34)                                  y=fig.bbox.height // 2 + 15)
(4)     l0, = ax.plot([0, 1], [0, 1], picker=True)
(4)     found, indices = l0.contains(mouse_event)
(4)     assert not found
(4)     l1, = ax.plot([0, 1], [0, 1], picker=True, pickradius=20)
(4)     found, indices = l1.contains(mouse_event)
(4)     assert found
(4)     assert_array_equal(indices['ind'], [0])
(4)     l2, = ax.plot([0, 1], [0, 1], picker=True)
(4)     found, indices = l2.contains(mouse_event)
(4)     assert not found
(4)     l2.set_pickradius(20)
(4)     found, indices = l2.contains(mouse_event)
(4)     assert found
(4)     assert_array_equal(indices['ind'], [0])
(0) @check_figures_equal()
(0) def test_input_copy(fig_test, fig_ref):
(4)     t = np.arange(0, 6, 2)
(4)     l, = fig_test.add_subplot().plot(t, t, ".-")
(4)     t[:] = range(3)
(4)     l.set_drawstyle("steps")
(4)     fig_ref.add_subplot().plot([0, 2, 4], [0, 2, 4], ".-", drawstyle="steps")
(0) @check_figures_equal(extensions=["png"])
(0) def test_markevery_prop_cycle(fig_test, fig_ref):
(4)     """Test that we can set markevery prop_cycle."""
(4)     cases = [None, 8, (30, 8), [16, 24, 30], [0, -1],
(13)             slice(100, 200, 3), 0.1, 0.3, 1.5,
(13)             (0.0, 0.1), (0.45, 0.1)]
(4)     cmap = mpl.colormaps['jet']
(4)     colors = cmap(np.linspace(0.2, 0.8, len(cases)))
(4)     x = np.linspace(-1, 1)
```

```
277: (4)                y = 5 * x**2
278: (4)                axs = fig_ref.add_subplot()
279: (4)                for i, markevery in enumerate(cases):
280: (8)                    axs.plot(y - i, 'o-', markevery=markevery, color=colors[i])
281: (4)                matplotlib.rcParams['axes.prop_cycle'] = cycler(markevery=cases,
282: (52)                                                               color=colors)
283: (4)                ax = fig_test.add_subplot()
284: (4)                for i, _ in enumerate(cases):
285: (8)                    ax.plot(y - i, 'o-')
286: (0)            def test_axline_setters():
287: (4)                fig, ax = plt.subplots()
288: (4)                line1 = ax.axline((.1, .1), slope=0.6)
289: (4)                line2 = ax.axline((.1, .1), (.8, .4))
290: (4)                line1.set_xy1(.2, .3)
291: (4)                line1.set_slope(2.4)
292: (4)                line2.set_xy1(.3, .2)
293: (4)                line2.set_xy2(.6, .8)
294: (4)                assert line1.get_xy1() == (.2, .3)
295: (4)                assert line1.get_slope() == 2.4
296: (4)                assert line2.get_xy1() == (.3, .2)
297: (4)                assert line2.get_xy2() == (.6, .8)
298: (4)                with pytest.raises(ValueError,
299: (23)                                  match="Cannot set an 'xy2' value while 'slope' is
set"):
300: (8)                    line1.set_xy2(.2, .3)
301: (4)                with pytest.raises(ValueError,
302: (23)                                  match="Cannot set a 'slope' value while 'xy2' is set"):
303: (8)                    line2.set_slope(3)

----------------------------------------

File 193 - test_marker.py:

1: (0)              import numpy as np
2: (0)              import matplotlib.pyplot as plt
3: (0)              from matplotlib import markers
4: (0)              from matplotlib.path import Path
5: (0)              from matplotlib.testing.decorators import check_figures_equal
6: (0)              from matplotlib.transforms import Affine2D
7: (0)              import pytest
8: (0)              def test_marker_fillstyle():
9: (4)                  marker_style = markers.MarkerStyle(marker='o', fillstyle='none')
10: (4)                 assert marker_style.get_fillstyle() == 'none'
11: (4)                 assert not marker_style.is_filled()
12: (0)             @pytest.mark.parametrize('marker', [
13: (4)                 'o',
14: (4)                 'x',
15: (4)                 '',
16: (4)                 'None',
17: (4)                 r'$\frac{1}{2}$',
18: (4)                 "$\u266B$",
19: (4)                 1,
20: (4)                 markers.TICKLEFT,
21: (4)                 [[-1, 0], [1, 0]],
22: (4)                 np.array([[-1, 0], [1, 0]]),
23: (4)                 Path([[0, 0], [1, 0]], [Path.MOVETO, Path.LINETO]),
24: (4)                 (5, 0),  # a pentagon
25: (4)                 (7, 1),  # a 7-pointed star
26: (4)                 (5, 2),  # asterisk
27: (4)                 (5, 0, 10),  # a pentagon, rotated by 10 degrees
28: (4)                 (7, 1, 10),  # a 7-pointed star, rotated by 10 degrees
29: (4)                 (5, 2, 10),  # asterisk, rotated by 10 degrees
30: (4)                 markers.MarkerStyle('o'),
31: (0)             ])
32: (0)             def test_markers_valid(marker):
33: (4)                 markers.MarkerStyle(marker)
34: (0)             @pytest.mark.parametrize('marker', [
35: (4)                 'square',  # arbitrary string
36: (4)                 np.array([[-0.5, 0, 1, 2, 3]]),  # 1D array
```

```
 37: (4)                 (1,),
 38: (4)                 (5, 3),  # second parameter of tuple must be 0, 1, or 2
 39: (4)                 (1, 2, 3, 4),
 40: (0)             ])
 41: (0)             def test_markers_invalid(marker):
 42: (4)                 with pytest.raises(ValueError):
 43: (8)                     markers.MarkerStyle(marker)
 44: (0)             class UnsnappedMarkerStyle(markers.MarkerStyle):
 45: (4)                 """
 46: (4)                 A MarkerStyle where the snap threshold is force-disabled.
 47: (4)                 This is used to compare to polygon/star/asterisk markers which do not have
 48: (4)                 any snap threshold set.
 49: (4)                 """
 50: (4)                 def _recache(self):
 51: (8)                     super()._recache()
 52: (8)                     self._snap_threshold = None
 53: (0)             @check_figures_equal()
 54: (0)             def test_poly_marker(fig_test, fig_ref):
 55: (4)                 ax_test = fig_test.add_subplot()
 56: (4)                 ax_ref = fig_ref.add_subplot()
 57: (4)                 size = 20**2
 58: (4)                 ax_test.scatter([0], [0], marker=(4, 0, 45), s=size)
 59: (4)                 ax_ref.scatter([0], [0], marker='s', s=size/2)
 60: (4)                 ax_test.scatter([1], [1], marker=(4, 0), s=size)
 61: (4)                 ax_ref.scatter([1], [1], marker=UnsnappedMarkerStyle('D'), s=size/2)
 62: (4)                 ax_test.scatter([1], [1.5], marker=(4, 0, 0), s=size)
 63: (4)                 ax_ref.scatter([1], [1.5], marker=UnsnappedMarkerStyle('D'), s=size/2)
 64: (4)                 ax_test.scatter([2], [2], marker=(5, 0), s=size)
 65: (4)                 ax_ref.scatter([2], [2], marker=UnsnappedMarkerStyle('p'), s=size)
 66: (4)                 ax_test.scatter([2], [2.5], marker=(5, 0, 0), s=size)
 67: (4)                 ax_ref.scatter([2], [2.5], marker=UnsnappedMarkerStyle('p'), s=size)
 68: (4)                 ax_test.scatter([3], [3], marker=(6, 0), s=size)
 69: (4)                 ax_ref.scatter([3], [3], marker='h', s=size)
 70: (4)                 ax_test.scatter([3], [3.5], marker=(6, 0, 0), s=size)
 71: (4)                 ax_ref.scatter([3], [3.5], marker='h', s=size)
 72: (4)                 ax_test.scatter([4], [4], marker=(6, 0, 30), s=size)
 73: (4)                 ax_ref.scatter([4], [4], marker='H', s=size)
 74: (4)                 ax_test.scatter([5], [5], marker=(8, 0, 22.5), s=size)
 75: (4)                 ax_ref.scatter([5], [5], marker=UnsnappedMarkerStyle('8'), s=size)
 76: (4)                 ax_test.set(xlim=(-0.5, 5.5), ylim=(-0.5, 5.5))
 77: (4)                 ax_ref.set(xlim=(-0.5, 5.5), ylim=(-0.5, 5.5))
 78: (0)             def test_star_marker():
 79: (4)                 size = 20**2
 80: (4)                 fig, ax = plt.subplots()
 81: (4)                 ax.scatter([0], [0], marker=(5, 1), s=size)
 82: (4)                 ax.scatter([1], [1], marker=(5, 1, 0), s=size)
 83: (4)                 ax.set(xlim=(-0.5, 0.5), ylim=(-0.5, 1.5))
 84: (0)             @check_figures_equal(tol=1.45)
 85: (0)             def test_asterisk_marker(fig_test, fig_ref, request):
 86: (4)                 ax_test = fig_test.add_subplot()
 87: (4)                 ax_ref = fig_ref.add_subplot()
 88: (4)                 size = 20**2
 89: (4)                 def draw_ref_marker(y, style, size):
 90: (8)                     ax_ref.scatter([y], [y], marker=UnsnappedMarkerStyle(style), s=size)
 91: (8)                     if request.getfixturevalue('ext') == 'png':
 92: (12)                        ax_ref.scatter([y], [y], marker=UnsnappedMarkerStyle(style),
 93: (27)                                       s=size)
 94: (4)                 ax_test.scatter([0], [0], marker=(4, 2), s=size)
 95: (4)                 draw_ref_marker(0, '+', size)
 96: (4)                 ax_test.scatter([0.5], [0.5], marker=(4, 2, 0), s=size)
 97: (4)                 draw_ref_marker(0.5, '+', size)
 98: (4)                 ax_test.scatter([1], [1], marker=(4, 2, 45), s=size)
 99: (4)                 draw_ref_marker(1, 'x', size/2)
100: (4)                 ax_test.set(xlim=(-0.5, 1.5), ylim=(-0.5, 1.5))
101: (4)                 ax_ref.set(xlim=(-0.5, 1.5), ylim=(-0.5, 1.5))
102: (0)             @check_figures_equal(extensions=['png'], tol=1.86)
103: (0)             def test_text_marker(fig_ref, fig_test):
104: (4)                 ax_ref = fig_ref.add_subplot()
105: (4)                 ax_test = fig_test.add_subplot()
```

```
106: (4)                ax_ref.plot(0, 0, marker=r'o', markersize=100, markeredgewidth=0)
107: (4)                ax_test.plot(0, 0, marker=r'$\bullet$', markersize=100, markeredgewidth=0)
108: (0)            @check_figures_equal()
109: (0)            def test_marker_clipping(fig_ref, fig_test):
110: (4)                marker_count = len(markers.MarkerStyle.markers)
111: (4)                marker_size = 50
112: (4)                ncol = 7
113: (4)                nrow = marker_count // ncol + 1
114: (4)                width = 2 * marker_size * ncol
115: (4)                height = 2 * marker_size * nrow * 2
116: (4)                fig_ref.set_size_inches((width / fig_ref.dpi, height / fig_ref.dpi))
117: (4)                ax_ref = fig_ref.add_axes([0, 0, 1, 1])
118: (4)                fig_test.set_size_inches((width / fig_test.dpi, height / fig_ref.dpi))
119: (4)                ax_test = fig_test.add_axes([0, 0, 1, 1])
120: (4)                for i, marker in enumerate(markers.MarkerStyle.markers):
121: (8)                    x = i % ncol
122: (8)                    y = i // ncol * 2
123: (8)                    ax_ref.plot([x, x], [y, y + 1], c='k', linestyle='-', lw=3)
124: (8)                    ax_ref.plot(x, y, c='k',
125: (20)                               marker=marker, markersize=marker_size, markeredgewidth=10,
126: (20)                               fillstyle='full', markerfacecolor='white')
127: (8)                    ax_ref.plot(x, y + 1, c='k',
128: (20)                               marker=marker, markersize=marker_size, markeredgewidth=10,
129: (20)                               fillstyle='full', markerfacecolor='white')
130: (8)                    ax_test.plot([x, x], [y, y + 1], c='k', linestyle='-', lw=3,
131: (21)                                marker=marker, markersize=marker_size,
markeredgewidth=10,
132: (21)                                fillstyle='full', markerfacecolor='white')
133: (4)                ax_ref.set(xlim=(-0.5, ncol), ylim=(-0.5, 2 * nrow))
134: (4)                ax_test.set(xlim=(-0.5, ncol), ylim=(-0.5, 2 * nrow))
135: (4)                ax_ref.axis('off')
136: (4)                ax_test.axis('off')
137: (0)            def test_marker_init_transforms():
138: (4)                """Test that initializing marker with transform is a simple addition."""
139: (4)                marker = markers.MarkerStyle("o")
140: (4)                t = Affine2D().translate(1, 1)
141: (4)                t_marker = markers.MarkerStyle("o", transform=t)
142: (4)                assert marker.get_transform() + t == t_marker.get_transform()
143: (0)            def test_marker_init_joinstyle():
144: (4)                marker = markers.MarkerStyle("*")
145: (4)                styled_marker = markers.MarkerStyle("*", joinstyle="round")
146: (4)                assert styled_marker.get_joinstyle() == "round"
147: (4)                assert marker.get_joinstyle() != "round"
148: (0)            def test_marker_init_captyle():
149: (4)                marker = markers.MarkerStyle("*")
150: (4)                styled_marker = markers.MarkerStyle("*", capstyle="round")
151: (4)                assert styled_marker.get_capstyle() == "round"
152: (4)                assert marker.get_capstyle() != "round"
153: (0)            @pytest.mark.parametrize("marker,transform,expected", [
154: (4)                (markers.MarkerStyle("o"), Affine2D().translate(1, 1),
155: (8)                    Affine2D().translate(1, 1)),
156: (4)                (markers.MarkerStyle("o", transform=Affine2D().translate(1, 1)),
157: (8)                    Affine2D().translate(1, 1), Affine2D().translate(2, 2)),
158: (4)                (markers.MarkerStyle("$||$", transform=Affine2D().translate(1, 1)),
159: (5)                 Affine2D().translate(1, 1), Affine2D().translate(2, 2)),
160: (4)                (markers.MarkerStyle(
161: (8)                    markers.TICKLEFT, transform=Affine2D().translate(1, 1)),
162: (8)                    Affine2D().translate(1, 1), Affine2D().translate(2, 2)),
163: (0)            ])
164: (0)            def test_marker_transformed(marker, transform, expected):
165: (4)                new_marker = marker.transformed(transform)
166: (4)                assert new_marker is not marker
167: (4)                assert new_marker.get_user_transform() == expected
168: (4)                assert marker._user_transform is not new_marker._user_transform
169: (0)            def test_marker_rotated_invalid():
170: (4)                marker = markers.MarkerStyle("o")
171: (4)                with pytest.raises(ValueError):
172: (8)                    new_marker = marker.rotated()
173: (4)                with pytest.raises(ValueError):
```

```
174: (8)                    new_marker = marker.rotated(deg=10, rad=10)
175: (0)            @pytest.mark.parametrize("marker,deg,rad,expected", [
176: (4)                (markers.MarkerStyle("o"), 10, None, Affine2D().rotate_deg(10)),
177: (4)                (markers.MarkerStyle("o"), None, 0.01, Affine2D().rotate(0.01)),
178: (4)                (markers.MarkerStyle("o", transform=Affine2D().translate(1, 1)),
179: (8)                    10, None, Affine2D().translate(1, 1).rotate_deg(10)),
180: (4)                (markers.MarkerStyle("o", transform=Affine2D().translate(1, 1)),
181: (8)                    None, 0.01, Affine2D().translate(1, 1).rotate(0.01)),
182: (4)                (markers.MarkerStyle("$|||$", transform=Affine2D().translate(1, 1)),
183: (6)                  10, None, Affine2D().translate(1, 1).rotate_deg(10)),
184: (4)                (markers.MarkerStyle(
185: (8)                    markers.TICKLEFT, transform=Affine2D().translate(1, 1)),
186: (8)                    10, None, Affine2D().translate(1, 1).rotate_deg(10)),
187: (0)            ])
188: (0)            def test_marker_rotated(marker, deg, rad, expected):
189: (4)                new_marker = marker.rotated(deg=deg, rad=rad)
190: (4)                assert new_marker is not marker
191: (4)                assert new_marker.get_user_transform() == expected
192: (4)                assert marker._user_transform is not new_marker._user_transform
193: (0)            def test_marker_scaled():
194: (4)                marker = markers.MarkerStyle("1")
195: (4)                new_marker = marker.scaled(2)
196: (4)                assert new_marker is not marker
197: (4)                assert new_marker.get_user_transform() == Affine2D().scale(2)
198: (4)                assert marker._user_transform is not new_marker._user_transform
199: (4)                new_marker = marker.scaled(2, 3)
200: (4)                assert new_marker is not marker
201: (4)                assert new_marker.get_user_transform() == Affine2D().scale(2, 3)
202: (4)                assert marker._user_transform is not new_marker._user_transform
203: (4)                marker = markers.MarkerStyle("1", transform=Affine2D().translate(1, 1))
204: (4)                new_marker = marker.scaled(2)
205: (4)                assert new_marker is not marker
206: (4)                expected = Affine2D().translate(1, 1).scale(2)
207: (4)                assert new_marker.get_user_transform() == expected
208: (4)                assert marker._user_transform is not new_marker._user_transform
209: (0)            def test_alt_transform():
210: (4)                m1 = markers.MarkerStyle("o", "left")
211: (4)                m2 = markers.MarkerStyle("o", "left", Affine2D().rotate_deg(90))
212: (4)                assert m1.get_alt_transform().rotate_deg(90) == m2.get_alt_transform()

----------------------------------------

File 194 - test_mathtext.py:

1: (0)              from __future__ import annotations
2: (0)              import io
3: (0)              from pathlib import Path
4: (0)              import platform
5: (0)              import re
6: (0)              import shlex
7: (0)              from xml.etree import ElementTree as ET
8: (0)              from typing import Any
9: (0)              import numpy as np
10: (0)             from packaging.version import parse as parse_version
11: (0)             import pyparsing
12: (0)             import pytest
13: (0)             import matplotlib as mpl
14: (0)             from matplotlib.testing.decorators import check_figures_equal,
image_comparison
15: (0)             import matplotlib.pyplot as plt
16: (0)             from matplotlib import mathtext, _mathtext
17: (0)             pyparsing_version = parse_version(pyparsing.__version__)
18: (0)             math_tests = [
19: (4)                 r'$a+b+\dot s+\dot{s}+\ldots$',
20: (4)                 r'$x\hspace{-0.2}\doteq\hspace{-0.2}y$',
21: (4)                 r'\$100.00 $\alpha \_$',
22: (4)                 r'$\frac{\$100.00}{y}$',
23: (4)                 r'$x   y$',
24: (4)                 r'$x+y\ x=y\ x<y\ x:y\ x,y\ x@y$',
```

```
25: (4)                 r'$100\%y\ x*y\ x/y x\$y$',
26: (4)                 r'$x\leftarrow y\ x\forall y\ x-y$',
27: (4)                 r'$x \sf x \bf x {\cal X} \rm x$',
28: (4)                 r'$x\ x\,x\;x\quad x\qquad x\!x\hspace{ 0.5 }y$',
29: (4)                 r'$\{ \rm braces \}$',
30: (4)                 r'$\left[\left\lfloor\frac{5}{\frac{\left(3\right)}{4}} y\right)\right]$',
31: (4)                 r'$\left(x\right)$',
32: (4)                 r'$\sin(x)$',
33: (4)                 r'$x_2$',
34: (4)                 r'$x^2$',
35: (4)                 r'$x^2_y$',
36: (4)                 r'$x_y^2$',
37: (4)                 (r'$\sum _{\genfrac{}{}{0}{}{0\leq i\leq m}{0<j<n}}f\left(i,j\right)'
38: (5)                  r'\mathcal{R}\prod_{i=\alpha_{i+1}}^\infty a_i \sin(2 \pi f x_i)'
39: (5)                  r"\sqrt[2]{\prod^\frac{x}{2\pi^2}_\infty}$"),
40: (4)                 r'$x = \frac{x+\frac{5}{2}}{\frac{y+3}{8}}$',
41: (4)                 r'$dz/dt = \gamma x^2 + {\rm sin}(2\pi y+\phi)$',
42: (4)                 r'Foo: $\alpha_{i+1}^j = {\rm sin}(2\pi f_j t_i) e^{-5 t_i/\tau}$',
43: (4)                 None,
44: (4)                 r'Variable $i$ is good',
45: (4)                 r'$\Delta_i^j$',
46: (4)                 r'$\Delta^j_{i+1}$',
47: (4)                 r'$\ddot{o}\acute{e}\grave{e}\hat{O}\breve{\imath}\tilde{n}\vec{q}$',
48: (4)                 r"$\arccos((x^i))$",
49: (4)                 r"$\gamma = \frac{x=\frac{6}{8}}{y} \delta$",
50: (4)                 r'$\limsup_{x\to\infty}$',
51: (4)                 None,
52: (4)                 r"$f'\quad f'''(x)\quad ''/\mathrm{yr}$",
53: (4)                 r'$\frac{x_2888}{y}$',
54: (4)                 r"$\sqrt[3]{\frac{X_2}{Y}}=5$",
55: (4)                 None,
56: (4)                 r"$\sqrt[3]{x}=5$",
57: (4)                 r'$\frac{X}{\frac{X}{Y}}$',
58: (4)                 r"$W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = U^{3\beta}_{\delta_1 \rho_1} +
\frac{1}{8 \pi 2} \int^{\alpha_2}_{\alpha_2} d \alpha^\prime_2 \left[\frac{ U^{2\beta}_{\delta_1
\rho_1} - \alpha^\prime_2U^{1\beta}_{\rho_1 \sigma_2} }{U^{0\beta}_{\rho_1 \sigma_2}}\right]$",
59: (4)                 r'$\mathcal{H} = \int d \tau \left(\epsilon E^2 + \mu H^2\right)$',
60: (4)                 r'$\widehat{abc}\widetilde{def}$',
61: (4)                 '$\\Gamma \\Delta \\Theta \\Lambda \\Xi \\Pi \\Sigma \\Upsilon \\Phi \\Psi
\\Omega$',
62: (4)                 '$\\alpha \\beta \\gamma \\delta \\epsilon \\zeta \\eta \\theta \\iota
\\lambda \\mu \\nu \\xi \\pi \\kappa \\rho \\sigma \\tau \\upsilon \\phi \\chi \\psi$',
63: (4)                 r'${x}^{2}{y}^{2}$',
64: (4)                 r'${}_{2}F_{3}$',
65: (4)                 r'$\frac{x+{y}^{2}}{k+1}$',
66: (4)                 r'$x+{y}^{\frac{2}{k+1}}$',
67: (4)                 r'$\frac{a}{b/2}$',
68: (4)                 r'${a}_{0}+\frac{1}{{a}_{1}+\frac{1}{{a}_{2}+\frac{1}{{a}_{3}+\frac{1}
{{a}_{4}}}}}}$',
69: (4)                 r'${a}_{0}+\frac{1}{{a}_{1}+\frac{1}{{a}_{2}+\frac{1}{{a}_{3}+\frac{1}
{{a}_{4}}}}}}$',
70: (4)                 r'$\binom{n}{k/2}$',
71: (4)                 r'$\binom{p}{2}{x}^{2}{y}^{p-2}-\frac{1}{1-x}\frac{1}{1-{x}^{2}}$',
72: (4)                 r'${x}^{2y}$',
73: (4)                 r'$\sum _{i=1}^{p}\sum _{j=1}^{q}\sum _{k=1}^{r}{a}_{ij}{b}_{jk}
{c}_{ki}$',
74: (4)                 r'$\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+x}}}}}}}$',
75: (4)                 r'$\left(\frac{{\partial }^{2}}{\partial {x}^{2}}+\frac{{\partial }^{2}}
{\partial {y}^{2}}\right){|\varphi \left(x+iy\right)|}^{2}=0$',
76: (4)                 r'${2}^{{2}^{{2}^{x}}}$',
77: (4)                 r'${\int }_{1}^{x}\frac{\mathrm{dt}}{t}$',
78: (4)                 r'$\int {\int }_{D}\mathrm{dx} \mathrm{dy}$',
79: (4)                 r'${y}_{{x}^{2}}$',
80: (4)                 r'${y}_{{x}_{2}}$',
81: (4)                 r'${x}_{92}^{31415}+\pi $',
82: (4)                 r'${x}_{{y}_{b}^{a}}^{{z}_{c}^{d}}$',
83: (4)                 r'${y}_{3}^{\prime \prime \prime }$',
84: (4)                 r"$\left( \xi \left( 1 - \xi \right) \right)$",  # Bug 2969451
85: (4)                 r"$\left(2 \, a=b\right)$",  # Sage bug #8125
```

```
86: (4)                 r"$? ! &$",  # github issue #466
87: (4)                 None,
88: (4)                 None,
89: (4)                 r"$\left\Vert \frac{a}{b} \right\Vert \left\vert \frac{a}{b} \right\vert
\left\| \frac{a}{b}\right\| \left| \frac{a}{b} \right| \Vert a \Vert \vert b \vert \| a \| | b
|$",
90: (4)                 r'$\mathring{A}  \AA$',
91: (4)                 r'$M \, M \thinspace M \/ M \> M \: M \; M \ M \enspace M \quad M \qquad M
\! M$',
92: (4)                 r'$\Cap$ $\Cup$ $\leftharpoonup$ $\barwedge$ $\rightharpoonup$',
93: (4)                 r'$\hspace{-0.2}\dotplus\hspace{-0.2}$ $\hspace{-0.2}\doteq\hspace{-0.2}$
$\hspace{-0.2}\doteqdot\hspace{-0.2}$ $\ddots$',
94: (4)                 r'$xyz^kx_kx^py^{p-2} d_i^jb_jc_kd x^j_i E^0 E^0_u$',  # github issue
#4873
95: (4)                 r'${xyz}^k{x}_{k}{x}^{p}{y}^{p-2} {d}_{i}^{j}{b}_{j}{c}_{k}{d} {x}^{j}_{i}
{E}^{0}{E}^0_u$',
96: (4)                 r'${\int}_x^x x\oint_x^x x\int_{X}^{X}x\int_x x \int^x x \int_{x}
x\int^{x}{\int}_{x} x{\int}^{x}_{x}x$',
97: (4)                 r'testing$^{123}$',
98: (4)                 None,
99: (4)                 r'$6-2$; $-2$; $ -2$; ${-2}$; ${  -2}$; $20^{+3}_{-2}$',
100: (4)                r'$\overline{\omega}^x \frac{1}{2}_0^x$',  # github issue #5444
101: (4)                r'$,$ $.$ $1{,}234{, }567{ , }890$ and $1,234,567,890$',  # github issue
5799
102: (4)                r'$\left(X\right)_{a}^{b}$',  # github issue 7615
103: (4)                r'$\dfrac{\$100.00}{y}$',  # github issue #1888
104: (0)            ]
105: (0)            svgastext_math_tests = [
106: (4)                r'$-$-',
107: (0)            ]
108: (0)            lightweight_math_tests = [
109: (4)                r'$\sqrt[ab]{123}$',  # github issue #8665
110: (4)                r'$x \overset{f}{\rightarrow} \overset{f}{x} \underset{xx}{ff}
\overset{xx}{ff} \underset{f}{x} \underset{f}{\leftarrow} x$',  # github issue #18241
111: (4)                r'$\sum x\quad\sum^nx\quad\sum_nx\quad\sum_n^nx\quad\prod
x\quad\prod^nx\quad\prod_nx\quad\prod_n^nx$',  # GitHub issue 18085
112: (4)                r'$1.$ $2.$ $19680801.$ $a.$ $b.$ $mpl.$',
113: (4)                r'$\text{text}_{\text{sub}}^{\text{sup}} + \text{\$foo\$} +
\frac{\text{num}}{\mathbf{\text{den}}}\text{with space, curly brackets \{\}, and dash -}$',
114: (4)                r'$\boldsymbol{abcde} \boldsymbol{+} \boldsymbol{\Gamma + \Omega}
\boldsymbol{01234} \boldsymbol{\alpha * \beta}$',
115: (4)                r'$\left\lbrace\frac{\left\lbrack A^b_c\right\rbrace}{\left\leftbrace
D^e_f \right\rbrack}\right\rightbrace\ \left\leftparen\max_{x} \left\lgroup \frac{A}
{B}\right\rgroup \right\rightparen$',
116: (4)                r'$\left( a\middle. b \right)$ $\left( \frac{a}{b} \middle\vert x_i \in
P^S \right)$ $\left[ 1 - \middle| a\middle| + \left( x  - \left\lfloor \dfrac{a}{b}\right\rfloor
\right)  \right]$',
117: (4)                r'$\sum_{\substack{k = 1\\ k \neq \lfloor n/2\rfloor}}^{n}P(i,j)
\sum_{\substack{i \neq 0\\ -1 \leq i \leq 3\\ 1 \leq j \leq 5}} F^i(x,y) \sum_{\substack{\left
\lfloor \frac{n}{2} \right\rfloor}} F(n)$',
118: (0)            ]
119: (0)            digits = "0123456789"
120: (0)            uppercase = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
121: (0)            lowercase = "abcdefghijklmnopqrstuvwxyz"
122: (0)            uppergreek = ("\\Gamma \\Delta \\Theta \\Lambda \\Xi \\Pi \\Sigma \\Upsilon
\\Phi \\Psi "
123: (14)                         "\\Omega")
124: (0)            lowergreek = ("\\alpha \\beta \\gamma \\delta \\epsilon \\zeta \\eta \\theta
\\iota "
125: (14)                         "\\lambda \\mu \\nu \\xi \\pi \\kappa \\rho \\sigma \\tau
\\upsilon "
126: (14)                         "\\phi \\chi \\psi")
127: (0)            all = [digits, uppercase, lowercase, uppergreek, lowergreek]
128: (0)            font_test_specs: list[tuple[None | list[str], Any]] = [
129: (4)                ([], all),
130: (4)                (['mathrm'], all),
131: (4)                (['mathbf'], all),
132: (4)                (['mathit'], all),
133: (4)                (['mathtt'], [digits, uppercase, lowercase]),
```

```
134: (4)                (None, 3),
135: (4)                (None, 3),
136: (4)                (None, 3),
137: (4)                (['mathbb'], [digits, uppercase, lowercase,
138: (18)                             r'\Gamma \Pi \Sigma \gamma \pi']),
139: (4)                (['mathrm', 'mathbb'], [digits, uppercase, lowercase,
140: (28)                                       r'\Gamma \Pi \Sigma \gamma \pi']),
141: (4)                (['mathbf', 'mathbb'], [digits, uppercase, lowercase,
142: (28)                                       r'\Gamma \Pi \Sigma \gamma \pi']),
143: (4)                (['mathcal'], [uppercase]),
144: (4)                (['mathfrak'], [uppercase, lowercase]),
145: (4)                (['mathbf', 'mathfrak'], [uppercase, lowercase]),
146: (4)                (['mathscr'], [uppercase, lowercase]),
147: (4)                (['mathsf'], [digits, uppercase, lowercase]),
148: (4)                (['mathrm', 'mathsf'], [digits, uppercase, lowercase]),
149: (4)                (['mathbf', 'mathsf'], [digits, uppercase, lowercase]),
150: (4)                (['mathbfit'], all),
151: (4)                ]
152: (0)            font_tests: list[None | str] = []
153: (0)            for fonts, chars in font_test_specs:
154: (4)                if fonts is None:
155: (8)                    font_tests.extend([None] * chars)
156: (4)                else:
157: (8)                    wrapper = ''.join([
158: (12)                       ' '.join(fonts),
159: (12)                       ' $',
160: (12)                       *(r'\%s{' % font for font in fonts),
161: (12)                       '%s',
162: (12)                       *('}' for font in fonts),
163: (12)                       '$',
164: (8)                    ])
165: (8)                    for set in chars:
166: (12)                       font_tests.append(wrapper % set)
167: (0)            @pytest.fixture
168: (0)            def baseline_images(request, fontset, index, text):
169: (4)                if text is None:
170: (8)                    pytest.skip("test has been removed")
171: (4)                return ['%s_%s_%02d' % (request.param, fontset, index)]
172: (0)            @pytest.mark.parametrize(
173: (4)                'index, text', enumerate(math_tests), ids=range(len(math_tests)))
174: (0)            @pytest.mark.parametrize(
175: (4)                'fontset', ['cm', 'stix', 'stixsans', 'dejavusans', 'dejavuserif'])
176: (0)            @pytest.mark.parametrize('baseline_images', ['mathtext'], indirect=True)
177: (0)            @image_comparison(baseline_images=None,
178: (18)                             tol=0.011 if platform.machine() in ('ppc64le', 's390x') else
0)
179: (0)            def test_mathtext_rendering(baseline_images, fontset, index, text):
180: (4)                mpl.rcParams['mathtext.fontset'] = fontset
181: (4)                fig = plt.figure(figsize=(5.25, 0.75))
182: (4)                fig.text(0.5, 0.5, text,
183: (13)                         horizontalalignment='center', verticalalignment='center')
184: (0)            @pytest.mark.parametrize('index, text', enumerate(svgastext_math_tests),
185: (25)                                    ids=range(len(svgastext_math_tests)))
186: (0)            @pytest.mark.parametrize('fontset', ['cm', 'dejavusans'])
187: (0)            @pytest.mark.parametrize('baseline_images', ['mathtext0'], indirect=True)
188: (0)            @image_comparison(
189: (4)                baseline_images=None, extensions=['svg'],
190: (4)                savefig_kwarg={'metadata': {  # Minimize image size.
191: (8)                    'Creator': None, 'Date': None, 'Format': None, 'Type': None}})
192: (0)            def test_mathtext_rendering_svgastext(baseline_images, fontset, index, text):
193: (4)                mpl.rcParams['mathtext.fontset'] = fontset
194: (4)                mpl.rcParams['svg.fonttype'] = 'none'  # Minimize image size.
195: (4)                fig = plt.figure(figsize=(5.25, 0.75))
196: (4)                fig.patch.set(visible=False)  # Minimize image size.
197: (4)                fig.text(0.5, 0.5, text,
198: (13)                         horizontalalignment='center', verticalalignment='center')
199: (0)            @pytest.mark.parametrize('index, text', enumerate(lightweight_math_tests),
200: (25)                                    ids=range(len(lightweight_math_tests)))
201: (0)            @pytest.mark.parametrize('fontset', ['dejavusans'])
```

```
(0)            @pytest.mark.parametrize('baseline_images', ['mathtext1'], indirect=True)
(0)            @image_comparison(baseline_images=None, extensions=['png'])
(0)            def test_mathtext_rendering_lightweight(baseline_images, fontset, index,
text):
(4)                fig = plt.figure(figsize=(5.25, 0.75))
(4)                fig.text(0.5, 0.5, text, math_fontfamily=fontset,
(13)                         horizontalalignment='center', verticalalignment='center')
(0)            @pytest.mark.parametrize(
(4)                'index, text', enumerate(font_tests), ids=range(len(font_tests)))
(0)            @pytest.mark.parametrize(
(4)                'fontset', ['cm', 'stix', 'stixsans', 'dejavusans', 'dejavuserif'])
(0)            @pytest.mark.parametrize('baseline_images', ['mathfont'], indirect=True)
(0)            @image_comparison(baseline_images=None, extensions=['png'],
(18)                             tol=0.011 if platform.machine() in ('ppc64le', 's390x') else
0)
(0)            def test_mathfont_rendering(baseline_images, fontset, index, text):
(4)                mpl.rcParams['mathtext.fontset'] = fontset
(4)                fig = plt.figure(figsize=(5.25, 0.75))
(4)                fig.text(0.5, 0.5, text,
(13)                         horizontalalignment='center', verticalalignment='center')
(0)            @check_figures_equal(extensions=["png"])
(0)            def test_short_long_accents(fig_test, fig_ref):
(4)                acc_map = _mathtext.Parser._accent_map
(4)                short_accs = [s for s in acc_map if len(s) == 1]
(4)                corresponding_long_accs = []
(4)                for s in short_accs:
(8)                    l, = [l for l in acc_map if len(l) > 1 and acc_map[l] == acc_map[s]]
(8)                    corresponding_long_accs.append(l)
(4)                fig_test.text(0, .5, "$" + "".join(rf"\{s}a" for s in short_accs) + "$")
(4)                fig_ref.text(
(8)                    0, .5, "$" + "".join(fr"\{l} a" for l in corresponding_long_accs) +
"$")
(0)            def test_fontinfo():
(4)                fontpath = mpl.font_manager.findfont("DejaVu Sans")
(4)                font = mpl.ft2font.FT2Font(fontpath)
(4)                table = font.get_sfnt_table("head")
(4)                assert table is not None
(4)                assert table['version'] == (1, 0)
(0)            @pytest.mark.xfail(pyparsing_version.release == (3, 1, 0),
(19)                              reason="Error messages are incorrect for this version")
(0)            @pytest.mark.parametrize(
(4)                'math, msg',
(4)                [
(8)                    (r'$\hspace{}$', r'Expected \hspace{space}'),
(8)                    (r'$\hspace{foo}$', r'Expected \hspace{space}'),
(8)                    (r'$\sinx$', r'Unknown symbol: \sinx'),
(8)                    (r'$\dotx$', r'Unknown symbol: \dotx'),
(8)                    (r'$\frac$', r'Expected \frac{num}{den}'),
(8)                    (r'$\frac{}{}$', r'Expected \frac{num}{den}'),
(8)                    (r'$\binom$', r'Expected \binom{num}{den}'),
(8)                    (r'$\binom{}{}$', r'Expected \binom{num}{den}'),
(8)                    (r'$\genfrac$',
(9)                     r'Expected \genfrac{ldelim}{rdelim}{rulesize}{style}{num}{den}'),
(8)                    (r'$\genfrac{}{}{}{}{}{}$',
(9)                     r'Expected \genfrac{ldelim}{rdelim}{rulesize}{style}{num}{den}'),
(8)                    (r'$\sqrt$', r'Expected \sqrt{value}'),
(8)                    (r'$\sqrt f$', r'Expected \sqrt{value}'),
(8)                    (r'$\overline$', r'Expected \overline{body}'),
(8)                    (r'$\overline{}$', r'Expected \overline{body}'),
(8)                    (r'$\leftF$', r'Expected a delimiter'),
(8)                    (r'$\rightF$', r'Unknown symbol: \rightF'),
(8)                    (r'$\left(\right$', r'Expected a delimiter'),
(8)                    (r'$\left($', re.compile(r'Expected ("|\'\\)\\right["\']')),
(8)                    (r'$\dfrac$', r'Expected \dfrac{num}{den}'),
(8)                    (r'$\dfrac{}{}$', r'Expected \dfrac{num}{den}'),
(8)                    (r'$\overset$', r'Expected \overset{annotation}{body}'),
(8)                    (r'$\underset$', r'Expected \underset{annotation}{body}'),
(8)                    (r'$\foo$', r'Unknown symbol: \foo'),
(8)                    (r'$a^2^2$', r'Double superscript'),
```

```
(8)                     (r'$a_2_2$', r'Double subscript'),
(8)                     (r'$a^2_a^2$', r'Double superscript'),
(8)                     (r'$a = {b$', r"Expected '}'"),
(4)                 ],
(4)                 ids=[
(8)                     'hspace without value',
(8)                     'hspace with invalid value',
(8)                     'function without space',
(8)                     'accent without space',
(8)                     'frac without parameters',
(8)                     'frac with empty parameters',
(8)                     'binom without parameters',
(8)                     'binom with empty parameters',
(8)                     'genfrac without parameters',
(8)                     'genfrac with empty parameters',
(8)                     'sqrt without parameters',
(8)                     'sqrt with invalid value',
(8)                     'overline without parameters',
(8)                     'overline with empty parameter',
(8)                     'left with invalid delimiter',
(8)                     'right with invalid delimiter',
(8)                     'unclosed parentheses with sizing',
(8)                     'unclosed parentheses without sizing',
(8)                     'dfrac without parameters',
(8)                     'dfrac with empty parameters',
(8)                     'overset without parameters',
(8)                     'underset without parameters',
(8)                     'unknown symbol',
(8)                     'double superscript',
(8)                     'double subscript',
(8)                     'super on sub without braces',
(8)                     'unclosed group',
(4)                 ]
(0)             )
(0)             def test_mathtext_exceptions(math, msg):
(4)                 parser = mathtext.MathTextParser('agg')
(4)                 match = re.escape(msg) if isinstance(msg, str) else msg
(4)                 with pytest.raises(ValueError, match=match):
(8)                     parser.parse(math)
(0)             def test_get_unicode_index_exception():
(4)                 with pytest.raises(ValueError):
(8)                     _mathtext.get_unicode_index(r'\foo')
(0)             def test_single_minus_sign():
(4)                 fig = plt.figure()
(4)                 fig.text(0.5, 0.5, '$-$')
(4)                 fig.canvas.draw()
(4)                 t = np.asarray(fig.canvas.renderer.buffer_rgba())
(4)                 assert (t != 0xff).any()  # assert that canvas is not all white.
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_spaces(fig_test, fig_ref):
(4)                 fig_test.text(.5, .5, r"$1\,2\>3\ 4$")
(4)                 fig_ref.text(.5, .5, r"$1\/2\:3~4$")
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_operator_space(fig_test, fig_ref):
(4)                 fig_test.text(0.1, 0.1, r"$\log 6$")
(4)                 fig_test.text(0.1, 0.2, r"$\log(6)$")
(4)                 fig_test.text(0.1, 0.3, r"$\arcsin 6$")
(4)                 fig_test.text(0.1, 0.4, r"$\arcsin|6|$")
(4)                 fig_test.text(0.1, 0.5, r"$\operatorname{op} 6$")  # GitHub issue #553
(4)                 fig_test.text(0.1, 0.6, r"$\operatorname{op}[6]$")
(4)                 fig_test.text(0.1, 0.7, r"$\cos^2$")
(4)                 fig_test.text(0.1, 0.8, r"$\log_2$")
(4)                 fig_test.text(0.1, 0.9, r"$\sin^2 \cos$")  # GitHub issue #17852
(4)                 fig_ref.text(0.1, 0.1, r"$\mathrm{log\,}6$")
(4)                 fig_ref.text(0.1, 0.2, r"$\mathrm{log}(6)$")
(4)                 fig_ref.text(0.1, 0.3, r"$\mathrm{arcsin\,}6$")
(4)                 fig_ref.text(0.1, 0.4, r"$\mathrm{arcsin}|6|$")
(4)                 fig_ref.text(0.1, 0.5, r"$\mathrm{op\,}6$")
(4)                 fig_ref.text(0.1, 0.6, r"$\mathrm{op}[6]$")
```

```
(4)                 fig_ref.text(0.1, 0.7, r"$\mathrm{cos}^2$")
(4)                 fig_ref.text(0.1, 0.8, r"$\mathrm{log}_2$")
(4)                 fig_ref.text(0.1, 0.9, r"$\mathrm{sin}^2 \mathrm{\,cos}$")
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_inverted_delimiters(fig_test, fig_ref):
(4)                 fig_test.text(.5, .5, r"$\left)\right($", math_fontfamily="dejavusans")
(4)                 fig_ref.text(.5, .5, r"$)($", math_fontfamily="dejavusans")
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_genfrac_displaystyle(fig_test, fig_ref):
(4)                 fig_test.text(0.1, 0.1, r"$\dfrac{2x}{3y}$")
(4)                 thickness = _mathtext.TruetypeFonts.get_underline_thickness(
(8)                     None, None, fontsize=mpl.rcParams["font.size"],
(8)                     dpi=mpl.rcParams["savefig.dpi"])
(4)                 fig_ref.text(0.1, 0.1, r"$\genfrac{}{}{%f}{0}{2x}{3y}$" % thickness)
(0)             def test_mathtext_fallback_valid():
(4)                 for fallback in ['cm', 'stix', 'stixsans', 'None']:
(8)                     mpl.rcParams['mathtext.fallback'] = fallback
(0)             def test_mathtext_fallback_invalid():
(4)                 for fallback in ['abc', '']:
(8)                     with pytest.raises(ValueError, match="not a valid fallback font
name"):
(12)                        mpl.rcParams['mathtext.fallback'] = fallback
(0)             @pytest.mark.parametrize(
(4)                 "fallback,fontlist",
(4)                 [("cm", ['DejaVu Sans', 'mpltest', 'STIXGeneral', 'cmr10',
'STIXGeneral']),
(5)                  ("stix", ['DejaVu Sans', 'mpltest', 'STIXGeneral'])])
(0)             def test_mathtext_fallback(fallback, fontlist):
(4)                 mpl.font_manager.fontManager.addfont(
(8)                     str(Path(__file__).resolve().parent / 'mpltest.ttf'))
(4)                 mpl.rcParams["svg.fonttype"] = 'none'
(4)                 mpl.rcParams['mathtext.fontset'] = 'custom'
(4)                 mpl.rcParams['mathtext.rm'] = 'mpltest'
(4)                 mpl.rcParams['mathtext.it'] = 'mpltest:italic'
(4)                 mpl.rcParams['mathtext.bf'] = 'mpltest:bold'
(4)                 mpl.rcParams['mathtext.bfit'] = 'mpltest:italic:bold'
(4)                 mpl.rcParams['mathtext.fallback'] = fallback
(4)                 test_str = r'a$A\AA\breve\gimel$'
(4)                 buff = io.BytesIO()
(4)                 fig, ax = plt.subplots()
(4)                 fig.text(.5, .5, test_str, fontsize=40, ha='center')
(4)                 fig.savefig(buff, format="svg")
(4)                 tspans = (ET.fromstring(buff.getvalue())
(14)                          .findall(".//{http://www.w3.org/2000/svg}tspan[@style]"))
(4)                 char_fonts = [shlex.split(tspan.attrib["style"])[-1] for tspan in tspans]
(4)                 assert char_fonts == fontlist
(4)                 mpl.font_manager.fontManager.ttflist.pop()
(0)             def test_math_to_image(tmp_path):
(4)                 mathtext.math_to_image('$x^2$', tmp_path / 'example.png')
(4)                 mathtext.math_to_image('$x^2$', io.BytesIO())
(4)                 mathtext.math_to_image('$x^2$', io.BytesIO(), color='Maroon')
(0)             @image_comparison(baseline_images=['math_fontfamily_image.png'],
(18)                              savefig_kwarg={'dpi': 40})
(0)             def test_math_fontfamily():
(4)                 fig = plt.figure(figsize=(10, 3))
(4)                 fig.text(0.2, 0.7, r"$This\ text\ should\ have\ one\ font$",
(13)                         size=24, math_fontfamily='dejavusans')
(4)                 fig.text(0.2, 0.3, r"$This\ text\ should\ have\ another$",
(13)                         size=24, math_fontfamily='stix')
(0)             def test_default_math_fontfamily():
(4)                 mpl.rcParams['mathtext.fontset'] = 'cm'
(4)                 test_str = r'abc$abc\alpha$'
(4)                 fig, ax = plt.subplots()
(4)                 text1 = fig.text(0.1, 0.1, test_str, font='Arial')
(4)                 prop1 = text1.get_fontproperties()
(4)                 assert prop1.get_math_fontfamily() == 'cm'
(4)                 text2 = fig.text(0.2, 0.2, test_str, fontproperties='Arial')
(4)                 prop2 = text2.get_fontproperties()
(4)                 assert prop2.get_math_fontfamily() == 'cm'
```

```
404: (4)                fig.draw_without_rendering()
405: (0)            def test_argument_order():
406: (4)                mpl.rcParams['mathtext.fontset'] = 'cm'
407: (4)                test_str = r'abc$abc\alpha$'
408: (4)                fig, ax = plt.subplots()
409: (4)                text1 = fig.text(0.1, 0.1, test_str,
410: (21)                                math_fontfamily='dejavusans', font='Arial')
411: (4)                prop1 = text1.get_fontproperties()
412: (4)                assert prop1.get_math_fontfamily() == 'dejavusans'
413: (4)                text2 = fig.text(0.2, 0.2, test_str,
414: (21)                                math_fontfamily='dejavusans', fontproperties='Arial')
415: (4)                prop2 = text2.get_fontproperties()
416: (4)                assert prop2.get_math_fontfamily() == 'dejavusans'
417: (4)                text3 = fig.text(0.3, 0.3, test_str,
418: (21)                                font='Arial', math_fontfamily='dejavusans')
419: (4)                prop3 = text3.get_fontproperties()
420: (4)                assert prop3.get_math_fontfamily() == 'dejavusans'
421: (4)                text4 = fig.text(0.4, 0.4, test_str,
422: (21)                                fontproperties='Arial', math_fontfamily='dejavusans')
423: (4)                prop4 = text4.get_fontproperties()
424: (4)                assert prop4.get_math_fontfamily() == 'dejavusans'
425: (4)                fig.draw_without_rendering()
426: (0)            def test_mathtext_cmr10_minus_sign():
427: (4)                mpl.rcParams['font.family'] = 'cmr10'
428: (4)                mpl.rcParams['axes.formatter.use_mathtext'] = True
429: (4)                fig, ax = plt.subplots()
430: (4)                ax.plot(range(-1, 1), range(-1, 1))
431: (4)                fig.canvas.draw()
432: (0)            def test_mathtext_operators():
433: (4)                test_str = r'''
434: (4)                \increment \smallin \notsmallowns
435: (4)                \smallowns \QED \rightangle
436: (4)                \smallintclockwise \smallvarointclockwise
437: (4)                \smallointctrcclockwise
438: (4)                \ratio \minuscolon \dotsminusdots
439: (4)                \sinewave \simneqq \nlesssim
440: (4)                \ngtrsim \nlessgtr \ngtrless
441: (4)                \cupleftarrow \oequal \rightassert
442: (4)                \rightModels \hermitmatrix \barvee
443: (4)                \measuredrightangle \varlrtriangle
444: (4)                \equalparallel \npreccurlyeq \nsucccurlyeq
445: (4)                \nsqsubseteq \nsqsupseteq \sqsubsetneq
446: (4)                \sqsupsetneq  \disin \varisins
447: (4)                \isins \isindot \varisinobar
448: (4)                \isinobar \isinvb \isinE
449: (4)                \nisd \varnis \nis
450: (4)                \varniobar \niobar \bagmember
451: (4)                \triangle'''.split()
452: (4)                fig = plt.figure()
453: (4)                for x, i in enumerate(test_str):
454: (8)                    fig.text(0.5, (x + 0.5)/len(test_str), r'${%s}$' % i)
455: (4)                fig.draw_without_rendering()
456: (0)            @check_figures_equal(extensions=["png"])
457: (0)            def test_boldsymbol(fig_test, fig_ref):
458: (4)                fig_test.text(0.1, 0.2, r"$\boldsymbol{\mathrm{abc0123\alpha}}$")
459: (4)                fig_ref.text(0.1, 0.2, r"$\mathrm{abc0123\alpha}$")
```

File 195 - test_matplotlib.py:

```
1: (0)              import os
2: (0)              import subprocess
3: (0)              import sys
4: (0)              import pytest
5: (0)              import matplotlib
6: (0)              from matplotlib.testing import subprocess_run_for_testing
7: (0)              @pytest.mark.parametrize('version_str, version_tuple', [
8: (4)                  ('3.5.0', (3, 5, 0, 'final', 0)),
```

```
 9: (4)                 ('3.5.0rc2', (3, 5, 0, 'candidate', 2)),
10: (4)                 ('3.5.0.dev820+g6768ef8c4c', (3, 5, 0, 'alpha', 820)),
11: (4)                 ('3.5.0.post820+g6768ef8c4c', (3, 5, 1, 'alpha', 820)),
12: (0)             ])
13: (0)             def test_parse_to_version_info(version_str, version_tuple):
14: (4)                 assert matplotlib._parse_to_version_info(version_str) == version_tuple
15: (0)             @pytest.mark.skipif(sys.platform == "win32",
16: (20)                                reason="chmod() doesn't work as is on Windows")
17: (0)             @pytest.mark.skipif(sys.platform != "win32" and os.geteuid() == 0,
18: (20)                                reason="chmod() doesn't work as root")
19: (0)             def test_tmpconfigdir_warning(tmp_path):
20: (4)                 """Test that a warning is emitted if a temporary configdir must be
used."""
21: (4)                 mode = os.stat(tmp_path).st_mode
22: (4)                 try:
23: (8)                     os.chmod(tmp_path, 0)
24: (8)                     proc = subprocess_run_for_testing(
25: (12)                        [sys.executable, "-c", "import matplotlib"],
26: (12)                        env={**os.environ, "MPLCONFIGDIR": str(tmp_path)},
27: (12)                        stderr=subprocess.PIPE, text=True, check=True)
28: (8)                     assert "set the MPLCONFIGDIR" in proc.stderr
29: (4)                 finally:
30: (8)                     os.chmod(tmp_path, mode)
31: (0)             def test_importable_with_no_home(tmp_path):
32: (4)                 subprocess_run_for_testing(
33: (8)                     [sys.executable, "-c",
34: (9)                      "import pathlib; pathlib.Path.home = lambda *args: 1/0; "
35: (9)                      "import matplotlib.pyplot"],
36: (8)                     env={**os.environ, "MPLCONFIGDIR": str(tmp_path)}, check=True)
37: (0)             def test_use_doc_standard_backends():
38: (4)                 """
39: (4)                 Test that the standard backends mentioned in the docstring of
40: (4)                 matplotlib.use() are the same as in matplotlib.rcsetup.
41: (4)                 """
42: (4)                 def parse(key):
43: (8)                     backends = []
44: (8)                     for line in matplotlib.use.__doc__.split(key)[1].split('\n'):
45: (12)                        if not line.strip():
46: (16)                            break
47: (12)                        backends += [e.strip().lower() for e in line.split(',') if e]
48: (8)                     return backends
49: (4)                 from matplotlib.backends import BackendFilter, backend_registry
50: (4)                 assert (set(parse('- interactive backends:\n')) ==
51: (12)                        set(backend_registry.list_builtin(BackendFilter.INTERACTIVE)))
52: (4)                 assert (set(parse('- non-interactive backends:\n')) ==
53: (12)                        set(backend_registry.list_builtin(BackendFilter.NON_INTERACTIVE)))
54: (0)             def test_importable_with__OO():
55: (4)                 """
56: (4)                 When using -OO or export PYTHONOPTIMIZE=2, docstrings are discarded,
57: (4)                 this simple test may prevent something like issue #17970.
58: (4)                 """
59: (4)                 program = (
60: (8)                     "import matplotlib as mpl; "
61: (8)                     "import matplotlib.pyplot as plt; "
62: (8)                     "import matplotlib.cbook as cbook; "
63: (8)                     "import matplotlib.patches as mpatches"
64: (4)                 )
65: (4)                 subprocess_run_for_testing(
66: (8)                     [sys.executable, "-OO", "-c", program],
67: (8)                     env={**os.environ, "MPLBACKEND": ""}, check=True
68: (8)                     )
```

File 196 - test_mlab.py:

```
 1: (0)             from numpy.testing import (assert_allclose, assert_almost_equal,
 2: (27)                                       assert_array_equal, assert_array_almost_equal_nulp)
 3: (0)             import numpy as np
```

```
4: (0)              import pytest
5: (0)              from matplotlib import mlab
6: (0)              def test_window():
7: (4)                  np.random.seed(0)
8: (4)                  n = 1000
9: (4)                  rand = np.random.standard_normal(n) + 100
10: (4)                 ones = np.ones(n)
11: (4)                 assert_array_equal(mlab.window_none(ones), ones)
12: (4)                 assert_array_equal(mlab.window_none(rand), rand)
13: (4)                 assert_array_equal(np.hanning(len(rand)) * rand,
mlab.window_hanning(rand))
14: (4)                 assert_array_equal(np.hanning(len(ones)), mlab.window_hanning(ones))
15: (0)             class TestDetrend:
16: (4)                 def setup_method(self):
17: (8)                     np.random.seed(0)
18: (8)                     n = 1000
19: (8)                     x = np.linspace(0., 100, n)
20: (8)                     self.sig_zeros = np.zeros(n)
21: (8)                     self.sig_off = self.sig_zeros + 100.
22: (8)                     self.sig_slope = np.linspace(-10., 90., n)
23: (8)                     self.sig_slope_mean = x - x.mean()
24: (8)                     self.sig_base = (
25: (12)                        np.random.standard_normal(n) + np.sin(x*2*np.pi/(n/100)))
26: (8)                     self.sig_base -= self.sig_base.mean()
27: (4)                 def allclose(self, *args):
28: (8)                     assert_allclose(*args, atol=1e-8)
29: (4)                 def test_detrend_none(self):
30: (8)                     assert mlab.detrend_none(0.) == 0.
31: (8)                     assert mlab.detrend_none(0., axis=1) == 0.
32: (8)                     assert mlab.detrend(0., key="none") == 0.
33: (8)                     assert mlab.detrend(0., key=mlab.detrend_none) == 0.
34: (8)                     for sig in [
35: (16)                            5.5, self.sig_off, self.sig_slope, self.sig_base,
36: (16)                            (self.sig_base + self.sig_slope + self.sig_off).tolist(),
37: (16)                            np.vstack([self.sig_base,  # 2D case.
38: (27)                                       self.sig_base + self.sig_off,
39: (27)                                       self.sig_base + self.sig_slope,
40: (27)                                       self.sig_base + self.sig_off + self.sig_slope]),
41: (16)                            np.vstack([self.sig_base,  # 2D transposed case.
42: (27)                                       self.sig_base + self.sig_off,
43: (27)                                       self.sig_base + self.sig_slope,
44: (27)                                       self.sig_base + self.sig_off + self.sig_slope]).T,
45: (8)                     ]:
46: (12)                        if isinstance(sig, np.ndarray):
47: (16)                            assert_array_equal(mlab.detrend_none(sig), sig)
48: (12)                        else:
49: (16)                            assert mlab.detrend_none(sig) == sig
50: (4)                 def test_detrend_mean(self):
51: (8)                     for sig in [0., 5.5]:  # 0D.
52: (12)                        assert mlab.detrend_mean(sig) == 0.
53: (12)                        assert mlab.detrend(sig, key="mean") == 0.
54: (12)                        assert mlab.detrend(sig, key=mlab.detrend_mean) == 0.
55: (8)                     self.allclose(mlab.detrend_mean(self.sig_zeros), self.sig_zeros)
56: (8)                     self.allclose(mlab.detrend_mean(self.sig_base), self.sig_base)
57: (8)                     self.allclose(mlab.detrend_mean(self.sig_base + self.sig_off),
58: (22)                                  self.sig_base)
59: (8)                     self.allclose(mlab.detrend_mean(self.sig_base + self.sig_slope),
60: (22)                                  self.sig_base + self.sig_slope_mean)
61: (8)                     self.allclose(
62: (12)                        mlab.detrend_mean(self.sig_base + self.sig_slope + self.sig_off),
63: (12)                        self.sig_base + self.sig_slope_mean)
64: (4)                 def test_detrend_mean_1d_base_slope_off_list_andor_axis0(self):
65: (8)                     input = self.sig_base + self.sig_slope + self.sig_off
66: (8)                     target = self.sig_base + self.sig_slope_mean
67: (8)                     self.allclose(mlab.detrend_mean(input, axis=0), target)
68: (8)                     self.allclose(mlab.detrend_mean(input.tolist()), target)
69: (8)                     self.allclose(mlab.detrend_mean(input.tolist(), axis=0), target)
70: (4)                 def test_detrend_mean_2d(self):
71: (8)                     input = np.vstack([self.sig_off,
```

```
 72: (27)                                    self.sig_base + self.sig_off])
 73: (8)                     target = np.vstack([self.sig_zeros,
 74: (28)                                     self.sig_base])
 75: (8)                     self.allclose(mlab.detrend_mean(input), target)
 76: (8)                     self.allclose(mlab.detrend_mean(input, axis=None), target)
 77: (8)                     self.allclose(mlab.detrend_mean(input.T, axis=None).T, target)
 78: (8)                     self.allclose(mlab.detrend(input), target)
 79: (8)                     self.allclose(mlab.detrend(input, axis=None), target)
 80: (8)                     self.allclose(
 81: (12)                        mlab.detrend(input.T, key="constant", axis=None), target.T)
 82: (8)                     input = np.vstack([self.sig_base,
 83: (27)                                    self.sig_base + self.sig_off,
 84: (27)                                    self.sig_base + self.sig_slope,
 85: (27)                                    self.sig_base + self.sig_off + self.sig_slope])
 86: (8)                     target = np.vstack([self.sig_base,
 87: (28)                                     self.sig_base,
 88: (28)                                     self.sig_base + self.sig_slope_mean,
 89: (28)                                     self.sig_base + self.sig_slope_mean])
 90: (8)                     self.allclose(mlab.detrend_mean(input.T, axis=0), target.T)
 91: (8)                     self.allclose(mlab.detrend_mean(input, axis=1), target)
 92: (8)                     self.allclose(mlab.detrend_mean(input, axis=-1), target)
 93: (8)                     self.allclose(mlab.detrend(input, key="default", axis=1), target)
 94: (8)                     self.allclose(mlab.detrend(input.T, key="mean", axis=0), target.T)
 95: (8)                     self.allclose(
 96: (12)                        mlab.detrend(input.T, key=mlab.detrend_mean, axis=0), target.T)
 97: (4)                 def test_detrend_ValueError(self):
 98: (8)                     for signal, kwargs in [
 99: (16)                            (self.sig_slope[np.newaxis], {"key": "spam"}),
100: (16)                            (self.sig_slope[np.newaxis], {"key": 5}),
101: (16)                            (5.5, {"axis": 0}),
102: (16)                            (self.sig_slope, {"axis": 1}),
103: (16)                            (self.sig_slope[np.newaxis], {"axis": 2}),
104: (8)                     ]:
105: (12)                        with pytest.raises(ValueError):
106: (16)                            mlab.detrend(signal, **kwargs)
107: (4)                 def test_detrend_mean_ValueError(self):
108: (8)                     for signal, kwargs in [
109: (16)                            (5.5, {"axis": 0}),
110: (16)                            (self.sig_slope, {"axis": 1}),
111: (16)                            (self.sig_slope[np.newaxis], {"axis": 2}),
112: (8)                     ]:
113: (12)                        with pytest.raises(ValueError):
114: (16)                            mlab.detrend_mean(signal, **kwargs)
115: (4)                 def test_detrend_linear(self):
116: (8)                     assert mlab.detrend_linear(0.) == 0.
117: (8)                     assert mlab.detrend_linear(5.5) == 0.
118: (8)                     assert mlab.detrend(5.5, key="linear") == 0.
119: (8)                     assert mlab.detrend(5.5, key=mlab.detrend_linear) == 0.
120: (8)                     for sig in [  # 1D.
121: (16)                            self.sig_off,
122: (16)                            self.sig_slope,
123: (16)                            self.sig_slope + self.sig_off,
124: (8)                     ]:
125: (12)                        self.allclose(mlab.detrend_linear(sig), self.sig_zeros)
126: (4)                 def test_detrend_str_linear_1d(self):
127: (8)                     input = self.sig_slope + self.sig_off
128: (8)                     target = self.sig_zeros
129: (8)                     self.allclose(mlab.detrend(input, key="linear"), target)
130: (8)                     self.allclose(mlab.detrend(input, key=mlab.detrend_linear), target)
131: (8)                     self.allclose(mlab.detrend_linear(input.tolist()), target)
132: (4)                 def test_detrend_linear_2d(self):
133: (8)                     input = np.vstack([self.sig_off,
134: (27)                                    self.sig_slope,
135: (27)                                    self.sig_slope + self.sig_off])
136: (8)                     target = np.vstack([self.sig_zeros,
137: (28)                                     self.sig_zeros,
138: (28)                                     self.sig_zeros])
139: (8)                     self.allclose(
140: (12)                        mlab.detrend(input.T, key="linear", axis=0), target.T)
```

```
(8)                     self.allclose(
(12)                        mlab.detrend(input.T, key=mlab.detrend_linear, axis=0), target.T)
(8)                     self.allclose(
(12)                        mlab.detrend(input, key="linear", axis=1), target)
(8)                     self.allclose(
(12)                        mlab.detrend(input, key=mlab.detrend_linear, axis=1), target)
(8)                     with pytest.raises(ValueError):
(12)                        mlab.detrend_linear(self.sig_slope[np.newaxis])
(0)             @pytest.mark.parametrize('iscomplex', [False, True],
(25)                                     ids=['real', 'complex'], scope='class')
(0)             @pytest.mark.parametrize('sides', ['onesided', 'twosided', 'default'],
(25)                                     scope='class')
(0)             @pytest.mark.parametrize(
(4)                 'fstims,len_x,NFFT_density,nover_density,pad_to_density,pad_to_spectrum',
(4)                 [
(8)                     ([], None, -1, -1, -1, -1),
(8)                     ([4], None, -1, -1, -1, -1),
(8)                     ([4, 5, 10], None, -1, -1, -1, -1),
(8)                     ([], None, None, -1, -1, None),
(8)                     ([], None, -1, -1, None, None),
(8)                     ([], None, None, -1, None, None),
(8)                     ([], 1024, 512, -1, -1, 128),
(8)                     ([], 256, -1, -1, 33, 257),
(8)                     ([], 255, 33, -1, -1, None),
(8)                     ([], 256, 128, -1, 256, 256),
(8)                     ([], None, -1, 32, -1, -1),
(4)                 ],
(4)                 ids=[
(8)                     'nosig',
(8)                     'Fs4',
(8)                     'FsAll',
(8)                     'nosig_noNFFT',
(8)                     'nosig_nopad_to',
(8)                     'nosig_noNFFT_no_pad_to',
(8)                     'nosig_trim',
(8)                     'nosig_odd',
(8)                     'nosig_oddlen',
(8)                     'nosig_stretch',
(8)                     'nosig_overlap',
(4)                 ],
(4)                 scope='class')
(0)             class TestSpectral:
(4)                 @pytest.fixture(scope='class', autouse=True)
(4)                 def stim(self, request, fstims, iscomplex, sides, len_x, NFFT_density,
(13)                          nover_density, pad_to_density, pad_to_spectrum):
(8)                     Fs = 100.
(8)                     x = np.arange(0, 10, 1 / Fs)
(8)                     if len_x is not None:
(12)                        x = x[:len_x]
(8)                     fstims = [Fs / fstim for fstim in fstims]
(8)                     if NFFT_density is None:
(12)                        NFFT_density_real = 256
(8)                     elif NFFT_density < 0:
(12)                        NFFT_density_real = NFFT_density = 100
(8)                     else:
(12)                        NFFT_density_real = NFFT_density
(8)                     if nover_density is None:
(12)                        nover_density_real = 0
(8)                     elif nover_density < 0:
(12)                        nover_density_real = nover_density = NFFT_density_real // 2
(8)                     else:
(12)                        nover_density_real = nover_density
(8)                     if pad_to_density is None:
(12)                        pad_to_density_real = NFFT_density_real
(8)                     elif pad_to_density < 0:
(12)                        pad_to_density = int(2**np.ceil(np.log2(NFFT_density_real)))
(12)                        pad_to_density_real = pad_to_density
(8)                     else:
(12)                        pad_to_density_real = pad_to_density
```

```
210: (8)                    if pad_to_spectrum is None:
211: (12)                       pad_to_spectrum_real = len(x)
212: (8)                    elif pad_to_spectrum < 0:
213: (12)                       pad_to_spectrum_real = pad_to_spectrum = len(x)
214: (8)                    else:
215: (12)                       pad_to_spectrum_real = pad_to_spectrum
216: (8)                    if pad_to_spectrum is None:
217: (12)                       NFFT_spectrum_real = NFFT_spectrum = pad_to_spectrum_real
218: (8)                    else:
219: (12)                       NFFT_spectrum_real = NFFT_spectrum = len(x)
220: (8)                    nover_spectrum = 0
221: (8)                    NFFT_specgram = NFFT_density
222: (8)                    nover_specgram = nover_density
223: (8)                    pad_to_specgram = pad_to_density
224: (8)                    NFFT_specgram_real = NFFT_density_real
225: (8)                    nover_specgram_real = nover_density_real
226: (8)                    if sides == 'onesided' or (sides == 'default' and not iscomplex):
227: (12)                       if pad_to_density_real % 2:
228: (16)                           freqs_density = np.linspace(0, Fs / 2,
229: (44)                                                       num=pad_to_density_real,
230: (44)                                                       endpoint=False)[::2]
231: (12)                       else:
232: (16)                           freqs_density = np.linspace(0, Fs / 2,
233: (44)                                                       num=pad_to_density_real // 2 + 1)
234: (12)                       if pad_to_spectrum_real % 2:
235: (16)                           freqs_spectrum = np.linspace(0, Fs / 2,
236: (45)                                                        num=pad_to_spectrum_real,
237: (45)                                                        endpoint=False)[::2]
238: (12)                       else:
239: (16)                           freqs_spectrum = np.linspace(0, Fs / 2,
240: (45)                                                        num=pad_to_spectrum_real // 2 +
1)
241: (8)                    else:
242: (12)                       if pad_to_density_real % 2:
243: (16)                           freqs_density = np.linspace(-Fs / 2, Fs / 2,
244: (44)                                                       num=2 * pad_to_density_real,
245: (44)                                                       endpoint=False)[1::2]
246: (12)                       else:
247: (16)                           freqs_density = np.linspace(-Fs / 2, Fs / 2,
248: (44)                                                       num=pad_to_density_real,
249: (44)                                                       endpoint=False)
250: (12)                       if pad_to_spectrum_real % 2:
251: (16)                           freqs_spectrum = np.linspace(-Fs / 2, Fs / 2,
252: (45)                                                        num=2 * pad_to_spectrum_real,
253: (45)                                                        endpoint=False)[1::2]
254: (12)                       else:
255: (16)                           freqs_spectrum = np.linspace(-Fs / 2, Fs / 2,
256: (45)                                                        num=pad_to_spectrum_real,
257: (45)                                                        endpoint=False)
258: (8)                    freqs_specgram = freqs_density
259: (8)                    t_start = NFFT_specgram_real // 2
260: (8)                    t_stop = len(x) - NFFT_specgram_real // 2 + 1
261: (8)                    t_step = NFFT_specgram_real - nover_specgram_real
262: (8)                    t_specgram = x[t_start:t_stop:t_step]
263: (8)                    if NFFT_specgram_real % 2:
264: (12)                       t_specgram += 1 / Fs / 2
265: (8)                    if len(t_specgram) == 0:
266: (12)                       t_specgram = np.array([NFFT_specgram_real / (2 * Fs)])
267: (8)                    t_spectrum = np.array([NFFT_spectrum_real / (2 * Fs)])
268: (8)                    t_density = t_specgram
269: (8)                    y = np.zeros_like(x)
270: (8)                    for i, fstim in enumerate(fstims):
271: (12)                       y += np.sin(fstim * x * np.pi * 2) * 10**i
272: (8)                    if iscomplex:
273: (12)                       y = y.astype('complex')
274: (8)                    cls = request.cls
275: (8)                    cls.Fs = Fs
276: (8)                    cls.sides = sides
277: (8)                    cls.fstims = fstims
```

```
(8)                 cls.NFFT_density = NFFT_density
(8)                 cls.nover_density = nover_density
(8)                 cls.pad_to_density = pad_to_density
(8)                 cls.NFFT_spectrum = NFFT_spectrum
(8)                 cls.nover_spectrum = nover_spectrum
(8)                 cls.pad_to_spectrum = pad_to_spectrum
(8)                 cls.NFFT_specgram = NFFT_specgram
(8)                 cls.nover_specgram = nover_specgram
(8)                 cls.pad_to_specgram = pad_to_specgram
(8)                 cls.t_specgram = t_specgram
(8)                 cls.t_density = t_density
(8)                 cls.t_spectrum = t_spectrum
(8)                 cls.y = y
(8)                 cls.freqs_density = freqs_density
(8)                 cls.freqs_spectrum = freqs_spectrum
(8)                 cls.freqs_specgram = freqs_specgram
(8)                 cls.NFFT_density_real = NFFT_density_real
(4)             def check_freqs(self, vals, targfreqs, resfreqs, fstims):
(8)                 assert resfreqs.argmin() == 0
(8)                 assert resfreqs.argmax() == len(resfreqs)-1
(8)                 assert_allclose(resfreqs, targfreqs, atol=1e-06)
(8)                 for fstim in fstims:
(12)                    i = np.abs(resfreqs - fstim).argmin()
(12)                    assert vals[i] > vals[i+2]
(12)                    assert vals[i] > vals[i-2]
(4)             def check_maxfreq(self, spec, fsp, fstims):
(8)                 if len(fstims) == 0:
(12)                    return
(8)                 if fsp.min() < 0:
(12)                    fspa = np.abs(fsp)
(12)                    zeroind = fspa.argmin()
(12)                    self.check_maxfreq(spec[:zeroind], fspa[:zeroind], fstims)
(12)                    self.check_maxfreq(spec[zeroind:], fspa[zeroind:], fstims)
(12)                    return
(8)                 fstimst = fstims[:]
(8)                 spect = spec.copy()
(8)                 while fstimst:
(12)                    maxind = spect.argmax()
(12)                    maxfreq = fsp[maxind]
(12)                    assert_almost_equal(maxfreq, fstimst[-1])
(12)                    del fstimst[-1]
(12)                    spect[maxind-5:maxind+5] = 0
(4)             def test_spectral_helper_raises(self):
(8)                 for kwargs in [  # Various error conditions:
(12)                    {"y": self.y+1, "mode": "complex"},  # Modes requiring ``x is y``.
(12)                    {"y": self.y+1, "mode": "magnitude"},
(12)                    {"y": self.y+1, "mode": "angle"},
(12)                    {"y": self.y+1, "mode": "phase"},
(12)                    {"mode": "spam"},  # Bad mode.
(12)                    {"y": self.y, "sides": "eggs"},  # Bad sides.
(12)                    {"y": self.y, "NFFT": 10, "noverlap": 20},  # noverlap > NFFT.
(12)                    {"NFFT": 10, "noverlap": 10},  # noverlap == NFFT.
(12)                    {"y": self.y, "NFFT": 10,
(13)                     "window": np.ones(9)},  # len(win) != NFFT.
(8)                 ]:
(12)                    with pytest.raises(ValueError):
(16)                        mlab._spectral_helper(x=self.y, **kwargs)
(4)             @pytest.mark.parametrize('mode', ['default', 'psd'])
(4)             def test_single_spectrum_helper_unsupported_modes(self, mode):
(8)                 with pytest.raises(ValueError):
(12)                    mlab._single_spectrum_helper(x=self.y, mode=mode)
(4)             @pytest.mark.parametrize("mode, case", [
(8)                 ("psd", "density"),
(8)                 ("magnitude", "specgram"),
(8)                 ("magnitude", "spectrum"),
(4)             ])
(4)             def test_spectral_helper_psd(self, mode, case):
(8)                 freqs = getattr(self, f"freqs_{case}")
(8)                 spec, fsp, t = mlab._spectral_helper(
```

```
347: (12)                        x=self.y, y=self.y,
348: (12)                        NFFT=getattr(self, f"NFFT_{case}"),
349: (12)                        Fs=self.Fs,
350: (12)                        noverlap=getattr(self, f"nover_{case}"),
351: (12)                        pad_to=getattr(self, f"pad_to_{case}"),
352: (12)                        sides=self.sides,
353: (12)                        mode=mode)
354: (8)                    assert_allclose(fsp, freqs, atol=1e-06)
355: (8)                    assert_allclose(t, getattr(self, f"t_{case}"), atol=1e-06)
356: (8)                    assert spec.shape[0] == freqs.shape[0]
357: (8)                    assert spec.shape[1] == getattr(self, f"t_{case}").shape[0]
358: (4)                def test_csd(self):
359: (8)                    freqs = self.freqs_density
360: (8)                    spec, fsp = mlab.csd(x=self.y, y=self.y+1,
361: (29)                                         NFFT=self.NFFT_density,
362: (29)                                         Fs=self.Fs,
363: (29)                                         noverlap=self.nover_density,
364: (29)                                         pad_to=self.pad_to_density,
365: (29)                                         sides=self.sides)
366: (8)                    assert_allclose(fsp, freqs, atol=1e-06)
367: (8)                    assert spec.shape == freqs.shape
368: (4)                def test_csd_padding(self):
369: (8)                    """Test zero padding of csd()."""
370: (8)                    if self.NFFT_density is None:  # for derived classes
371: (12)                       return
372: (8)                    sargs = dict(x=self.y, y=self.y+1, Fs=self.Fs,
window=mlab.window_none,
373: (21)                                 sides=self.sides)
374: (8)                    spec0, _ = mlab.csd(NFFT=self.NFFT_density, **sargs)
375: (8)                    spec1, _ = mlab.csd(NFFT=self.NFFT_density*2, **sargs)
376: (8)                    assert_almost_equal(np.sum(np.conjugate(spec0)*spec0).real,
377: (28)                                       np.sum(np.conjugate(spec1/2)*spec1/2).real)
378: (4)                def test_psd(self):
379: (8)                    freqs = self.freqs_density
380: (8)                    spec, fsp = mlab.psd(x=self.y,
381: (29)                                         NFFT=self.NFFT_density,
382: (29)                                         Fs=self.Fs,
383: (29)                                         noverlap=self.nover_density,
384: (29)                                         pad_to=self.pad_to_density,
385: (29)                                         sides=self.sides)
386: (8)                    assert spec.shape == freqs.shape
387: (8)                    self.check_freqs(spec, freqs, fsp, self.fstims)
388: (4)                @pytest.mark.parametrize(
389: (8)                    'make_data, detrend',
390: (8)                    [(np.zeros, mlab.detrend_mean), (np.zeros, 'mean'),
391: (9)                     (np.arange, mlab.detrend_linear), (np.arange, 'linear')])
392: (4)                def test_psd_detrend(self, make_data, detrend):
393: (8)                    if self.NFFT_density is None:
394: (12)                       return
395: (8)                    ydata = make_data(self.NFFT_density)
396: (8)                    ydata1 = ydata+5
397: (8)                    ydata2 = ydata+3.3
398: (8)                    ydata = np.vstack([ydata1, ydata2])
399: (8)                    ydata = np.tile(ydata, (20, 1))
400: (8)                    ydatab = ydata.T.flatten()
401: (8)                    ydata = ydata.flatten()
402: (8)                    ycontrol = np.zeros_like(ydata)
403: (8)                    spec_g, fsp_g = mlab.psd(x=ydata,
404: (33)                                            NFFT=self.NFFT_density,
405: (33)                                            Fs=self.Fs,
406: (33)                                            noverlap=0,
407: (33)                                            sides=self.sides,
408: (33)                                            detrend=detrend)
409: (8)                    spec_b, fsp_b = mlab.psd(x=ydatab,
410: (33)                                            NFFT=self.NFFT_density,
411: (33)                                            Fs=self.Fs,
412: (33)                                            noverlap=0,
413: (33)                                            sides=self.sides,
414: (33)                                            detrend=detrend)
```

```
(8)                 spec_c, fsp_c = mlab.psd(x=ycontrol,
(33)                                         NFFT=self.NFFT_density,
(33)                                         Fs=self.Fs,
(33)                                         noverlap=0,
(33)                                         sides=self.sides)
(8)                 assert_array_equal(fsp_g, fsp_c)
(8)                 assert_array_equal(fsp_b, fsp_c)
(8)                 assert_allclose(spec_g, spec_c, atol=1e-08)
(8)                 with pytest.raises(AssertionError):
(12)                    assert_allclose(spec_b, spec_c, atol=1e-08)
(4)             def test_psd_window_hanning(self):
(8)                 if self.NFFT_density is None:
(12)                    return
(8)                 ydata = np.arange(self.NFFT_density)
(8)                 ydata1 = ydata+5
(8)                 ydata2 = ydata+3.3
(8)                 windowVals = mlab.window_hanning(np.ones_like(ydata1))
(8)                 ycontrol1 = ydata1 * windowVals
(8)                 ycontrol2 = mlab.window_hanning(ydata2)
(8)                 ydata = np.vstack([ydata1, ydata2])
(8)                 ycontrol = np.vstack([ycontrol1, ycontrol2])
(8)                 ydata = np.tile(ydata, (20, 1))
(8)                 ycontrol = np.tile(ycontrol, (20, 1))
(8)                 ydatab = ydata.T.flatten()
(8)                 ydataf = ydata.flatten()
(8)                 ycontrol = ycontrol.flatten()
(8)                 spec_g, fsp_g = mlab.psd(x=ydataf,
(33)                                         NFFT=self.NFFT_density,
(33)                                         Fs=self.Fs,
(33)                                         noverlap=0,
(33)                                         sides=self.sides,
(33)                                         window=mlab.window_hanning)
(8)                 spec_b, fsp_b = mlab.psd(x=ydatab,
(33)                                         NFFT=self.NFFT_density,
(33)                                         Fs=self.Fs,
(33)                                         noverlap=0,
(33)                                         sides=self.sides,
(33)                                         window=mlab.window_hanning)
(8)                 spec_c, fsp_c = mlab.psd(x=ycontrol,
(33)                                         NFFT=self.NFFT_density,
(33)                                         Fs=self.Fs,
(33)                                         noverlap=0,
(33)                                         sides=self.sides,
(33)                                         window=mlab.window_none)
(8)                 spec_c *= len(ycontrol1)/(windowVals**2).sum()
(8)                 assert_array_equal(fsp_g, fsp_c)
(8)                 assert_array_equal(fsp_b, fsp_c)
(8)                 assert_allclose(spec_g, spec_c, atol=1e-08)
(8)                 with pytest.raises(AssertionError):
(12)                    assert_allclose(spec_b, spec_c, atol=1e-08)
(4)             def test_psd_window_hanning_detrend_linear(self):
(8)                 if self.NFFT_density is None:
(12)                    return
(8)                 ydata = np.arange(self.NFFT_density)
(8)                 ycontrol = np.zeros(self.NFFT_density)
(8)                 ydata1 = ydata+5
(8)                 ydata2 = ydata+3.3
(8)                 ycontrol1 = ycontrol
(8)                 ycontrol2 = ycontrol
(8)                 windowVals = mlab.window_hanning(np.ones_like(ycontrol1))
(8)                 ycontrol1 = ycontrol1 * windowVals
(8)                 ycontrol2 = mlab.window_hanning(ycontrol2)
(8)                 ydata = np.vstack([ydata1, ydata2])
(8)                 ycontrol = np.vstack([ycontrol1, ycontrol2])
(8)                 ydata = np.tile(ydata, (20, 1))
(8)                 ycontrol = np.tile(ycontrol, (20, 1))
(8)                 ydatab = ydata.T.flatten()
(8)                 ydataf = ydata.flatten()
(8)                 ycontrol = ycontrol.flatten()
```

```
(8)                     spec_g, fsp_g = mlab.psd(x=ydataf,
(33)                                             NFFT=self.NFFT_density,
(33)                                             Fs=self.Fs,
(33)                                             noverlap=0,
(33)                                             sides=self.sides,
(33)                                             detrend=mlab.detrend_linear,
(33)                                             window=mlab.window_hanning)
(8)                     spec_b, fsp_b = mlab.psd(x=ydatab,
(33)                                             NFFT=self.NFFT_density,
(33)                                             Fs=self.Fs,
(33)                                             noverlap=0,
(33)                                             sides=self.sides,
(33)                                             detrend=mlab.detrend_linear,
(33)                                             window=mlab.window_hanning)
(8)                     spec_c, fsp_c = mlab.psd(x=ycontrol,
(33)                                             NFFT=self.NFFT_density,
(33)                                             Fs=self.Fs,
(33)                                             noverlap=0,
(33)                                             sides=self.sides,
(33)                                             window=mlab.window_none)
(8)                     spec_c *= len(ycontrol1)/(windowVals**2).sum()
(8)                     assert_array_equal(fsp_g, fsp_c)
(8)                     assert_array_equal(fsp_b, fsp_c)
(8)                     assert_allclose(spec_g, spec_c, atol=1e-08)
(8)                     with pytest.raises(AssertionError):
(12)                        assert_allclose(spec_b, spec_c, atol=1e-08)
(4)                 def test_psd_window_flattop(self):
(8)                     a = [0.21557895, 0.41663158, 0.277263158, 0.083578947, 0.006947368]
(8)                     fac = np.linspace(-np.pi, np.pi, self.NFFT_density_real)
(8)                     win = np.zeros(self.NFFT_density_real)
(8)                     for k in range(len(a)):
(12)                        win += a[k] * np.cos(k * fac)
(8)                     spec, fsp = mlab.psd(x=self.y,
(29)                                         NFFT=self.NFFT_density,
(29)                                         Fs=self.Fs,
(29)                                         noverlap=0,
(29)                                         sides=self.sides,
(29)                                         window=win,
(29)                                         scale_by_freq=False)
(8)                     spec_a, fsp_a = mlab.psd(x=self.y,
(33)                                             NFFT=self.NFFT_density,
(33)                                             Fs=self.Fs,
(33)                                             noverlap=0,
(33)                                             sides=self.sides,
(33)                                             window=win)
(8)                     assert_allclose(spec*win.sum()**2,
(24)                                    spec_a*self.Fs*(win**2).sum(),
(24)                                    atol=1e-08)
(4)                 def test_psd_windowarray(self):
(8)                     freqs = self.freqs_density
(8)                     spec, fsp = mlab.psd(x=self.y,
(29)                                         NFFT=self.NFFT_density,
(29)                                         Fs=self.Fs,
(29)                                         noverlap=self.nover_density,
(29)                                         pad_to=self.pad_to_density,
(29)                                         sides=self.sides,
(29)                                         window=np.ones(self.NFFT_density_real))
(8)                     assert_allclose(fsp, freqs, atol=1e-06)
(8)                     assert spec.shape == freqs.shape
(4)                 def test_psd_windowarray_scale_by_freq(self):
(8)                     win = mlab.window_hanning(np.ones(self.NFFT_density_real))
(8)                     spec, fsp = mlab.psd(x=self.y,
(29)                                         NFFT=self.NFFT_density,
(29)                                         Fs=self.Fs,
(29)                                         noverlap=self.nover_density,
(29)                                         pad_to=self.pad_to_density,
(29)                                         sides=self.sides,
(29)                                         window=mlab.window_hanning)
(8)                     spec_s, fsp_s = mlab.psd(x=self.y,
```

```
553: (33)                                           NFFT=self.NFFT_density,
554: (33)                                           Fs=self.Fs,
555: (33)                                           noverlap=self.nover_density,
556: (33)                                           pad_to=self.pad_to_density,
557: (33)                                           sides=self.sides,
558: (33)                                           window=mlab.window_hanning,
559: (33)                                           scale_by_freq=True)
560: (8)                    spec_n, fsp_n = mlab.psd(x=self.y,
561: (33)                                           NFFT=self.NFFT_density,
562: (33)                                           Fs=self.Fs,
563: (33)                                           noverlap=self.nover_density,
564: (33)                                           pad_to=self.pad_to_density,
565: (33)                                           sides=self.sides,
566: (33)                                           window=mlab.window_hanning,
567: (33)                                           scale_by_freq=False)
568: (8)                    assert_array_equal(fsp, fsp_s)
569: (8)                    assert_array_equal(fsp, fsp_n)
570: (8)                    assert_array_equal(spec, spec_s)
571: (8)                    assert_allclose(spec_s*(win**2).sum(),
572: (24)                                   spec_n/self.Fs*win.sum()**2,
573: (24)                                   atol=1e-08)
574: (4)                @pytest.mark.parametrize(
575: (8)                    "kind", ["complex", "magnitude", "angle", "phase"])
576: (4)                def test_spectrum(self, kind):
577: (8)                    freqs = self.freqs_spectrum
578: (8)                    spec, fsp = getattr(mlab, f"{kind}_spectrum")(
579: (12)                       x=self.y,
580: (12)                       Fs=self.Fs, sides=self.sides, pad_to=self.pad_to_spectrum)
581: (8)                    assert_allclose(fsp, freqs, atol=1e-06)
582: (8)                    assert spec.shape == freqs.shape
583: (8)                    if kind == "magnitude":
584: (12)                       self.check_maxfreq(spec, fsp, self.fstims)
585: (12)                       self.check_freqs(spec, freqs, fsp, self.fstims)
586: (4)                @pytest.mark.parametrize(
587: (8)                    'kwargs',
588: (8)                    [{}, {'mode': 'default'}, {'mode': 'psd'}, {'mode': 'magnitude'},
589: (9)                     {'mode': 'complex'}, {'mode': 'angle'}, {'mode': 'phase'}])
590: (4)                def test_specgram(self, kwargs):
591: (8)                    freqs = self.freqs_specgram
592: (8)                    spec, fsp, t = mlab.specgram(x=self.y,
593: (37)                                                NFFT=self.NFFT_specgram,
594: (37)                                                Fs=self.Fs,
595: (37)                                                noverlap=self.nover_specgram,
596: (37)                                                pad_to=self.pad_to_specgram,
597: (37)                                                sides=self.sides,
598: (37)                                                **kwargs)
599: (8)                    if kwargs.get('mode') == 'complex':
600: (12)                       spec = np.abs(spec)
601: (8)                    specm = np.mean(spec, axis=1)
602: (8)                    assert_allclose(fsp, freqs, atol=1e-06)
603: (8)                    assert_allclose(t, self.t_specgram, atol=1e-06)
604: (8)                    assert spec.shape[0] == freqs.shape[0]
605: (8)                    assert spec.shape[1] == self.t_specgram.shape[0]
606: (8)                    if kwargs.get('mode') not in ['complex', 'angle', 'phase']:
607: (12)                       if np.abs(spec.max()) != 0:
608: (16)                           assert_allclose(
609: (20)                               np.diff(spec, axis=1).max() / np.abs(spec.max()), 0,
610: (20)                               atol=1e-02)
611: (8)                    if kwargs.get('mode') not in ['angle', 'phase']:
612: (12)                       self.check_freqs(specm, freqs, fsp, self.fstims)
613: (4)                def test_specgram_warn_only1seg(self):
614: (8)                    """Warning should be raised if len(x) <= NFFT."""
615: (8)                    with pytest.warns(UserWarning, match="Only one segment is
calculated"):
616: (12)                       mlab.specgram(x=self.y, NFFT=len(self.y), Fs=self.Fs)
617: (4)                def test_psd_csd_equal(self):
618: (8)                    Pxx, freqsxx = mlab.psd(x=self.y,
619: (32)                                           NFFT=self.NFFT_density,
620: (32)                                           Fs=self.Fs,
```

```
621: (32)                                        noverlap=self.nover_density,
622: (32)                                        pad_to=self.pad_to_density,
623: (32)                                        sides=self.sides)
624: (8)                    Pxy, freqsxy = mlab.csd(x=self.y, y=self.y,
625: (32)                                        NFFT=self.NFFT_density,
626: (32)                                        Fs=self.Fs,
627: (32)                                        noverlap=self.nover_density,
628: (32)                                        pad_to=self.pad_to_density,
629: (32)                                        sides=self.sides)
630: (8)                    assert_array_almost_equal_nulp(Pxx, Pxy)
631: (8)                    assert_array_equal(freqsxx, freqsxy)
632: (4)                @pytest.mark.parametrize("mode", ["default", "psd"])
633: (4)                def test_specgram_auto_default_psd_equal(self, mode):
634: (8)                    """
635: (8)                    Test that mlab.specgram without mode and with mode 'default' and 'psd'
636: (8)                    are all the same.
637: (8)                    """
638: (8)                    speca, freqspeca, ta = mlab.specgram(x=self.y,
639: (45)                                                NFFT=self.NFFT_specgram,
640: (45)                                                Fs=self.Fs,
641: (45)                                                noverlap=self.nover_specgram,
642: (45)                                                pad_to=self.pad_to_specgram,
643: (45)                                                sides=self.sides)
644: (8)                    specb, freqspecb, tb = mlab.specgram(x=self.y,
645: (45)                                                NFFT=self.NFFT_specgram,
646: (45)                                                Fs=self.Fs,
647: (45)                                                noverlap=self.nover_specgram,
648: (45)                                                pad_to=self.pad_to_specgram,
649: (45)                                                sides=self.sides,
650: (45)                                                mode=mode)
651: (8)                    assert_array_equal(speca, specb)
652: (8)                    assert_array_equal(freqspeca, freqspecb)
653: (8)                    assert_array_equal(ta, tb)
654: (4)                @pytest.mark.parametrize(
655: (8)                    "mode, conv", [
656: (12)                       ("magnitude", np.abs),
657: (12)                       ("angle", np.angle),
658: (12)                       ("phase", lambda x: np.unwrap(np.angle(x), axis=0))
659: (8)                    ])
660: (4)                def test_specgram_complex_equivalent(self, mode, conv):
661: (8)                    specc, freqspecc, tc = mlab.specgram(x=self.y,
662: (45)                                                NFFT=self.NFFT_specgram,
663: (45)                                                Fs=self.Fs,
664: (45)                                                noverlap=self.nover_specgram,
665: (45)                                                pad_to=self.pad_to_specgram,
666: (45)                                                sides=self.sides,
667: (45)                                                mode='complex')
668: (8)                    specm, freqspecm, tm = mlab.specgram(x=self.y,
669: (45)                                                NFFT=self.NFFT_specgram,
670: (45)                                                Fs=self.Fs,
671: (45)                                                noverlap=self.nover_specgram,
672: (45)                                                pad_to=self.pad_to_specgram,
673: (45)                                                sides=self.sides,
674: (45)                                                mode=mode)
675: (8)                    assert_array_equal(freqspecc, freqspecm)
676: (8)                    assert_array_equal(tc, tm)
677: (8)                    assert_allclose(conv(specc), specm, atol=1e-06)
678: (4)                def test_psd_windowarray_equal(self):
679: (8)                    win = mlab.window_hanning(np.ones(self.NFFT_density_real))
680: (8)                    speca, fspa = mlab.psd(x=self.y,
681: (31)                                           NFFT=self.NFFT_density,
682: (31)                                           Fs=self.Fs,
683: (31)                                           noverlap=self.nover_density,
684: (31)                                           pad_to=self.pad_to_density,
685: (31)                                           sides=self.sides,
686: (31)                                           window=win)
687: (8)                    specb, fspb = mlab.psd(x=self.y,
688: (31)                                           NFFT=self.NFFT_density,
689: (31)                                           Fs=self.Fs,
```

```
690: (31)                                         noverlap=self.nover_density,
691: (31)                                         pad_to=self.pad_to_density,
692: (31)                                         sides=self.sides)
693: (8)                    assert_array_equal(fspa, fspb)
694: (8)                    assert_allclose(speca, specb, atol=1e-08)
695: (0)            def test_cohere():
696: (4)                N = 1024
697: (4)                np.random.seed(19680801)
698: (4)                x = np.random.randn(N)
699: (4)                y = np.roll(x, 20)
700: (4)                y = np.convolve(y, np.ones(20) / 20., mode='same')
701: (4)                cohsq, f = mlab.cohere(x, y, NFFT=256, Fs=2, noverlap=128)
702: (4)                assert_allclose(np.mean(cohsq), 0.837, atol=1.e-3)
703: (4)                assert np.isreal(np.mean(cohsq))
704: (0)            class TestGaussianKDE:
705: (4)                def test_kde_integer_input(self):
706: (8)                    """Regression test for #1181."""
707: (8)                    x1 = np.arange(5)
708: (8)                    kde = mlab.GaussianKDE(x1)
709: (8)                    y_expected = [0.13480721, 0.18222869, 0.19514935, 0.18222869,
710: (22)                                  0.13480721]
711: (8)                    np.testing.assert_array_almost_equal(kde(x1), y_expected, decimal=6)
712: (4)                def test_gaussian_kde_covariance_caching(self):
713: (8)                    x1 = np.array([-7, -5, 1, 4, 5], dtype=float)
714: (8)                    xs = np.linspace(-10, 10, num=5)
715: (8)                    y_expected = [0.02463386, 0.04689208, 0.05395444, 0.05337754,
716: (22)                                  0.01664475]
717: (8)                    kde2 = mlab.GaussianKDE(x1, 'scott')
718: (8)                    y2 = kde2(xs)
719: (8)                    np.testing.assert_array_almost_equal(y_expected, y2, decimal=7)
720: (4)                def test_kde_bandwidth_method(self):
721: (8)                    np.random.seed(8765678)
722: (8)                    n_basesample = 50
723: (8)                    xn = np.random.randn(n_basesample)
724: (8)                    gkde = mlab.GaussianKDE(xn)
725: (8)                    gkde2 = mlab.GaussianKDE(xn, 'scott')
726: (8)                    gkde3 = mlab.GaussianKDE(xn, bw_method=gkde.factor)
727: (8)                    xs = np.linspace(-7, 7, 51)
728: (8)                    kdepdf = gkde.evaluate(xs)
729: (8)                    kdepdf2 = gkde2.evaluate(xs)
730: (8)                    assert kdepdf.all() == kdepdf2.all()
731: (8)                    kdepdf3 = gkde3.evaluate(xs)
732: (8)                    assert kdepdf.all() == kdepdf3.all()
733: (0)            class TestGaussianKDECustom:
734: (4)                def test_no_data(self):
735: (8)                    """Pass no data into the GaussianKDE class."""
736: (8)                    with pytest.raises(ValueError):
737: (12)                       mlab.GaussianKDE([])
738: (4)                def test_single_dataset_element(self):
739: (8)                    """Pass a single dataset element into the GaussianKDE class."""
740: (8)                    with pytest.raises(ValueError):
741: (12)                       mlab.GaussianKDE([42])
742: (4)                def test_silverman_multidim_dataset(self):
743: (8)                    """Test silverman's for a multi-dimensional array."""
744: (8)                    x1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
745: (8)                    with pytest.raises(np.linalg.LinAlgError):
746: (12)                       mlab.GaussianKDE(x1, "silverman")
747: (4)                def test_silverman_singledim_dataset(self):
748: (8)                    """Test silverman's output for a single dimension list."""
749: (8)                    x1 = np.array([-7, -5, 1, 4, 5])
750: (8)                    mygauss = mlab.GaussianKDE(x1, "silverman")
751: (8)                    y_expected = 0.76770389927475502
752: (8)                    assert_almost_equal(mygauss.covariance_factor(), y_expected, 7)
753: (4)                def test_scott_multidim_dataset(self):
754: (8)                    """Test scott's output for a multi-dimensional array."""
755: (8)                    x1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
756: (8)                    with pytest.raises(np.linalg.LinAlgError):
757: (12)                       mlab.GaussianKDE(x1, "scott")
758: (4)                def test_scott_singledim_dataset(self):
```

```
(8)                 """Test scott's output a single-dimensional array."""
(8)                 x1 = np.array([-7, -5, 1, 4, 5])
(8)                 mygauss = mlab.GaussianKDE(x1, "scott")
(8)                 y_expected = 0.72477966367769553
(8)                 assert_almost_equal(mygauss.covariance_factor(), y_expected, 7)
(4)             def test_scalar_empty_dataset(self):
(8)                 """Test the scalar's cov factor for an empty array."""
(8)                 with pytest.raises(ValueError):
(12)                    mlab.GaussianKDE([], bw_method=5)
(4)             def test_scalar_covariance_dataset(self):
(8)                 """Test a scalar's cov factor."""
(8)                 np.random.seed(8765678)
(8)                 n_basesample = 50
(8)                 multidim_data = [np.random.randn(n_basesample) for i in range(5)]
(8)                 kde = mlab.GaussianKDE(multidim_data, bw_method=0.5)
(8)                 assert kde.covariance_factor() == 0.5
(4)             def test_callable_covariance_dataset(self):
(8)                 """Test the callable's cov factor for a multi-dimensional array."""
(8)                 np.random.seed(8765678)
(8)                 n_basesample = 50
(8)                 multidim_data = [np.random.randn(n_basesample) for i in range(5)]
(8)                 def callable_fun(x):
(12)                    return 0.55
(8)                 kde = mlab.GaussianKDE(multidim_data, bw_method=callable_fun)
(8)                 assert kde.covariance_factor() == 0.55
(4)             def test_callable_singledim_dataset(self):
(8)                 """Test the callable's cov factor for a single-dimensional array."""
(8)                 np.random.seed(8765678)
(8)                 n_basesample = 50
(8)                 multidim_data = np.random.randn(n_basesample)
(8)                 kde = mlab.GaussianKDE(multidim_data, bw_method='silverman')
(8)                 y_expected = 0.48438841363348911
(8)                 assert_almost_equal(kde.covariance_factor(), y_expected, 7)
(4)             def test_wrong_bw_method(self):
(8)                 """Test the error message that should be called when bw is invalid."""
(8)                 np.random.seed(8765678)
(8)                 n_basesample = 50
(8)                 data = np.random.randn(n_basesample)
(8)                 with pytest.raises(ValueError):
(12)                    mlab.GaussianKDE(data, bw_method="invalid")
(0)         class TestGaussianKDEEvaluate:
(4)             def test_evaluate_diff_dim(self):
(8)                 """
(8)                 Test the evaluate method when the dim's of dataset and points have
(8)                 different dimensions.
(8)                 """
(8)                 x1 = np.arange(3, 10, 2)
(8)                 kde = mlab.GaussianKDE(x1)
(8)                 x2 = np.arange(3, 12, 2)
(8)                 y_expected = [
(12)                    0.08797252, 0.11774109, 0.11774109, 0.08797252, 0.0370153
(8)                 ]
(8)                 y = kde.evaluate(x2)
(8)                 np.testing.assert_array_almost_equal(y, y_expected, 7)
(4)             def test_evaluate_inv_dim(self):
(8)                 """
(8)                 Invert the dimensions; i.e., for a dataset of dimension 1 [3, 2, 4],
(8)                 the points should have a dimension of 3 [[3], [2], [4]].
(8)                 """
(8)                 np.random.seed(8765678)
(8)                 n_basesample = 50
(8)                 multidim_data = np.random.randn(n_basesample)
(8)                 kde = mlab.GaussianKDE(multidim_data)
(8)                 x2 = [[1], [2], [3]]
(8)                 with pytest.raises(ValueError):
(12)                    kde.evaluate(x2)
(4)             def test_evaluate_dim_and_num(self):
(8)                 """Tests if evaluated against a one by one array"""
(8)                 x1 = np.arange(3, 10, 2)
```

```
828: (8)                    x2 = np.array([3])
829: (8)                    kde = mlab.GaussianKDE(x1)
830: (8)                    y_expected = [0.08797252]
831: (8)                    y = kde.evaluate(x2)
832: (8)                    np.testing.assert_array_almost_equal(y, y_expected, 7)
833: (4)                def test_evaluate_point_dim_not_one(self):
834: (8)                    x1 = np.arange(3, 10, 2)
835: (8)                    x2 = [np.arange(3, 10, 2), np.arange(3, 10, 2)]
836: (8)                    kde = mlab.GaussianKDE(x1)
837: (8)                    with pytest.raises(ValueError):
838: (12)                       kde.evaluate(x2)
839: (4)                def test_evaluate_equal_dim_and_num_lt(self):
840: (8)                    x1 = np.arange(3, 10, 2)
841: (8)                    x2 = np.arange(3, 8, 2)
842: (8)                    kde = mlab.GaussianKDE(x1)
843: (8)                    y_expected = [0.08797252, 0.11774109, 0.11774109]
844: (8)                    y = kde.evaluate(x2)
845: (8)                    np.testing.assert_array_almost_equal(y, y_expected, 7)
846: (0)            def test_psd_onesided_norm():
847: (4)                u = np.array([0, 1, 2, 3, 1, 2, 1])
848: (4)                dt = 1.0
849: (4)                Su = np.abs(np.fft.fft(u) * dt)**2 / (dt * u.size)
850: (4)                P, f = mlab.psd(u, NFFT=u.size, Fs=1/dt, window=mlab.window_none,
851: (20)                               detrend=mlab.detrend_none, noverlap=0, pad_to=None,
852: (20)                               scale_by_freq=None,
853: (20)                               sides='onesided')
854: (4)                Su_1side = np.append([Su[0]], Su[1:4] + Su[4:][::-1])
855: (4)                assert_allclose(P, Su_1side, atol=1e-06)
856: (0)            def test_psd_oversampling():
857: (4)                """Test the case len(x) < NFFT for psd()."""
858: (4)                u = np.array([0, 1, 2, 3, 1, 2, 1])
859: (4)                dt = 1.0
860: (4)                Su = np.abs(np.fft.fft(u) * dt)**2 / (dt * u.size)
861: (4)                P, f = mlab.psd(u, NFFT=u.size*2, Fs=1/dt, window=mlab.window_none,
862: (20)                               detrend=mlab.detrend_none, noverlap=0, pad_to=None,
863: (20)                               scale_by_freq=None,
864: (20)                               sides='onesided')
865: (4)                Su_1side = np.append([Su[0]], Su[1:4] + Su[4:][::-1])
866: (4)                assert_almost_equal(np.sum(P), np.sum(Su_1side))  # same energy
```

File 197 - test_offsetbox.py:

```
1: (0)              from collections import namedtuple
2: (0)              import io
3: (0)              import numpy as np
4: (0)              from numpy.testing import assert_allclose
5: (0)              import pytest
6: (0)              from matplotlib.testing.decorators import check_figures_equal,
image_comparison
7: (0)              import matplotlib.pyplot as plt
8: (0)              import matplotlib.patches as mpatches
9: (0)              import matplotlib.lines as mlines
10: (0)             from matplotlib.backend_bases import MouseButton, MouseEvent
11: (0)             from matplotlib.offsetbox import (
12: (4)                 AnchoredOffsetbox, AnnotationBbox, AnchoredText, DrawingArea, HPacker,
13: (4)                 OffsetBox, OffsetImage, PaddedBox, TextArea, VPacker, _get_packed_offsets)
14: (0)             @image_comparison(['offsetbox_clipping'], remove_text=True)
15: (0)             def test_offsetbox_clipping():
16: (4)                 fig, ax = plt.subplots()
17: (4)                 size = 100
18: (4)                 da = DrawingArea(size, size, clip=True)
19: (4)                 assert da.clip_children
20: (4)                 bg = mpatches.Rectangle((0, 0), size, size,
21: (28)                                        facecolor='#CCCCCC',
22: (28)                                        edgecolor='None',
23: (28)                                        linewidth=0)
24: (4)                 line = mlines.Line2D([-size*.5, size*1.5], [size/2, size/2],
```

```
 25: (25)                                    color='black',
 26: (25)                                    linewidth=10)
 27: (4)                 anchored_box = AnchoredOffsetbox(
 28: (8)                     loc='center',
 29: (8)                     child=da,
 30: (8)                     pad=0.,
 31: (8)                     frameon=False,
 32: (8)                     bbox_to_anchor=(.5, .5),
 33: (8)                     bbox_transform=ax.transAxes,
 34: (8)                     borderpad=0.)
 35: (4)                 da.add_artist(bg)
 36: (4)                 da.add_artist(line)
 37: (4)                 ax.add_artist(anchored_box)
 38: (4)                 ax.set_xlim((0, 1))
 39: (4)                 ax.set_ylim((0, 1))
 40: (0)             def test_offsetbox_clip_children():
 41: (4)                 fig, ax = plt.subplots()
 42: (4)                 size = 100
 43: (4)                 da = DrawingArea(size, size, clip=True)
 44: (4)                 bg = mpatches.Rectangle((0, 0), size, size,
 45: (28)                                         facecolor='#CCCCCC',
 46: (28)                                         edgecolor='None',
 47: (28)                                         linewidth=0)
 48: (4)                 line = mlines.Line2D([-size*.5, size*1.5], [size/2, size/2],
 49: (25)                                    color='black',
 50: (25)                                    linewidth=10)
 51: (4)                 anchored_box = AnchoredOffsetbox(
 52: (8)                     loc='center',
 53: (8)                     child=da,
 54: (8)                     pad=0.,
 55: (8)                     frameon=False,
 56: (8)                     bbox_to_anchor=(.5, .5),
 57: (8)                     bbox_transform=ax.transAxes,
 58: (8)                     borderpad=0.)
 59: (4)                 da.add_artist(bg)
 60: (4)                 da.add_artist(line)
 61: (4)                 ax.add_artist(anchored_box)
 62: (4)                 fig.canvas.draw()
 63: (4)                 assert not fig.stale
 64: (4)                 da.clip_children = True
 65: (4)                 assert fig.stale
 66: (0)             def test_offsetbox_loc_codes():
 67: (4)                 codes = {'upper right': 1,
 68: (13)                         'upper left': 2,
 69: (13)                         'lower left': 3,
 70: (13)                         'lower right': 4,
 71: (13)                         'right': 5,
 72: (13)                         'center left': 6,
 73: (13)                         'center right': 7,
 74: (13)                         'lower center': 8,
 75: (13)                         'upper center': 9,
 76: (13)                         'center': 10,
 77: (13)                         }
 78: (4)                 fig, ax = plt.subplots()
 79: (4)                 da = DrawingArea(100, 100)
 80: (4)                 for code in codes:
 81: (8)                     anchored_box = AnchoredOffsetbox(loc=code, child=da)
 82: (8)                     ax.add_artist(anchored_box)
 83: (4)                 fig.canvas.draw()
 84: (0)             def test_expand_with_tight_layout():
 85: (4)                 fig, ax = plt.subplots()
 86: (4)                 d1 = [1, 2]
 87: (4)                 d2 = [2, 1]
 88: (4)                 ax.plot(d1, label='series 1')
 89: (4)                 ax.plot(d2, label='series 2')
 90: (4)                 ax.legend(ncols=2, mode='expand')
 91: (4)                 fig.tight_layout()  # where the crash used to happen
 92: (0)             @pytest.mark.parametrize('widths',
 93: (25)                                    ([150], [150, 150, 150], [0.1], [0.1, 0.1]))
```

```
@pytest.mark.parametrize('total', (250, 100, 0, -1, None))
@pytest.mark.parametrize('sep', (250, 1, 0, -1))
@pytest.mark.parametrize('mode', ("expand", "fixed", "equal"))
def test_get_packed_offsets(widths, total, sep, mode):
    _get_packed_offsets(widths, total, sep, mode=mode)
_Params = namedtuple('_Params', 'wd_list, total, sep, expected')
@pytest.mark.parametrize('widths, total, sep, expected', [
    _Params(  # total=None
        [3, 1, 2], total=None, sep=1, expected=(8, [0, 4, 6])),
    _Params(  # total larger than required
        [3, 1, 2], total=10, sep=1, expected=(10, [0, 4, 6])),
    _Params(  # total smaller than required
        [3, 1, 2], total=5, sep=1, expected=(5, [0, 4, 6])),
])
def test_get_packed_offsets_fixed(widths, total, sep, expected):
    result = _get_packed_offsets(widths, total, sep, mode='fixed')
    assert result[0] == expected[0]
    assert_allclose(result[1], expected[1])
@pytest.mark.parametrize('widths, total, sep, expected', [
    _Params(  # total=None (implicit 1)
        [.1, .1, .1], total=None, sep=None, expected=(1, [0, .45, .9])),
    _Params(  # total larger than sum of widths
        [3, 1, 2], total=10, sep=1, expected=(10, [0, 5, 8])),
    _Params(  # total smaller sum of widths: overlapping boxes
        [3, 1, 2], total=5, sep=1, expected=(5, [0, 2.5, 3])),
])
def test_get_packed_offsets_expand(widths, total, sep, expected):
    result = _get_packed_offsets(widths, total, sep, mode='expand')
    assert result[0] == expected[0]
    assert_allclose(result[1], expected[1])
@pytest.mark.parametrize('widths, total, sep, expected', [
    _Params(  # total larger than required
        [3, 2, 1], total=6, sep=None, expected=(6, [0, 2, 4])),
    _Params(  # total smaller sum of widths: overlapping boxes
        [3, 2, 1, .5], total=2, sep=None, expected=(2, [0, 0.5, 1, 1.5])),
    _Params(  # total larger than required
        [.5, 1, .2], total=None, sep=1, expected=(6, [0, 2, 4])),
])
def test_get_packed_offsets_equal(widths, total, sep, expected):
    result = _get_packed_offsets(widths, total, sep, mode='equal')
    assert result[0] == expected[0]
    assert_allclose(result[1], expected[1])
def test_get_packed_offsets_equal_total_none_sep_none():
    with pytest.raises(ValueError):
        _get_packed_offsets([1, 1, 1], total=None, sep=None, mode='equal')
@pytest.mark.parametrize('child_type', ['draw', 'image', 'text'])
@pytest.mark.parametrize('boxcoords',
                         ['axes fraction', 'axes pixels', 'axes points',
                          'data'])
def test_picking(child_type, boxcoords):
    if child_type == 'draw':
        picking_child = DrawingArea(5, 5)
        picking_child.add_artist(mpatches.Rectangle((0, 0), 5, 5, linewidth=0))
    elif child_type == 'image':
        im = np.ones((5, 5))
        im[2, 2] = 0
        picking_child = OffsetImage(im)
    elif child_type == 'text':
        picking_child = TextArea('\N{Black Square}', textprops={'fontsize': 5})
    else:
        assert False, f'Unknown picking child type {child_type}'
    fig, ax = plt.subplots()
    ab = AnnotationBbox(picking_child, (0.5, 0.5), boxcoords=boxcoords)
    ab.set_picker(True)
    ax.add_artist(ab)
    calls = []
    fig.canvas.mpl_connect('pick_event', lambda event: calls.append(event))
```

```
(4)                 if boxcoords == 'axes points':
(8)                     x, y = ax.transAxes.transform_point((0, 0))
(8)                     x += 0.5 * fig.dpi / 72
(8)                     y += 0.5 * fig.dpi / 72
(4)                 elif boxcoords == 'axes pixels':
(8)                     x, y = ax.transAxes.transform_point((0, 0))
(8)                     x += 0.5
(8)                     y += 0.5
(4)                 else:
(8)                     x, y = ax.transAxes.transform_point((0.5, 0.5))
(4)                 fig.canvas.draw()
(4)                 calls.clear()
(4)                 MouseEvent(
(8)                     "button_press_event", fig.canvas, x, y, MouseButton.LEFT)._process()
(4)                 assert len(calls) == 1 and calls[0].artist == ab
(4)                 ax.set_xlim(-1, 0)
(4)                 ax.set_ylim(-1, 0)
(4)                 fig.canvas.draw()
(4)                 calls.clear()
(4)                 MouseEvent(
(8)                     "button_press_event", fig.canvas, x, y, MouseButton.LEFT)._process()
(4)                 assert len(calls) == 0
(0)             @image_comparison(['anchoredtext_align.png'], remove_text=True, style='mpl20')
(0)             def test_anchoredtext_horizontal_alignment():
(4)                 fig, ax = plt.subplots()
(4)                 text0 = AnchoredText("test\ntest long text", loc="center left",
(25)                                     pad=0.2, prop={"ha": "left"})
(4)                 ax.add_artist(text0)
(4)                 text1 = AnchoredText("test\ntest long text", loc="center",
(25)                                     pad=0.2, prop={"ha": "center"})
(4)                 ax.add_artist(text1)
(4)                 text2 = AnchoredText("test\ntest long text", loc="center right",
(25)                                     pad=0.2, prop={"ha": "right"})
(4)                 ax.add_artist(text2)
(0)             @pytest.mark.parametrize("extent_kind", ["window_extent", "tightbbox"])
(0)             def test_annotationbbox_extents(extent_kind):
(4)                 plt.rcParams.update(plt.rcParamsDefault)
(4)                 fig, ax = plt.subplots(figsize=(4, 3), dpi=100)
(4)                 ax.axis([0, 1, 0, 1])
(4)                 an1 = ax.annotate("Annotation", xy=(.9, .9), xytext=(1.1, 1.1),
(22)                                  arrowprops=dict(arrowstyle="->"), clip_on=False,
(22)                                  va="baseline", ha="left")
(4)                 da = DrawingArea(20, 20, 0, 0, clip=True)
(4)                 p = mpatches.Circle((-10, 30), 32)
(4)                 da.add_artist(p)
(4)                 ab3 = AnnotationBbox(da, [.5, .5], xybox=(-0.2, 0.5), xycoords='data',
(25)                                     boxcoords="axes fraction", box_alignment=(0., .5),
(25)                                     arrowprops=dict(arrowstyle="->"))
(4)                 ax.add_artist(ab3)
(4)                 im = OffsetImage(np.random.rand(10, 10), zoom=3)
(4)                 im.image.axes = ax
(4)                 ab6 = AnnotationBbox(im, (0.5, -.3), xybox=(0, 75),
(25)                                     xycoords='axes fraction',
(25)                                     boxcoords="offset points", pad=0.3,
(25)                                     arrowprops=dict(arrowstyle="->"))
(4)                 ax.add_artist(ab6)
(4)                 bb1 = getattr(an1, f"get_{extent_kind}")()
(4)                 target1 = [332.9, 242.8, 467.0, 298.9]
(4)                 assert_allclose(bb1.extents, target1, atol=2)
(4)                 bb3 = getattr(ab3, f"get_{extent_kind}")()
(4)                 target3 = [-17.6, 129.0, 200.7, 167.9]
(4)                 assert_allclose(bb3.extents, target3, atol=2)
(4)                 bb6 = getattr(ab6, f"get_{extent_kind}")()
(4)                 target6 = [180.0, -32.0, 230.0, 92.9]
(4)                 assert_allclose(bb6.extents, target6, atol=2)
(4)                 buf = io.BytesIO()
(4)                 fig.savefig(buf, bbox_inches='tight')
(4)                 buf.seek(0)
(4)                 shape = plt.imread(buf).shape
```

```
230: (4)                targetshape = (350, 504, 4)
231: (4)                assert_allclose(shape, targetshape, atol=2)
232: (4)                fig.canvas.draw()
233: (4)                fig.tight_layout()
234: (4)                fig.canvas.draw()
235: (0)            def test_zorder():
236: (4)                assert OffsetBox(zorder=42).zorder == 42
237: (0)            def test_arrowprops_copied():
238: (4)                da = DrawingArea(20, 20, 0, 0, clip=True)
239: (4)                arrowprops = {"arrowstyle": "->", "relpos": (.3, .7)}
240: (4)                ab = AnnotationBbox(da, [.5, .5], xybox=(-0.2, 0.5), xycoords='data',
241: (24)                                   boxcoords="axes fraction", box_alignment=(0., .5),
242: (24)                                   arrowprops=arrowprops)
243: (4)                assert ab.arrowprops is not ab
244: (4)                assert arrowprops["relpos"] == (.3, .7)
245: (0)            @pytest.mark.parametrize("align", ["baseline", "bottom", "top",
246: (35)                                              "left", "right", "center"])
247: (0)            def test_packers(align):
248: (4)                fig = plt.figure(dpi=72)
249: (4)                renderer = fig.canvas.get_renderer()
250: (4)                x1, y1 = 10, 30
251: (4)                x2, y2 = 20, 60
252: (4)                r1 = DrawingArea(x1, y1)
253: (4)                r2 = DrawingArea(x2, y2)
254: (4)                hpacker = HPacker(children=[r1, r2], align=align)
255: (4)                hpacker.draw(renderer)
256: (4)                bbox = hpacker.get_bbox(renderer)
257: (4)                px, py = hpacker.get_offset(bbox, renderer)
258: (4)                assert_allclose(bbox.bounds, (0, 0, x1 + x2, max(y1, y2)))
259: (4)                if align in ("baseline", "left", "bottom"):
260: (8)                    y_height = 0
261: (4)                elif align in ("right", "top"):
262: (8)                    y_height = y2 - y1
263: (4)                elif align == "center":
264: (8)                    y_height = (y2 - y1) / 2
265: (4)                assert_allclose([child.get_offset() for child in hpacker.get_children()],
266: (20)                               [(px, py + y_height), (px + x1, py)])
267: (4)                vpacker = VPacker(children=[r1, r2], align=align)
268: (4)                vpacker.draw(renderer)
269: (4)                bbox = vpacker.get_bbox(renderer)
270: (4)                px, py = vpacker.get_offset(bbox, renderer)
271: (4)                assert_allclose(bbox.bounds, (0, -max(y1, y2), max(x1, x2), y1 + y2))
272: (4)                if align in ("baseline", "left", "bottom"):
273: (8)                    x_height = 0
274: (4)                elif align in ("right", "top"):
275: (8)                    x_height = x2 - x1
276: (4)                elif align == "center":
277: (8)                    x_height = (x2 - x1) / 2
278: (4)                assert_allclose([child.get_offset() for child in vpacker.get_children()],
279: (20)                               [(px + x_height, py), (px, py - y2)])
280: (0)            def test_paddedbox_default_values():
281: (4)                fig, ax = plt.subplots()
282: (4)                at = AnchoredText("foo",  'upper left')
283: (4)                pb = PaddedBox(at, patch_attrs={'facecolor': 'r'}, draw_frame=True)
284: (4)                ax.add_artist(pb)
285: (4)                fig.draw_without_rendering()
286: (0)            def test_annotationbbox_properties():
287: (4)                ab = AnnotationBbox(DrawingArea(20, 20, 0, 0, clip=True), (0.5, 0.5),
288: (24)                                   xycoords='data')
289: (4)                assert ab.xyann == (0.5, 0.5)  # xy if xybox not given
290: (4)                assert ab.anncoords == 'data'  # xycoords if boxcoords not given
291: (4)                ab = AnnotationBbox(DrawingArea(20, 20, 0, 0, clip=True), (0.5, 0.5),
292: (24)                                   xybox=(-0.2, 0.4), xycoords='data',
293: (24)                                   boxcoords='axes fraction')
294: (4)                assert ab.xyann == (-0.2, 0.4)  # xybox if given
295: (4)                assert ab.anncoords == 'axes fraction'  # boxcoords if given
296: (0)            def test_textarea_properties():
297: (4)                ta = TextArea('Foo')
298: (4)                assert ta.get_text() == 'Foo'
```

```
299: (4)                assert not ta.get_multilinebaseline()
300: (4)                ta.set_text('Bar')
301: (4)                ta.set_multilinebaseline(True)
302: (4)                assert ta.get_text() == 'Bar'
303: (4)                assert ta.get_multilinebaseline()
304: (0)            @check_figures_equal()
305: (0)            def test_textarea_set_text(fig_test, fig_ref):
306: (4)                ax_ref = fig_ref.add_subplot()
307: (4)                text0 = AnchoredText("Foo", "upper left")
308: (4)                ax_ref.add_artist(text0)
309: (4)                ax_test = fig_test.add_subplot()
310: (4)                text1 = AnchoredText("Bar", "upper left")
311: (4)                ax_test.add_artist(text1)
312: (4)                text1.txt.set_text("Foo")
313: (0)            @image_comparison(['paddedbox.png'], remove_text=True, style='mpl20')
314: (0)            def test_paddedbox():
315: (4)                fig, ax = plt.subplots()
316: (4)                ta = TextArea("foo")
317: (4)                pb = PaddedBox(ta, pad=5, patch_attrs={'facecolor': 'r'}, draw_frame=True)
318: (4)                ab = AnchoredOffsetbox('upper left', child=pb)
319: (4)                ax.add_artist(ab)
320: (4)                ta = TextArea("bar")
321: (4)                pb = PaddedBox(ta, pad=10, patch_attrs={'facecolor': 'b'})
322: (4)                ab = AnchoredOffsetbox('upper right', child=pb)
323: (4)                ax.add_artist(ab)
324: (4)                ta = TextArea("foobar")
325: (4)                pb = PaddedBox(ta, pad=15, draw_frame=True)
326: (4)                ab = AnchoredOffsetbox('lower right', child=pb)
327: (4)                ax.add_artist(ab)
328: (0)            def test_remove_draggable():
329: (4)                fig, ax = plt.subplots()
330: (4)                an = ax.annotate("foo", (.5, .5))
331: (4)                an.draggable(True)
332: (4)                an.remove()
333: (4)                MouseEvent("button_release_event", fig.canvas, 1, 1)._process()
334: (0)            def test_draggable_in_subfigure():
335: (4)                fig = plt.figure()
336: (4)                ann = fig.subfigures().add_axes([0, 0, 1, 1]).annotate("foo", (0, 0))
337: (4)                ann.draggable(True)
338: (4)                fig.canvas.draw()  # Texts are non-pickable until the first draw.
339: (4)                MouseEvent("button_press_event", fig.canvas, 1, 1)._process()
340: (4)                assert ann._draggable.got_artist

----------------------------------------

File 198 - test_patches.py:

1: (0)              """
2: (0)              Tests specific to the patches module.
3: (0)              """
4: (0)              import platform
5: (0)              import numpy as np
6: (0)              from numpy.testing import assert_almost_equal, assert_array_equal
7: (0)              import pytest
8: (0)              import matplotlib as mpl
9: (0)              from matplotlib.patches import (Annulus, Ellipse, Patch, Polygon, Rectangle,
10: (32)                                            FancyArrowPatch, FancyArrow, BoxStyle, Arc)
11: (0)             from matplotlib.testing.decorators import image_comparison,
check_figures_equal
12: (0)             from matplotlib.transforms import Bbox
13: (0)             import matplotlib.pyplot as plt
14: (0)             from matplotlib import (
15: (4)                 collections as mcollections, colors as mcolors, patches as mpatches,
16: (4)                 path as mpath, transforms as mtransforms, rcParams)
17: (0)             def test_Polygon_close():
18: (4)                 xy = [[0, 0], [0, 1], [1, 1]]
19: (4)                 xyclosed = xy + [[0, 0]]
20: (4)                 p = Polygon(xy, closed=True)
21: (4)                 assert p.get_closed()
```

```
 22: (4)                 assert_array_equal(p.get_xy(), xyclosed)
 23: (4)                 p.set_xy(xy)
 24: (4)                 assert_array_equal(p.get_xy(), xyclosed)
 25: (4)                 p = Polygon(xyclosed, closed=False)
 26: (4)                 assert_array_equal(p.get_xy(), xy)
 27: (4)                 p.set_xy(xyclosed)
 28: (4)                 assert_array_equal(p.get_xy(), xy)
 29: (4)                 p = Polygon(xy, closed=False)
 30: (4)                 assert not p.get_closed()
 31: (4)                 assert_array_equal(p.get_xy(), xy)
 32: (4)                 p.set_xy(xy)
 33: (4)                 assert_array_equal(p.get_xy(), xy)
 34: (4)                 p = Polygon(xyclosed, closed=True)
 35: (4)                 assert_array_equal(p.get_xy(), xyclosed)
 36: (4)                 p.set_xy(xyclosed)
 37: (4)                 assert_array_equal(p.get_xy(), xyclosed)
 38: (0)             def test_corner_center():
 39: (4)                 loc = [10, 20]
 40: (4)                 width = 1
 41: (4)                 height = 2
 42: (4)                 corners = ((10, 20), (11, 20), (11, 22), (10, 22))
 43: (4)                 rect = Rectangle(loc, width, height)
 44: (4)                 assert_array_equal(rect.get_corners(), corners)
 45: (4)                 assert_array_equal(rect.get_center(), (10.5, 21))
 46: (4)                 corners_rot = ((10, 20), (10, 21), (8, 21), (8, 20))
 47: (4)                 rect.set_angle(90)
 48: (4)                 assert_array_equal(rect.get_corners(), corners_rot)
 49: (4)                 assert_array_equal(rect.get_center(), (9, 20.5))
 50: (4)                 theta = 33
 51: (4)                 t = mtransforms.Affine2D().rotate_around(*loc, np.deg2rad(theta))
 52: (4)                 corners_rot = t.transform(corners)
 53: (4)                 rect.set_angle(theta)
 54: (4)                 assert_almost_equal(rect.get_corners(), corners_rot)
 55: (4)                 loc = [loc[0] + width / 2,
 56: (11)                       loc[1] + height / 2]
 57: (4)                 ellipse = Ellipse(loc, width, height)
 58: (4)                 assert_array_equal(ellipse.get_corners(), corners)
 59: (4)                 corners_rot = ((11.5, 20.5), (11.5, 21.5), (9.5, 21.5), (9.5, 20.5))
 60: (4)                 ellipse.set_angle(90)
 61: (4)                 assert_array_equal(ellipse.get_corners(), corners_rot)
 62: (4)                 assert_array_equal(ellipse.get_center(), loc)
 63: (4)                 theta = 33
 64: (4)                 t = mtransforms.Affine2D().rotate_around(*loc, np.deg2rad(theta))
 65: (4)                 corners_rot = t.transform(corners)
 66: (4)                 ellipse.set_angle(theta)
 67: (4)                 assert_almost_equal(ellipse.get_corners(), corners_rot)
 68: (0)             def test_ellipse_vertices():
 69: (4)                 ellipse = Ellipse(xy=(0, 0), width=0, height=0, angle=0)
 70: (4)                 assert_almost_equal(
 71: (8)                     ellipse.get_vertices(),
 72: (8)                     [(0.0, 0.0), (0.0, 0.0)],
 73: (4)                 )
 74: (4)                 assert_almost_equal(
 75: (8)                     ellipse.get_co_vertices(),
 76: (8)                     [(0.0, 0.0), (0.0, 0.0)],
 77: (4)                 )
 78: (4)                 ellipse = Ellipse(xy=(0, 0), width=2, height=1, angle=30)
 79: (4)                 assert_almost_equal(
 80: (8)                     ellipse.get_vertices(),
 81: (8)                     [
 82: (12)                        (
 83: (16)                            ellipse.center[0] + ellipse.width / 4 * np.sqrt(3),
 84: (16)                            ellipse.center[1] + ellipse.width / 4,
 85: (12)                        ),
 86: (12)                        (
 87: (16)                            ellipse.center[0] - ellipse.width / 4 * np.sqrt(3),
 88: (16)                            ellipse.center[1] - ellipse.width / 4,
 89: (12)                        ),
 90: (8)                     ],
```

```
 91: (4)                 )
 92: (4)                 assert_almost_equal(
 93: (8)                     ellipse.get_co_vertices(),
 94: (8)                     [
 95: (12)                        (
 96: (16)                            ellipse.center[0] - ellipse.height / 4,
 97: (16)                            ellipse.center[1] + ellipse.height / 4 * np.sqrt(3),
 98: (12)                        ),
 99: (12)                        (
100: (16)                            ellipse.center[0] + ellipse.height / 4,
101: (16)                            ellipse.center[1] - ellipse.height / 4 * np.sqrt(3),
102: (12)                        ),
103: (8)                     ],
104: (4)                 )
105: (4)                 v1, v2 = np.array(ellipse.get_vertices())
106: (4)                 np.testing.assert_almost_equal((v1 + v2) / 2, ellipse.center)
107: (4)                 v1, v2 = np.array(ellipse.get_co_vertices())
108: (4)                 np.testing.assert_almost_equal((v1 + v2) / 2, ellipse.center)
109: (4)                 ellipse = Ellipse(xy=(2.252, -10.859), width=2.265, height=1.98,
angle=68.78)
110: (4)                 v1, v2 = np.array(ellipse.get_vertices())
111: (4)                 np.testing.assert_almost_equal((v1 + v2) / 2, ellipse.center)
112: (4)                 v1, v2 = np.array(ellipse.get_co_vertices())
113: (4)                 np.testing.assert_almost_equal((v1 + v2) / 2, ellipse.center)
114: (0)             def test_rotate_rect():
115: (4)                 loc = np.asarray([1.0, 2.0])
116: (4)                 width = 2
117: (4)                 height = 3
118: (4)                 angle = 30.0
119: (4)                 rect1 = Rectangle(loc, width, height, angle=angle)
120: (4)                 rect2 = Rectangle(loc, width, height)
121: (4)                 angle_rad = np.pi * angle / 180.0
122: (4)                 rotation_matrix = np.array([[np.cos(angle_rad), -np.sin(angle_rad)],
123: (32)                                            [np.sin(angle_rad),  np.cos(angle_rad)]])
124: (4)                 new_verts = np.inner(rotation_matrix, rect2.get_verts() - loc).T + loc
125: (4)                 assert_almost_equal(rect1.get_verts(), new_verts)
126: (0)             @check_figures_equal(extensions=['png'])
127: (0)             def test_rotate_rect_draw(fig_test, fig_ref):
128: (4)                 ax_test = fig_test.add_subplot()
129: (4)                 ax_ref = fig_ref.add_subplot()
130: (4)                 loc = (0, 0)
131: (4)                 width, height = (1, 1)
132: (4)                 angle = 30
133: (4)                 rect_ref = Rectangle(loc, width, height, angle=angle)
134: (4)                 ax_ref.add_patch(rect_ref)
135: (4)                 assert rect_ref.get_angle() == angle
136: (4)                 rect_test = Rectangle(loc, width, height)
137: (4)                 assert rect_test.get_angle() == 0
138: (4)                 ax_test.add_patch(rect_test)
139: (4)                 rect_test.set_angle(angle)
140: (4)                 assert rect_test.get_angle() == angle
141: (0)             @check_figures_equal(extensions=['png'])
142: (0)             def test_dash_offset_patch_draw(fig_test, fig_ref):
143: (4)                 ax_test = fig_test.add_subplot()
144: (4)                 ax_ref = fig_ref.add_subplot()
145: (4)                 loc = (0.1, 0.1)
146: (4)                 width, height = (0.8, 0.8)
147: (4)                 rect_ref = Rectangle(loc, width, height, linewidth=3, edgecolor='b',
148: (48)                                                         linestyle=(0, [6, 6]))
149: (4)                 rect_ref2 = Rectangle(loc, width, height, linewidth=3, edgecolor='r',
150: (44)                                                     linestyle=(0, [0, 6, 6, 0]))
151: (4)                 assert rect_ref.get_linestyle() == (0, [6, 6])
152: (4)                 assert rect_ref2.get_linestyle() == (0, [0, 6, 6, 0])
153: (4)                 ax_ref.add_patch(rect_ref)
154: (4)                 ax_ref.add_patch(rect_ref2)
155: (4)                 rect_test = Rectangle(loc, width, height, linewidth=3, edgecolor='b',
156: (52)                                                             linestyle=(0, [6, 6]))
157: (4)                 rect_test2 = Rectangle(loc, width, height, linewidth=3, edgecolor='r',
158: (52)                                                             linestyle=(6, [6, 6]))
```

```
159: (4)                assert rect_test.get_linestyle() == (0, [6, 6])
160: (4)                assert rect_test2.get_linestyle() == (6, [6, 6])
161: (4)                ax_test.add_patch(rect_test)
162: (4)                ax_test.add_patch(rect_test2)
163: (0)            def test_negative_rect():
164: (4)                pos_vertices = Rectangle((-3, -2), 3, 2).get_verts()[:-1]
165: (4)                neg_vertices = Rectangle((0, 0), -3, -2).get_verts()[:-1]
166: (4)                assert_array_equal(np.roll(neg_vertices, 2, 0), pos_vertices)
167: (0)            @image_comparison(['clip_to_bbox'])
168: (0)            def test_clip_to_bbox():
169: (4)                fig, ax = plt.subplots()
170: (4)                ax.set_xlim([-18, 20])
171: (4)                ax.set_ylim([-150, 100])
172: (4)                path = mpath.Path.unit_regular_star(8).deepcopy()
173: (4)                path.vertices *= [10, 100]
174: (4)                path.vertices -= [5, 25]
175: (4)                path2 = mpath.Path.unit_circle().deepcopy()
176: (4)                path2.vertices *= [10, 100]
177: (4)                path2.vertices += [10, -25]
178: (4)                combined = mpath.Path.make_compound_path(path, path2)
179: (4)                patch = mpatches.PathPatch(
180: (8)                    combined, alpha=0.5, facecolor='coral', edgecolor='none')
181: (4)                ax.add_patch(patch)
182: (4)                bbox = mtransforms.Bbox([[-12, -77.5], [50, -110]])
183: (4)                result_path = combined.clip_to_bbox(bbox)
184: (4)                result_patch = mpatches.PathPatch(
185: (8)                    result_path, alpha=0.5, facecolor='green', lw=4, edgecolor='black')
186: (4)                ax.add_patch(result_patch)
187: (0)            @image_comparison(['patch_alpha_coloring'], remove_text=True)
188: (0)            def test_patch_alpha_coloring():
189: (4)                """
190: (4)                Test checks that the patch and collection are rendered with the specified
191: (4)                alpha values in their facecolor and edgecolor.
192: (4)                """
193: (4)                star = mpath.Path.unit_regular_star(6)
194: (4)                circle = mpath.Path.unit_circle()
195: (4)                verts = np.concatenate([circle.vertices, star.vertices[::-1]])
196: (4)                codes = np.concatenate([circle.codes, star.codes])
197: (4)                cut_star1 = mpath.Path(verts, codes)
198: (4)                cut_star2 = mpath.Path(verts + 1, codes)
199: (4)                ax = plt.axes()
200: (4)                col = mcollections.PathCollection([cut_star2],
201: (38)                                                  linewidth=5, linestyles='dashdot',
202: (38)                                                  facecolor=(1, 0, 0, 0.5),
203: (38)                                                  edgecolor=(0, 0, 1, 0.75))
204: (4)                ax.add_collection(col)
205: (4)                patch = mpatches.PathPatch(cut_star1,
206: (31)                                           linewidth=5, linestyle='dashdot',
207: (31)                                           facecolor=(1, 0, 0, 0.5),
208: (31)                                           edgecolor=(0, 0, 1, 0.75))
209: (4)                ax.add_patch(patch)
210: (4)                ax.set_xlim(-1, 2)
211: (4)                ax.set_ylim(-1, 2)
212: (0)            @image_comparison(['patch_alpha_override'], remove_text=True)
213: (0)            def test_patch_alpha_override():
214: (4)                star = mpath.Path.unit_regular_star(6)
215: (4)                circle = mpath.Path.unit_circle()
216: (4)                verts = np.concatenate([circle.vertices, star.vertices[::-1]])
217: (4)                codes = np.concatenate([circle.codes, star.codes])
218: (4)                cut_star1 = mpath.Path(verts, codes)
219: (4)                cut_star2 = mpath.Path(verts + 1, codes)
220: (4)                ax = plt.axes()
221: (4)                col = mcollections.PathCollection([cut_star2],
222: (38)                                                  linewidth=5, linestyles='dashdot',
223: (38)                                                  alpha=0.25,
224: (38)                                                  facecolor=(1, 0, 0, 0.5),
225: (38)                                                  edgecolor=(0, 0, 1, 0.75))
226: (4)                ax.add_collection(col)
227: (4)                patch = mpatches.PathPatch(cut_star1,
```

```
228: (31)                                       linewidth=5, linestyle='dashdot',
229: (31)                                       alpha=0.25,
230: (31)                                       facecolor=(1, 0, 0, 0.5),
231: (31)                                       edgecolor=(0, 0, 1, 0.75))
232: (4)                ax.add_patch(patch)
233: (4)                ax.set_xlim(-1, 2)
234: (4)                ax.set_ylim(-1, 2)
235: (0)            @mpl.style.context('default')
236: (0)            def test_patch_color_none():
237: (4)                c = plt.Circle((0, 0), 1, facecolor='none', alpha=1)
238: (4)                assert c.get_facecolor()[0] == 0
239: (0)            @image_comparison(['patch_custom_linestyle'], remove_text=True)
240: (0)            def test_patch_custom_linestyle():
241: (4)                star = mpath.Path.unit_regular_star(6)
242: (4)                circle = mpath.Path.unit_circle()
243: (4)                verts = np.concatenate([circle.vertices, star.vertices[::-1]])
244: (4)                codes = np.concatenate([circle.codes, star.codes])
245: (4)                cut_star1 = mpath.Path(verts, codes)
246: (4)                cut_star2 = mpath.Path(verts + 1, codes)
247: (4)                ax = plt.axes()
248: (4)                col = mcollections.PathCollection(
249: (8)                    [cut_star2],
250: (8)                    linewidth=5, linestyles=[(0, (5, 7, 10, 7))],
251: (8)                    facecolor=(1, 0, 0), edgecolor=(0, 0, 1))
252: (4)                ax.add_collection(col)
253: (4)                patch = mpatches.PathPatch(
254: (8)                    cut_star1,
255: (8)                    linewidth=5, linestyle=(0, (5, 7, 10, 7)),
256: (8)                    facecolor=(1, 0, 0), edgecolor=(0, 0, 1))
257: (4)                ax.add_patch(patch)
258: (4)                ax.set_xlim(-1, 2)
259: (4)                ax.set_ylim(-1, 2)
260: (0)            def test_patch_linestyle_accents():
261: (4)                star = mpath.Path.unit_regular_star(6)
262: (4)                circle = mpath.Path.unit_circle()
263: (4)                verts = np.concatenate([circle.vertices, star.vertices[::-1]])
264: (4)                codes = np.concatenate([circle.codes, star.codes])
265: (4)                linestyles = ["-", "--", "-.", ":",
266: (18)                             "solid", "dashed", "dashdot", "dotted"]
267: (4)                fig, ax = plt.subplots()
268: (4)                for i, ls in enumerate(linestyles):
269: (8)                    star = mpath.Path(verts + i, codes)
270: (8)                    patch = mpatches.PathPatch(star,
271: (35)                                              linewidth=3, linestyle=ls,
272: (35)                                              facecolor=(1, 0, 0),
273: (35)                                              edgecolor=(0, 0, 1))
274: (8)                    ax.add_patch(patch)
275: (4)                ax.set_xlim([-1, i + 1])
276: (4)                ax.set_ylim([-1, i + 1])
277: (4)                fig.canvas.draw()
278: (0)            @check_figures_equal(extensions=['png'])
279: (0)            def test_patch_linestyle_none(fig_test, fig_ref):
280: (4)                circle = mpath.Path.unit_circle()
281: (4)                ax_test = fig_test.add_subplot()
282: (4)                ax_ref = fig_ref.add_subplot()
283: (4)                for i, ls in enumerate(['none', 'None', ' ', '']):
284: (8)                    path = mpath.Path(circle.vertices + i, circle.codes)
285: (8)                    patch = mpatches.PathPatch(path,
286: (35)                                              linewidth=3, linestyle=ls,
287: (35)                                              facecolor=(1, 0, 0),
288: (35)                                              edgecolor=(0, 0, 1))
289: (8)                    ax_test.add_patch(patch)
290: (8)                    patch = mpatches.PathPatch(path,
291: (35)                                              linewidth=3, linestyle='-',
292: (35)                                              facecolor=(1, 0, 0),
293: (35)                                              edgecolor='none')
294: (8)                    ax_ref.add_patch(patch)
295: (4)                ax_test.set_xlim([-1, i + 1])
296: (4)                ax_test.set_ylim([-1, i + 1])
```

```
297: (4)                ax_ref.set_xlim([-1, i + 1])
298: (4)                ax_ref.set_ylim([-1, i + 1])
299: (0)            def test_wedge_movement():
300: (4)                param_dict = {'center': ((0, 0), (1, 1), 'set_center'),
301: (18)                              'r': (5, 8, 'set_radius'),
302: (18)                              'width': (2, 3, 'set_width'),
303: (18)                              'theta1': (0, 30, 'set_theta1'),
304: (18)                              'theta2': (45, 50, 'set_theta2')}
305: (4)                init_args = {k: v[0] for k, v in param_dict.items()}
306: (4)                w = mpatches.Wedge(**init_args)
307: (4)                for attr, (old_v, new_v, func) in param_dict.items():
308: (8)                    assert getattr(w, attr) == old_v
309: (8)                    getattr(w, func)(new_v)
310: (8)                    assert getattr(w, attr) == new_v
311: (0)            @image_comparison(['wedge_range'], remove_text=True,
312: (18)                              tol=0.009 if platform.machine() == 'arm64' else 0)
313: (0)            def test_wedge_range():
314: (4)                ax = plt.axes()
315: (4)                t1 = 2.313869244286224
316: (4)                args = [[52.31386924, 232.31386924],
317: (12)                       [52.313869244286224, 232.31386924428622],
318: (12)                       [t1, t1 + 180.0],
319: (12)                       [0, 360],
320: (12)                       [90, 90 + 360],
321: (12)                       [-180, 180],
322: (12)                       [0, 380],
323: (12)                       [45, 46],
324: (12)                       [46, 45]]
325: (4)                for i, (theta1, theta2) in enumerate(args):
326: (8)                    x = i % 3
327: (8)                    y = i // 3
328: (8)                    wedge = mpatches.Wedge((x * 3, y * 3), 1, theta1, theta2,
329: (31)                                           facecolor='none', edgecolor='k', lw=3)
330: (8)                    ax.add_artist(wedge)
331: (4)                ax.set_xlim(-2, 8)
332: (4)                ax.set_ylim(-2, 9)
333: (0)            def test_patch_str():
334: (4)                """
335: (4)                Check that patches have nice and working `str` representation.
336: (4)                Note that the logic is that `__str__` is defined such that:
337: (4)                str(eval(str(p))) == str(p)
338: (4)                """
339: (4)                p = mpatches.Circle(xy=(1, 2), radius=3)
340: (4)                assert str(p) == 'Circle(xy=(1, 2), radius=3)'
341: (4)                p = mpatches.Ellipse(xy=(1, 2), width=3, height=4, angle=5)
342: (4)                assert str(p) == 'Ellipse(xy=(1, 2), width=3, height=4, angle=5)'
343: (4)                p = mpatches.Rectangle(xy=(1, 2), width=3, height=4, angle=5)
344: (4)                assert str(p) == 'Rectangle(xy=(1, 2), width=3, height=4, angle=5)'
345: (4)                p = mpatches.Wedge(center=(1, 2), r=3, theta1=4, theta2=5, width=6)
346: (4)                assert str(p) == 'Wedge(center=(1, 2), r=3, theta1=4, theta2=5, width=6)'
347: (4)                p = mpatches.Arc(xy=(1, 2), width=3, height=4, angle=5, theta1=6,
theta2=7)
348: (4)                expected = 'Arc(xy=(1, 2), width=3, height=4, angle=5, theta1=6,
theta2=7)'
349: (4)                assert str(p) == expected
350: (4)                p = mpatches.Annulus(xy=(1, 2), r=(3, 4), width=1, angle=2)
351: (4)                expected = "Annulus(xy=(1, 2), r=(3, 4), width=1, angle=2)"
352: (4)                assert str(p) == expected
353: (4)                p = mpatches.RegularPolygon((1, 2), 20, radius=5)
354: (4)                assert str(p) == "RegularPolygon((1, 2), 20, radius=5, orientation=0)"
355: (4)                p = mpatches.CirclePolygon(xy=(1, 2), radius=5, resolution=20)
356: (4)                assert str(p) == "CirclePolygon((1, 2), radius=5, resolution=20)"
357: (4)                p = mpatches.FancyBboxPatch((1, 2), width=3, height=4)
358: (4)                assert str(p) == "FancyBboxPatch((1, 2), width=3, height=4)"
359: (4)                path = mpath.Path([(1, 2), (2, 2), (1, 2)], closed=True)
360: (4)                p = mpatches.PathPatch(path)
361: (4)                assert str(p) == "PathPatch3((1, 2) ...)"
362: (4)                p = mpatches.Polygon(np.empty((0, 2)))
363: (4)                assert str(p) == "Polygon0()"
```

```
(4)                 data = [[1, 2], [2, 2], [1, 2]]
(4)                 p = mpatches.Polygon(data)
(4)                 assert str(p) == "Polygon3((1, 2) ...)"
(4)                 p = mpatches.FancyArrowPatch(path=path)
(4)                 assert str(p)[:27] == "FancyArrowPatch(Path(array("
(4)                 p = mpatches.FancyArrowPatch((1, 2), (3, 4))
(4)                 assert str(p) == "FancyArrowPatch((1, 2)->(3, 4))"
(4)                 p = mpatches.ConnectionPatch((1, 2), (3, 4), 'data')
(4)                 assert str(p) == "ConnectionPatch((1, 2), (3, 4))"
(4)                 s = mpatches.Shadow(p, 1, 1)
(4)                 assert str(s) == "Shadow(ConnectionPatch((1, 2), (3, 4)))"
(0)             @image_comparison(['multi_color_hatch'], remove_text=True, style='default')
(0)             def test_multi_color_hatch():
(4)                 fig, ax = plt.subplots()
(4)                 rects = ax.bar(range(5), range(1, 6))
(4)                 for i, rect in enumerate(rects):
(8)                     rect.set_facecolor('none')
(8)                     rect.set_edgecolor(f'C{i}')
(8)                     rect.set_hatch('/')
(4)                 ax.autoscale_view()
(4)                 ax.autoscale(False)
(4)                 for i in range(5):
(8)                     with mpl.style.context({'hatch.color': f'C{i}'}):
(12)                        r = Rectangle((i - .8 / 2, 5), .8, 1, hatch='//', fc='none')
(8)                     ax.add_patch(r)
(0)             @image_comparison(['units_rectangle.png'])
(0)             def test_units_rectangle():
(4)                 import matplotlib.testing.jpl_units as U
(4)                 U.register()
(4)                 p = mpatches.Rectangle((5*U.km, 6*U.km), 1*U.km, 2*U.km)
(4)                 fig, ax = plt.subplots()
(4)                 ax.add_patch(p)
(4)                 ax.set_xlim([4*U.km, 7*U.km])
(4)                 ax.set_ylim([5*U.km, 9*U.km])
(0)             @image_comparison(['connection_patch.png'], style='mpl20', remove_text=True,
(18)                            tol=0.024 if platform.machine() == 'arm64' else 0)
(0)             def test_connection_patch():
(4)                 fig, (ax1, ax2) = plt.subplots(1, 2)
(4)                 con = mpatches.ConnectionPatch(xyA=(0.1, 0.1), xyB=(0.9, 0.9),
(35)                                               coordsA='data', coordsB='data',
(35)                                               axesA=ax2, axesB=ax1,
(35)                                               arrowstyle="->")
(4)                 ax2.add_artist(con)
(4)                 xyA = (0.6, 1.0)  # in axes coordinates
(4)                 xyB = (0.0, 0.2)  # x in axes coordinates, y in data coordinates
(4)                 coordsA = "axes fraction"
(4)                 coordsB = ax2.get_yaxis_transform()
(4)                 con = mpatches.ConnectionPatch(xyA=xyA, xyB=xyB, coordsA=coordsA,
(35)                                               coordsB=coordsB, arrowstyle="-")
(4)                 ax2.add_artist(con)
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_connection_patch_fig(fig_test, fig_ref):
(4)                 ax1, ax2 = fig_test.subplots(1, 2)
(4)                 con = mpatches.ConnectionPatch(
(8)                     xyA=(.3, .2), coordsA="data", axesA=ax1,
(8)                     xyB=(-30, -20), coordsB="figure pixels",
(8)                     arrowstyle="->", shrinkB=5)
(4)                 fig_test.add_artist(con)
(4)                 ax1, ax2 = fig_ref.subplots(1, 2)
(4)                 bb = fig_ref.bbox
(4)                 plt.rcParams["savefig.dpi"] = plt.rcParams["figure.dpi"]
(4)                 con = mpatches.ConnectionPatch(
(8)                     xyA=(.3, .2), coordsA="data", axesA=ax1,
(8)                     xyB=(bb.width - 30, bb.height - 20), coordsB="figure pixels",
(8)                     arrowstyle="->", shrinkB=5)
(4)                 fig_ref.add_artist(con)
(0)             def test_datetime_rectangle():
(4)                 from datetime import datetime, timedelta
(4)                 start = datetime(2017, 1, 1, 0, 0, 0)
```

```
433: (4)                delta = timedelta(seconds=16)
434: (4)                patch = mpatches.Rectangle((start, 0), delta, 1)
435: (4)                fig, ax = plt.subplots()
436: (4)                ax.add_patch(patch)
437: (0)            def test_datetime_datetime_fails():
438: (4)                from datetime import datetime
439: (4)                start = datetime(2017, 1, 1, 0, 0, 0)
440: (4)                dt_delta = datetime(1970, 1, 5)  # Will be 5 days if units are done wrong.
441: (4)                with pytest.raises(TypeError):
442: (8)                    mpatches.Rectangle((start, 0), dt_delta, 1)
443: (4)                with pytest.raises(TypeError):
444: (8)                    mpatches.Rectangle((0, start), 1, dt_delta)
445: (0)            def test_contains_point():
446: (4)                ell = mpatches.Ellipse((0.5, 0.5), 0.5, 1.0)
447: (4)                points = [(0.0, 0.5), (0.2, 0.5), (0.25, 0.5), (0.5, 0.5)]
448: (4)                path = ell.get_path()
449: (4)                transform = ell.get_transform()
450: (4)                radius = ell._process_radius(None)
451: (4)                expected = np.array([path.contains_point(point,
452: (45)                                                         transform,
453: (45)                                                         radius) for point in points])
454: (4)                result = np.array([ell.contains_point(point) for point in points])
455: (4)                assert np.all(result == expected)
456: (0)            def test_contains_points():
457: (4)                ell = mpatches.Ellipse((0.5, 0.5), 0.5, 1.0)
458: (4)                points = [(0.0, 0.5), (0.2, 0.5), (0.25, 0.5), (0.5, 0.5)]
459: (4)                path = ell.get_path()
460: (4)                transform = ell.get_transform()
461: (4)                radius = ell._process_radius(None)
462: (4)                expected = path.contains_points(points, transform, radius)
463: (4)                result = ell.contains_points(points)
464: (4)                assert np.all(result == expected)
465: (0)            @check_figures_equal(extensions=["png"])
466: (0)            def test_shadow(fig_test, fig_ref):
467: (4)                xy = np.array([.2, .3])
468: (4)                dxy = np.array([.1, .2])
469: (4)                plt.rcParams["savefig.dpi"] = "figure"
470: (4)                a1 = fig_test.subplots()
471: (4)                rect = mpatches.Rectangle(xy=xy, width=.5, height=.5)
472: (4)                shadow = mpatches.Shadow(rect, ox=dxy[0], oy=dxy[1])
473: (4)                a1.add_patch(rect)
474: (4)                a1.add_patch(shadow)
475: (4)                a2 = fig_ref.subplots()
476: (4)                rect = mpatches.Rectangle(xy=xy, width=.5, height=.5)
477: (4)                shadow = mpatches.Rectangle(
478: (8)                    xy=xy + fig_ref.dpi / 72 * dxy, width=.5, height=.5,
479: (8)                    fc=np.asarray(mcolors.to_rgb(rect.get_facecolor())) * .3,
480: (8)                    ec=np.asarray(mcolors.to_rgb(rect.get_facecolor())) * .3,
481: (8)                    alpha=.5)
482: (4)                a2.add_patch(shadow)
483: (4)                a2.add_patch(rect)
484: (0)            def test_fancyarrow_units():
485: (4)                from datetime import datetime
486: (4)                dtime = datetime(2000, 1, 1)
487: (4)                fig, ax = plt.subplots()
488: (4)                arrow = FancyArrowPatch((0, dtime), (0.01, dtime))
489: (0)            def test_fancyarrow_setdata():
490: (4)                fig, ax = plt.subplots()
491: (4)                arrow = ax.arrow(0, 0, 10, 10, head_length=5, head_width=1, width=.5)
492: (4)                expected1 = np.array(
493: (6)                  [[13.54, 13.54],
494: (7)                   [10.35,  9.65],
495: (7)                   [10.18,  9.82],
496: (7)                   [0.18, -0.18],
497: (7)                   [-0.18,  0.18],
498: (7)                   [9.82, 10.18],
499: (7)                   [9.65, 10.35],
500: (7)                   [13.54, 13.54]]
501: (4)                )
```

```
(4)                 assert np.allclose(expected1, np.round(arrow.verts, 2))
(4)                 expected2 = np.array(
(6)                   [[16.71, 16.71],
(7)                    [16.71, 15.29],
(7)                    [16.71, 15.29],
(7)                    [1.71,  0.29],
(7)                    [0.29,  1.71],
(7)                    [15.29, 16.71],
(7)                    [15.29, 16.71],
(7)                    [16.71, 16.71]]
(4)                 )
(4)                 arrow.set_data(
(8)                     x=1, y=1, dx=15, dy=15, width=2, head_width=2, head_length=1
(4)                 )
(4)                 assert np.allclose(expected2, np.round(arrow.verts, 2))
(0)             @image_comparison(["large_arc.svg"], style="mpl20")
(0)             def test_large_arc():
(4)                 fig, (ax1, ax2) = plt.subplots(1, 2)
(4)                 x = 210
(4)                 y = -2115
(4)                 diameter = 4261
(4)                 for ax in [ax1, ax2]:
(8)                     a = Arc((x, y), diameter, diameter, lw=2, color='k')
(8)                     ax.add_patch(a)
(8)                     ax.set_axis_off()
(8)                     ax.set_aspect('equal')
(4)                 ax1.set_xlim(7, 8)
(4)                 ax1.set_ylim(5, 6)
(4)                 ax2.set_xlim(-25000, 18000)
(4)                 ax2.set_ylim(-20000, 6600)
(0)             @image_comparison(["all_quadrants_arcs.svg"], style="mpl20")
(0)             def test_rotated_arcs():
(4)                 fig, ax_arr = plt.subplots(2, 2, squeeze=False, figsize=(10, 10))
(4)                 scale = 10_000_000
(4)                 diag_centers = ((-1, -1), (-1, 1), (1, 1), (1, -1))
(4)                 on_axis_centers = ((0, 1), (1, 0), (0, -1), (-1, 0))
(4)                 skews = ((2, 2), (2, 1/10), (2,  1/100), (2, 1/1000))
(4)                 for ax, (sx, sy) in zip(ax_arr.ravel(), skews):
(8)                     k = 0
(8)                     for prescale, centers in zip((1 - .0001, (1 - .0001) / np.sqrt(2)),
(38)                                                 (on_axis_centers, diag_centers)):
(12)                        for j, (x_sign, y_sign) in enumerate(centers, start=k):
(16)                            a = Arc(
(20)                                (x_sign * scale * prescale,
(21)                                 y_sign * scale * prescale),
(20)                                scale * sx,
(20)                                scale * sy,
(20)                                lw=4,
(20)                                color=f"C{j}",
(20)                                zorder=1 + j,
(20)                                angle=np.rad2deg(np.arctan2(y_sign, x_sign)) % 360,
(20)                                label=f'big {j}',
(20)                                gid=f'big {j}'
(16)                            )
(16)                            ax.add_patch(a)
(12)                        k = j+1
(8)                     ax.set_xlim(-scale / 4000, scale / 4000)
(8)                     ax.set_ylim(-scale / 4000, scale / 4000)
(8)                     ax.axhline(0, color="k")
(8)                     ax.axvline(0, color="k")
(8)                     ax.set_axis_off()
(8)                     ax.set_aspect("equal")
(0)             def test_fancyarrow_shape_error():
(4)                 with pytest.raises(ValueError, match="Got unknown shape: 'foo'"):
(8)                     FancyArrow(0, 0, 0.2, 0.2, shape='foo')
(0)             @pytest.mark.parametrize('fmt, match', (
(4)                 ("foo", "Unknown style: 'foo'"),
(4)                 ("Round,foo", "Incorrect style argument: 'Round,foo'"),
(0)             ))
```

```
(0)             def test_boxstyle_errors(fmt, match):
(4)                 with pytest.raises(ValueError, match=match):
(8)                     BoxStyle(fmt)
(0)             @image_comparison(baseline_images=['annulus'], extensions=['png'])
(0)             def test_annulus():
(4)                 fig, ax = plt.subplots()
(4)                 cir = Annulus((0.5, 0.5), 0.2, 0.05, fc='g')        # circular annulus
(4)                 ell = Annulus((0.5, 0.5), (0.5, 0.3), 0.1, 45,      # elliptical
(18)                             fc='m', ec='b', alpha=0.5, hatch='xxx')
(4)                 ax.add_patch(cir)
(4)                 ax.add_patch(ell)
(4)                 ax.set_aspect('equal')
(0)             @image_comparison(baseline_images=['annulus'], extensions=['png'])
(0)             def test_annulus_setters():
(4)                 fig, ax = plt.subplots()
(4)                 cir = Annulus((0., 0.), 0.2, 0.01, fc='g')   # circular annulus
(4)                 ell = Annulus((0., 0.), (1, 2), 0.1, 0,      # elliptical
(18)                             fc='m', ec='b', alpha=0.5, hatch='xxx')
(4)                 ax.add_patch(cir)
(4)                 ax.add_patch(ell)
(4)                 ax.set_aspect('equal')
(4)                 cir.center = (0.5, 0.5)
(4)                 cir.radii = 0.2
(4)                 cir.width = 0.05
(4)                 ell.center = (0.5, 0.5)
(4)                 ell.radii = (0.5, 0.3)
(4)                 ell.width = 0.1
(4)                 ell.angle = 45
(0)             @image_comparison(baseline_images=['annulus'], extensions=['png'])
(0)             def test_annulus_setters2():
(4)                 fig, ax = plt.subplots()
(4)                 cir = Annulus((0., 0.), 0.2, 0.01, fc='g')   # circular annulus
(4)                 ell = Annulus((0., 0.), (1, 2), 0.1, 0,      # elliptical
(18)                             fc='m', ec='b', alpha=0.5, hatch='xxx')
(4)                 ax.add_patch(cir)
(4)                 ax.add_patch(ell)
(4)                 ax.set_aspect('equal')
(4)                 cir.center = (0.5, 0.5)
(4)                 cir.set_semimajor(0.2)
(4)                 cir.set_semiminor(0.2)
(4)                 assert cir.radii == (0.2, 0.2)
(4)                 cir.width = 0.05
(4)                 ell.center = (0.5, 0.5)
(4)                 ell.set_semimajor(0.5)
(4)                 ell.set_semiminor(0.3)
(4)                 assert ell.radii == (0.5, 0.3)
(4)                 ell.width = 0.1
(4)                 ell.angle = 45
(0)             def test_degenerate_polygon():
(4)                 point = [0, 0]
(4)                 correct_extents = Bbox([point, point]).extents
(4)                 assert np.all(Polygon([point]).get_extents().extents == correct_extents)
(0)             @pytest.mark.parametrize('kwarg', ('edgecolor', 'facecolor'))
(0)             def test_color_override_warning(kwarg):
(4)                 with pytest.warns(UserWarning,
(22)                                  match="Setting the 'color' property will override "
(28)                                        "the edgecolor or facecolor properties."):
(8)                     Patch(color='black', **{kwarg: 'black'})
(0)             def test_empty_verts():
(4)                 poly = Polygon(np.zeros((0, 2)))
(4)                 assert poly.get_verts() == []
(0)             def test_default_antialiased():
(4)                 patch = Patch()
(4)                 patch.set_antialiased(not rcParams['patch.antialiased'])
(4)                 assert patch.get_antialiased() == (not rcParams['patch.antialiased'])
(4)                 patch.set_antialiased(None)
(4)                 assert patch.get_antialiased() == rcParams['patch.antialiased']
(0)             def test_default_linestyle():
(4)                 patch = Patch()
```

```
(4)                 patch.set_linestyle('--')
(4)                 patch.set_linestyle(None)
(4)                 assert patch.get_linestyle() == 'solid'
(0)             def test_default_capstyle():
(4)                 patch = Patch()
(4)                 assert patch.get_capstyle() == 'butt'
(0)             def test_default_joinstyle():
(4)                 patch = Patch()
(4)                 assert patch.get_joinstyle() == 'miter'
(0)             @image_comparison(["autoscale_arc"], extensions=['png', 'svg'],
(18)                              style="mpl20", remove_text=True)
(0)             def test_autoscale_arc():
(4)                 fig, axs = plt.subplots(1, 3, figsize=(4, 1))
(4)                 arc_lists = (
(8)                     [Arc((0, 0), 1, 1, theta1=0, theta2=90)],
(8)                     [Arc((0.5, 0.5), 1.5, 0.5, theta1=10, theta2=20)],
(8)                     [Arc((0.5, 0.5), 1.5, 0.5, theta1=10, theta2=20),
(9)                      Arc((0.5, 0.5), 2.5, 0.5, theta1=110, theta2=120),
(9)                      Arc((0.5, 0.5), 3.5, 0.5, theta1=210, theta2=220),
(9)                      Arc((0.5, 0.5), 4.5, 0.5, theta1=310, theta2=320)])
(4)                 for ax, arcs in zip(axs, arc_lists):
(8)                     for arc in arcs:
(12)                        ax.add_patch(arc)
(8)                     ax.autoscale()
(0)             @check_figures_equal(extensions=["png", 'svg', 'pdf', 'eps'])
(0)             def test_arc_in_collection(fig_test, fig_ref):
(4)                 arc1 = Arc([.5, .5], .5, 1, theta1=0, theta2=60, angle=20)
(4)                 arc2 = Arc([.5, .5], .5, 1, theta1=0, theta2=60, angle=20)
(4)                 col = mcollections.PatchCollection(patches=[arc2], facecolors='none',
(39)                                                   edgecolors='k')
(4)                 fig_ref.subplots().add_patch(arc1)
(4)                 fig_test.subplots().add_collection(col)
(0)             @check_figures_equal(extensions=["png", 'svg', 'pdf', 'eps'])
(0)             def test_modifying_arc(fig_test, fig_ref):
(4)                 arc1 = Arc([.5, .5], .5, 1, theta1=0, theta2=60, angle=20)
(4)                 arc2 = Arc([.5, .5], 1.5, 1, theta1=0, theta2=60, angle=10)
(4)                 fig_ref.subplots().add_patch(arc1)
(4)                 fig_test.subplots().add_patch(arc2)
(4)                 arc2.set_width(.5)
(4)                 arc2.set_angle(20)
(0)             def test_arrow_set_data():
(4)                 fig, ax = plt.subplots()
(4)                 arrow = mpl.patches.Arrow(2, 0, 0, 10)
(4)                 expected1 = np.array(
(7)                    [[1.9,  0.],
(8)                     [2.1, -0.],
(8)                     [2.1, 8.],
(8)                     [2.3, 8.],
(8)                     [2., 10.],
(8)                     [1.7, 8.],
(8)                     [1.9, 8.],
(8)                     [1.9, 0.]]
(4)                 )
(4)                 assert np.allclose(expected1, np.round(arrow.get_verts(), 2))
(4)                 expected2 = np.array(
(8)                     [[0.39, 0.04],
(9)                      [0.61, -0.04],
(9)                      [3.01, 6.36],
(9)                      [3.24, 6.27],
(9)                      [3.5, 8.],
(9)                      [2.56, 6.53],
(9)                      [2.79, 6.44],
(9)                      [0.39, 0.04]]
(4)                 )
(4)                 arrow.set_data(x=.5, dx=3, dy=8, width=1.2)
(4)                 assert np.allclose(expected2, np.round(arrow.get_verts(), 2))
```

File 199 - test_patheffects.py:

```
1: (0)              import platform
2: (0)              import numpy as np
3: (0)              from matplotlib.testing.decorators import image_comparison
4: (0)              import matplotlib.pyplot as plt
5: (0)              import matplotlib.patheffects as path_effects
6: (0)              from matplotlib.path import Path
7: (0)              import matplotlib.patches as patches
8: (0)              from matplotlib.backend_bases import RendererBase
9: (0)              from matplotlib.patheffects import PathEffectRenderer
10: (0)             @image_comparison(['patheffect1'], remove_text=True)
11: (0)             def test_patheffect1():
12: (4)                 ax1 = plt.subplot()
13: (4)                 ax1.imshow([[1, 2], [2, 3]])
14: (4)                 txt = ax1.annotate("test", (1., 1.), (0., 0),
15: (23)                                   arrowprops=dict(arrowstyle="->",
16: (39)                                                   connectionstyle="angle3", lw=2),
17: (23)                                   size=20, ha="center",
18: (23)                                   path_effects=[path_effects.withStroke(linewidth=3,
19: (61)                                                                         foreground="w")])
20: (4)                 txt.arrow_patch.set_path_effects([path_effects.Stroke(linewidth=5,
21: (58)                                                                       foreground="w"),
22: (38)                                                  path_effects.Normal()])
23: (4)                 pe = [path_effects.withStroke(linewidth=3, foreground="w")]
24: (4)                 ax1.grid(True, linestyle="-", path_effects=pe)
25: (0)             @image_comparison(['patheffect2'], remove_text=True, style='mpl20',
26: (18)                              tol=0.06 if platform.machine() == 'arm64' else 0)
27: (0)             def test_patheffect2():
28: (4)                 ax2 = plt.subplot()
29: (4)                 arr = np.arange(25).reshape((5, 5))
30: (4)                 ax2.imshow(arr, interpolation='nearest')
31: (4)                 cntr = ax2.contour(arr, colors="k")
32: (4)                 cntr.set(path_effects=[path_effects.withStroke(linewidth=3,
foreground="w")])
33: (4)                 clbls = ax2.clabel(cntr, fmt="%2.0f", use_clabeltext=True)
34: (4)                 plt.setp(clbls,
35: (13)                         path_effects=[path_effects.withStroke(linewidth=3,
36: (51)                                                               foreground="w")])
37: (0)             @image_comparison(['patheffect3'], tol=0.019 if platform.machine() == 'arm64'
else 0)
38: (0)             def test_patheffect3():
39: (4)                 p1, = plt.plot([1, 3, 5, 4, 3], 'o-b', lw=4)
40: (4)                 p1.set_path_effects([path_effects.SimpleLineShadow(),
41: (25)                                     path_effects.Normal()])
42: (4)                 plt.title(
43: (8)                     r'testing$^{123}$',
44: (8)                     path_effects=[path_effects.withStroke(linewidth=1, foreground="r")])
45: (4)                 leg = plt.legend([p1], [r'Line 1$^2$'], fancybox=True, loc='upper left')
46: (4)                 leg.legendPatch.set_path_effects([path_effects.withSimplePatchShadow()])
47: (4)                 text = plt.text(2, 3, 'Drop test', color='white',
48: (20)                                bbox={'boxstyle': 'circle,pad=0.1', 'color': 'red'})
49: (4)                 pe = [path_effects.Stroke(linewidth=3.75, foreground='k'),
50: (10)                      path_effects.withSimplePatchShadow((6, -3), shadow_rgbFace='blue')]
51: (4)                 text.set_path_effects(pe)
52: (4)                 text.get_bbox_patch().set_path_effects(pe)
53: (4)                 pe = [path_effects.PathPatchEffect(offset=(4, -4), hatch='xxxx',
54: (39)                                                   facecolor='gray'),
55: (10)                      path_effects.PathPatchEffect(edgecolor='white', facecolor='black',
56: (39)                                                   lw=1.1)]
57: (4)                 t = plt.gcf().text(0.02, 0.1, 'Hatch shadow', fontsize=75, weight=1000,
58: (23)                                   va='center')
59: (4)                 t.set_path_effects(pe)
60: (0)             @image_comparison(['stroked_text.png'])
61: (0)             def test_patheffects_stroked_text():
62: (4)                 text_chunks = [
63: (8)                     'A B C D E F G H I J K L',
64: (8)                     'M N O P Q R S T U V W',
65: (8)                     'X Y Z a b c d e f g h i j',
```

```
 66: (8)                    'k l m n o p q r s t u v',
 67: (8)                    'w x y z 0123456789',
 68: (8)                    r"!@#$%^&*()-=_+[]\;'",
 69: (8)                    ',./{}|:"<>?'
 70: (4)                ]
 71: (4)                font_size = 50
 72: (4)                ax = plt.axes((0, 0, 1, 1))
 73: (4)                for i, chunk in enumerate(text_chunks):
 74: (8)                    text = ax.text(x=0.01, y=(0.9 - i * 0.13), s=chunk,
 75: (23)                                   fontdict={'ha': 'left', 'va': 'center',
 76: (33)                                             'size': font_size, 'color': 'white'})
 77: (8)                    text.set_path_effects([path_effects.Stroke(linewidth=font_size / 10,
 78: (51)                                                              foreground='black'),
 79: (31)                                          path_effects.Normal()])
 80: (4)                ax.set_xlim(0, 1)
 81: (4)                ax.set_ylim(0, 1)
 82: (4)                ax.axis('off')
 83: (0)            def test_PathEffect_points_to_pixels():
 84: (4)                fig = plt.figure(dpi=150)
 85: (4)                p1, = plt.plot(range(10))
 86: (4)                p1.set_path_effects([path_effects.SimpleLineShadow(),
 87: (25)                                     path_effects.Normal()])
 88: (4)                renderer = fig.canvas.get_renderer()
 89: (4)                pe_renderer = path_effects.PathEffectRenderer(
 90: (8)                    p1.get_path_effects(), renderer)
 91: (4)                assert renderer.points_to_pixels(15) == pe_renderer.points_to_pixels(15)
 92: (0)            def test_SimplePatchShadow_offset():
 93: (4)                pe = path_effects.SimplePatchShadow(offset=(4, 5))
 94: (4)                assert pe._offset == (4, 5)
 95: (0)            @image_comparison(['collection'], tol=0.03, style='mpl20')
 96: (0)            def test_collection():
 97: (4)                x, y = np.meshgrid(np.linspace(0, 10, 150), np.linspace(-5, 5, 100))
 98: (4)                data = np.sin(x) + np.cos(y)
 99: (4)                cs = plt.contour(data)
100: (4)                cs.set(path_effects=[
101: (8)                    path_effects.PathPatchEffect(edgecolor='black', facecolor='none',
linewidth=12),
102: (8)                    path_effects.Stroke(linewidth=5)])
103: (4)                for text in plt.clabel(cs, colors='white'):
104: (8)                    text.set_path_effects([path_effects.withStroke(foreground='k',
105: (55)                                                                  linewidth=3)])
106: (8)                    text.set_bbox({'boxstyle': 'sawtooth', 'facecolor': 'none',
107: (23)                                   'edgecolor': 'blue'})
108: (0)            @image_comparison(['tickedstroke'], remove_text=True, extensions=['png'],
109: (18)                             tol=0.22)  # Increased tolerance due to fixed clipping.
110: (0)            def test_tickedstroke():
111: (4)                fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 4))
112: (4)                path = Path.unit_circle()
113: (4)                patch = patches.PathPatch(path, facecolor='none', lw=2, path_effects=[
114: (8)                    path_effects.withTickedStroke(angle=-90, spacing=10,
115: (38)                                                 length=1)])
116: (4)                ax1.add_patch(patch)
117: (4)                ax1.axis('equal')
118: (4)                ax1.set_xlim(-2, 2)
119: (4)                ax1.set_ylim(-2, 2)
120: (4)                ax2.plot([0, 1], [0, 1], label=' ',
121: (13)                         path_effects=[path_effects.withTickedStroke(spacing=7,
122: (57)                                                                     angle=135)])
123: (4)                nx = 101
124: (4)                x = np.linspace(0.0, 1.0, nx)
125: (4)                y = 0.3 * np.sin(x * 8) + 0.4
126: (4)                ax2.plot(x, y, label=' ', path_effects=[path_effects.withTickedStroke()])
127: (4)                ax2.legend()
128: (4)                nx = 101
129: (4)                ny = 105
130: (4)                xvec = np.linspace(0.001, 4.0, nx)
131: (4)                yvec = np.linspace(0.001, 4.0, ny)
132: (4)                x1, x2 = np.meshgrid(xvec, yvec)
133: (4)                g1 = -(3 * x1 + x2 - 5.5)
```

```
134: (4)                g2 = -(x1 + 2 * x2 - 4)
135: (4)                g3 = .8 + x1 ** -3 - x2
136: (4)                cg1 = ax3.contour(x1, x2, g1, [0], colors=('k',))
137: (4)                cg1.set(path_effects=[path_effects.withTickedStroke(angle=135)])
138: (4)                cg2 = ax3.contour(x1, x2, g2, [0], colors=('r',))
139: (4)                cg2.set(path_effects=[path_effects.withTickedStroke(angle=60, length=2)])
140: (4)                cg3 = ax3.contour(x1, x2, g3, [0], colors=('b',))
141: (4)                cg3.set(path_effects=[path_effects.withTickedStroke(spacing=7)])
142: (4)                ax3.set_xlim(0, 4)
143: (4)                ax3.set_ylim(0, 4)
144: (0)            @image_comparison(['spaces_and_newlines.png'], remove_text=True)
145: (0)            def test_patheffects_spaces_and_newlines():
146: (4)                ax = plt.subplot()
147: (4)                s1 = "         "
148: (4)                s2 = "\nNewline also causes problems"
149: (4)                text1 = ax.text(0.5, 0.75, s1, ha='center', va='center', size=20,
150: (20)                               bbox={'color': 'salmon'})
151: (4)                text2 = ax.text(0.5, 0.25, s2, ha='center', va='center', size=20,
152: (20)                               bbox={'color': 'thistle'})
153: (4)                text1.set_path_effects([path_effects.Normal()])
154: (4)                text2.set_path_effects([path_effects.Normal()])
155: (0)            def test_patheffects_overridden_methods_open_close_group():
156: (4)                class CustomRenderer(RendererBase):
157: (8)                    def __init__(self):
158: (12)                       super().__init__()
159: (8)                    def open_group(self, s, gid=None):
160: (12)                       return "open_group overridden"
161: (8)                    def close_group(self, s):
162: (12)                       return "close_group overridden"
163: (4)                renderer = PathEffectRenderer([path_effects.Normal()], CustomRenderer())
164: (4)                assert renderer.open_group('s') == "open_group overridden"
165: (4)                assert renderer.close_group('s') == "close_group overridden"
```

File 200 - test_path.py:

```
1: (0)              import platform
2: (0)              import re
3: (0)              import numpy as np
4: (0)              from numpy.testing import assert_array_equal
5: (0)              import pytest
6: (0)              from matplotlib import patches
7: (0)              from matplotlib.path import Path
8: (0)              from matplotlib.patches import Polygon
9: (0)              from matplotlib.testing.decorators import image_comparison
10: (0)             import matplotlib.pyplot as plt
11: (0)             from matplotlib import transforms
12: (0)             from matplotlib.backend_bases import MouseEvent
13: (0)             def test_empty_closed_path():
14: (4)                 path = Path(np.zeros((0, 2)), closed=True)
15: (4)                 assert path.vertices.shape == (0, 2)
16: (4)                 assert path.codes is None
17: (4)                 assert_array_equal(path.get_extents().extents,
18: (23)                                   transforms.Bbox.null().extents)
19: (0)             def test_readonly_path():
20: (4)                 path = Path.unit_circle()
21: (4)                 def modify_vertices():
22: (8)                     path.vertices = path.vertices * 2.0
23: (4)                 with pytest.raises(AttributeError):
24: (8)                     modify_vertices()
25: (0)             def test_path_exceptions():
26: (4)                 bad_verts1 = np.arange(12).reshape(4, 3)
27: (4)                 with pytest.raises(ValueError,
28: (23)                                   match=re.escape(f'has shape {bad_verts1.shape}')):
29: (8)                     Path(bad_verts1)
30: (4)                 bad_verts2 = np.arange(12).reshape(2, 3, 2)
31: (4)                 with pytest.raises(ValueError,
32: (23)                                   match=re.escape(f'has shape {bad_verts2.shape}')):
```

```
 33: (8)                    Path(bad_verts2)
 34: (4)                good_verts = np.arange(12).reshape(6, 2)
 35: (4)                bad_codes = np.arange(2)
 36: (4)                msg = re.escape(f"Your vertices have shape {good_verts.shape} "
 37: (20)                               f"but your codes have shape {bad_codes.shape}")
 38: (4)                with pytest.raises(ValueError, match=msg):
 39: (8)                    Path(good_verts, bad_codes)
 40: (0)            def test_point_in_path():
 41: (4)                path = Path._create_closed([(0, 0), (0, 1), (1, 1), (1, 0)])
 42: (4)                points = [(0.5, 0.5), (1.5, 0.5)]
 43: (4)                ret = path.contains_points(points)
 44: (4)                assert ret.dtype == 'bool'
 45: (4)                np.testing.assert_equal(ret, [True, False])
 46: (0)            @pytest.mark.parametrize(
 47: (4)                "other_path, inside, inverted_inside",
 48: (4)                [(Path([(0.25, 0.25), (0.25, 0.75), (0.75, 0.75), (0.75, 0.25), (0.25,
 0.25)],
 49: (11)                      closed=True), True, False),
 50: (5)                 (Path([(-0.25, -0.25), (-0.25, 1.75), (1.75, 1.75), (1.75, -0.25),
 (-0.25, -0.25)],
 51: (11)                      closed=True), False, True),
 52: (5)                 (Path([(-0.25, -0.25), (-0.25, 1.75), (0.5, 0.5),
 53: (12)                       (1.75, 1.75), (1.75, -0.25), (-0.25, -0.25)],
 54: (11)                      closed=True), False, False),
 55: (5)                 (Path([(0.25, 0.25), (0.25, 1.25), (1.25, 1.25), (1.25, 0.25), (0.25,
 0.25)],
 56: (11)                      closed=True), False, False),
 57: (5)                 (Path([(0, 0), (0, 1), (1, 1), (1, 0), (0, 0)], closed=True), False,
 False),
 58: (5)                 (Path([(2, 2), (2, 3), (3, 3), (3, 2), (2, 2)], closed=True), False,
 False)])
 59: (0)            def test_contains_path(other_path, inside, inverted_inside):
 60: (4)                path = Path([(0, 0), (0, 1), (1, 1), (1, 0), (0, 0)], closed=True)
 61: (4)                assert path.contains_path(other_path) is inside
 62: (4)                assert other_path.contains_path(path) is inverted_inside
 63: (0)            def test_contains_points_negative_radius():
 64: (4)                path = Path.unit_circle()
 65: (4)                points = [(0.0, 0.0), (1.25, 0.0), (0.9, 0.9)]
 66: (4)                result = path.contains_points(points, radius=-0.5)
 67: (4)                np.testing.assert_equal(result, [True, False, False])
 68: (0)            _test_paths = [
 69: (4)                Path([[0, 0], [1, 0], [1, 1], [0, 1]],
 70: (11)                      [Path.MOVETO, Path.CURVE4, Path.CURVE4, Path.CURVE4]),
 71: (4)                Path([[0, 0], [0, 1], [1, 0]], [Path.MOVETO, Path.CURVE3, Path.CURVE3]),
 72: (4)                Path([[0, 1], [1, 1]], [Path.MOVETO, Path.LINETO]),
 73: (4)                Path([[1, 2]], [Path.MOVETO]),
 74: (0)            ]
 75: (0)            _test_path_extents = [(0., 0., 0.75, 1.), (0., 0., 1., 0.5), (0., 1., 1., 1.),
 76: (22)                                 (1., 2., 1., 2.)]
 77: (0)            @pytest.mark.parametrize('path, extents', zip(_test_paths,
 _test_path_extents))
 78: (0)            def test_exact_extents(path, extents):
 79: (4)                assert np.all(path.get_extents().extents == extents)
 80: (0)            @pytest.mark.parametrize('ignored_code', [Path.CLOSEPOLY, Path.STOP])
 81: (0)            def test_extents_with_ignored_codes(ignored_code):
 82: (4)                path = Path([[0, 0],
 83: (17)                            [1, 1],
 84: (17)                            [2, 2]], [Path.MOVETO, Path.MOVETO, ignored_code])
 85: (4)                assert np.all(path.get_extents().extents == (0., 0., 1., 1.))
 86: (0)            def test_point_in_path_nan():
 87: (4)                box = np.array([[0, 0], [1, 0], [1, 1], [0, 1], [0, 0]])
 88: (4)                p = Path(box)
 89: (4)                test = np.array([[np.nan, 0.5]])
 90: (4)                contains = p.contains_points(test)
 91: (4)                assert len(contains) == 1
 92: (4)                assert not contains[0]
 93: (0)            def test_nonlinear_containment():
 94: (4)                fig, ax = plt.subplots()
 95: (4)                ax.set(xscale="log", ylim=(0, 1))
```

```
 96: (4)                 polygon = ax.axvspan(1, 10)
 97: (4)                 assert polygon.get_path().contains_point(
 98: (8)                     ax.transData.transform((5, .5)), polygon.get_transform())
 99: (4)                 assert not polygon.get_path().contains_point(
100: (8)                     ax.transData.transform((.5, .5)), polygon.get_transform())
101: (4)                 assert not polygon.get_path().contains_point(
102: (8)                     ax.transData.transform((50, .5)), polygon.get_transform())
103: (0)             @image_comparison(['arrow_contains_point.png'], remove_text=True,
style='mpl20',
104: (18)                                tol=0.027 if platform.machine() == 'arm64' else 0)
105: (0)             def test_arrow_contains_point():
106: (4)                 fig, ax = plt.subplots()
107: (4)                 ax.set_xlim((0, 2))
108: (4)                 ax.set_ylim((0, 2))
109: (4)                 arrow = patches.FancyArrowPatch((0.5, 0.25), (1.5, 0.75),
110: (36)                                                 arrowstyle='->',
111: (36)                                                 mutation_scale=40)
112: (4)                 ax.add_patch(arrow)
113: (4)                 arrow1 = patches.FancyArrowPatch((0.5, 1), (1.5, 1.25),
114: (37)                                                  arrowstyle=']-[',
115: (37)                                                  mutation_scale=40)
116: (4)                 ax.add_patch(arrow1)
117: (4)                 arrow2 = patches.FancyArrowPatch((0.5, 1.5), (1.5, 1.75),
118: (37)                                                  arrowstyle='fancy',
119: (37)                                                  fill=False,
120: (37)                                                  mutation_scale=40)
121: (4)                 ax.add_patch(arrow2)
122: (4)                 patches_list = [arrow, arrow1, arrow2]
123: (4)                 X, Y = np.meshgrid(np.arange(0, 2, 0.1),
124: (23)                                   np.arange(0, 2, 0.1))
125: (4)                 for k, (x, y) in enumerate(zip(X.ravel(), Y.ravel())):
126: (8)                     xdisp, ydisp = ax.transData.transform([x, y])
127: (8)                     event = MouseEvent('button_press_event', fig.canvas, xdisp, ydisp)
128: (8)                     for m, patch in enumerate(patches_list):
129: (12)                        inside, res = patch.contains(event)
130: (12)                        if inside:
131: (16)                            ax.scatter(x, y, s=5, c="r")
132: (0)             @image_comparison(['path_clipping.svg'], remove_text=True)
133: (0)             def test_path_clipping():
134: (4)                 fig = plt.figure(figsize=(6.0, 6.2))
135: (4)                 for i, xy in enumerate([
136: (12)                        [(200, 200), (200, 350), (400, 350), (400, 200)],
137: (12)                        [(200, 200), (200, 350), (400, 350), (400, 100)],
138: (12)                        [(200, 100), (200, 350), (400, 350), (400, 100)],
139: (12)                        [(200, 100), (200, 415), (400, 350), (400, 100)],
140: (12)                        [(200, 100), (200, 415), (400, 415), (400, 100)],
141: (12)                        [(200, 415), (400, 415), (400, 100), (200, 100)],
142: (12)                        [(400, 415), (400, 100), (200, 100), (200, 415)]]):
143: (8)                     ax = fig.add_subplot(4, 2, i+1)
144: (8)                     bbox = [0, 140, 640, 260]
145: (8)                     ax.set_xlim(bbox[0], bbox[0] + bbox[2])
146: (8)                     ax.set_ylim(bbox[1], bbox[1] + bbox[3])
147: (8)                     ax.add_patch(Polygon(
148: (12)                        xy, facecolor='none', edgecolor='red', closed=True))
149: (0)             @image_comparison(['semi_log_with_zero.png'], style='mpl20')
150: (0)             def test_log_transform_with_zero():
151: (4)                 x = np.arange(-10, 10)
152: (4)                 y = (1.0 - 1.0/(x**2+1))**20
153: (4)                 fig, ax = plt.subplots()
154: (4)                 ax.semilogy(x, y, "-o", lw=15, markeredgecolor='k')
155: (4)                 ax.set_ylim(1e-7, 1)
156: (4)                 ax.grid(True)
157: (0)             def test_make_compound_path_empty():
158: (4)                 empty = Path.make_compound_path()
159: (4)                 assert empty.vertices.shape == (0, 2)
160: (4)                 r2 = Path.make_compound_path(empty, empty)
161: (4)                 assert r2.vertices.shape == (0, 2)
162: (4)                 assert r2.codes.shape == (0,)
163: (4)                 r3 = Path.make_compound_path(Path([(0, 0)]), empty)
```

```
(4)                 assert r3.vertices.shape == (1, 2)
(4)                 assert r3.codes.shape == (1,)
(0)             def test_make_compound_path_stops():
(4)                 zero = [0, 0]
(4)                 paths = 3*[Path([zero, zero], [Path.MOVETO, Path.STOP])]
(4)                 compound_path = Path.make_compound_path(*paths)
(4)                 assert np.sum(compound_path.codes == Path.STOP) == 0
(0)             @image_comparison(['xkcd.png'], remove_text=True)
(0)             def test_xkcd():
(4)                 np.random.seed(0)
(4)                 x = np.linspace(0, 2 * np.pi, 100)
(4)                 y = np.sin(x)
(4)                 with plt.xkcd():
(8)                     fig, ax = plt.subplots()
(8)                     ax.plot(x, y)
(0)             @image_comparison(['xkcd_marker.png'], remove_text=True)
(0)             def test_xkcd_marker():
(4)                 np.random.seed(0)
(4)                 x = np.linspace(0, 5, 8)
(4)                 y1 = x
(4)                 y2 = 5 - x
(4)                 y3 = 2.5 * np.ones(8)
(4)                 with plt.xkcd():
(8)                     fig, ax = plt.subplots()
(8)                     ax.plot(x, y1, '+', ms=10)
(8)                     ax.plot(x, y2, 'o', ms=10)
(8)                     ax.plot(x, y3, '^', ms=10)
(0)             @image_comparison(['marker_paths.pdf'], remove_text=True)
(0)             def test_marker_paths_pdf():
(4)                 N = 7
(4)                 plt.errorbar(np.arange(N),
(17)                             np.ones(N) + 4,
(17)                             np.ones(N))
(4)                 plt.xlim(-1, N)
(4)                 plt.ylim(-1, 7)
(0)             @image_comparison(['nan_path'], style='default', remove_text=True,
(18)                              extensions=['pdf', 'svg', 'eps', 'png'],
(18)                              tol=0.009 if platform.machine() == 'arm64' else 0)
(0)             def test_nan_isolated_points():
(4)                 y0 = [0, np.nan, 2, np.nan, 4, 5, 6]
(4)                 y1 = [np.nan, 7, np.nan, 9, 10, np.nan, 12]
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(y0, '-o')
(4)                 ax.plot(y1, '-o')
(0)             def test_path_no_doubled_point_in_to_polygon():
(4)                 hand = np.array(
(8)                     [[1.64516129, 1.16145833],
(9)                      [1.64516129, 1.59375],
(9)                      [1.35080645, 1.921875],
(9)                      [1.375, 2.18229167],
(9)                      [1.68548387, 1.9375],
(9)                      [1.60887097, 2.55208333],
(9)                      [1.68548387, 2.69791667],
(9)                      [1.76209677, 2.56770833],
(9)                      [1.83064516, 1.97395833],
(9)                      [1.89516129, 2.75],
(9)                      [1.9516129, 2.84895833],
(9)                      [2.01209677, 2.76041667],
(9)                      [1.99193548, 1.99479167],
(9)                      [2.11290323, 2.63020833],
(9)                      [2.2016129, 2.734375],
(9)                      [2.25403226, 2.60416667],
(9)                      [2.14919355, 1.953125],
(9)                      [2.30645161, 2.36979167],
(9)                      [2.39112903, 2.36979167],
(9)                      [2.41532258, 2.1875],
(9)                      [2.1733871, 1.703125],
(9)                      [2.07782258, 1.16666667]])
(4)                 (r0, c0, r1, c1) = (1.0, 1.5, 2.1, 2.5)
```

```
(4)                 poly = Path(np.vstack((hand[:, 1], hand[:, 0])).T, closed=True)
(4)                 clip_rect = transforms.Bbox([[r0, c0], [r1, c1]])
(4)                 poly_clipped = poly.clip_to_bbox(clip_rect).to_polygons()[0]
(4)                 assert np.all(poly_clipped[-2] != poly_clipped[-1])
(4)                 assert np.all(poly_clipped[-1] == poly_clipped[0])
(0)             def test_path_to_polygons():
(4)                 data = [[10, 10], [20, 20]]
(4)                 p = Path(data)
(4)                 assert_array_equal(p.to_polygons(width=40, height=40), [])
(4)                 assert_array_equal(p.to_polygons(width=40, height=40, closed_only=False),
(23)                                   [data])
(4)                 assert_array_equal(p.to_polygons(), [])
(4)                 assert_array_equal(p.to_polygons(closed_only=False), [data])
(4)                 data = [[10, 10], [20, 20], [30, 30]]
(4)                 closed_data = [[10, 10], [20, 20], [30, 30], [10, 10]]
(4)                 p = Path(data)
(4)                 assert_array_equal(p.to_polygons(width=40, height=40), [closed_data])
(4)                 assert_array_equal(p.to_polygons(width=40, height=40, closed_only=False),
(23)                                   [data])
(4)                 assert_array_equal(p.to_polygons(), [closed_data])
(4)                 assert_array_equal(p.to_polygons(closed_only=False), [data])
(0)             def test_path_deepcopy():
(4)                 verts = [[0, 0], [1, 1]]
(4)                 codes = [Path.MOVETO, Path.LINETO]
(4)                 path1 = Path(verts)
(4)                 path2 = Path(verts, codes)
(4)                 path1_copy = path1.deepcopy()
(4)                 path2_copy = path2.deepcopy()
(4)                 assert path1 is not path1_copy
(4)                 assert path1.vertices is not path1_copy.vertices
(4)                 assert path2 is not path2_copy
(4)                 assert path2.vertices is not path2_copy.vertices
(4)                 assert path2.codes is not path2_copy.codes
(0)             def test_path_shallowcopy():
(4)                 verts = [[0, 0], [1, 1]]
(4)                 codes = [Path.MOVETO, Path.LINETO]
(4)                 path1 = Path(verts)
(4)                 path2 = Path(verts, codes)
(4)                 path1_copy = path1.copy()
(4)                 path2_copy = path2.copy()
(4)                 assert path1 is not path1_copy
(4)                 assert path1.vertices is path1_copy.vertices
(4)                 assert path2 is not path2_copy
(4)                 assert path2.vertices is path2_copy.vertices
(4)                 assert path2.codes is path2_copy.codes
(0)             @pytest.mark.parametrize('phi', np.concatenate([
(4)                 np.array([0, 15, 30, 45, 60, 75, 90, 105, 120, 135]) + delta
(4)                 for delta in [-1, 0, 1]]))
(0)             def test_path_intersect_path(phi):
(4)                 eps_array = [1e-5, 1e-8, 1e-10, 1e-12]
(4)                 transform = transforms.Affine2D().rotate(np.deg2rad(phi))
(4)                 a = Path([(-2, 0), (2, 0)])
(4)                 b = transform.transform_path(a)
(4)                 assert a.intersects_path(b) and b.intersects_path(a)
(4)                 a = Path([(0, 0), (2, 0)])
(4)                 b = transform.transform_path(a)
(4)                 assert a.intersects_path(b) and b.intersects_path(a)
(4)                 a = transform.transform_path(Path([(0, 1), (0, 3)]))
(4)                 b = transform.transform_path(Path([(1, 3), (0, 3)]))
(4)                 assert a.intersects_path(b) and b.intersects_path(a)
(4)                 a = transform.transform_path(Path([(0, 1), (0, 3)]))
(4)                 b = transform.transform_path(Path([(0, 5), (0, 3)]))
(4)                 assert a.intersects_path(b) and b.intersects_path(a)
(4)                 assert a.intersects_path(a)
(4)                 a = transform.transform_path(Path([(0, 0), (5, 5)]))
(4)                 b = transform.transform_path(Path([(1, 1), (3, 3)]))
(4)                 assert a.intersects_path(b) and b.intersects_path(a)
(4)                 a = transform.transform_path(Path([(0, 1), (0, 5)]))
(4)                 b = transform.transform_path(Path([(3, 0), (3, 3)]))
```

```
(4)                 assert not a.intersects_path(b) and not b.intersects_path(a)
(4)                 a = transform.transform_path(Path([(0, 1), (0, 5)]))
(4)                 b = transform.transform_path(Path([(0, 6), (0, 7)]))
(4)                 assert not a.intersects_path(b) and not b.intersects_path(a)
(4)                 for eps in eps_array:
(8)                     a = transform.transform_path(Path([(0, 1), (0, 5)]))
(8)                     b = transform.transform_path(Path([(0 + eps, 1), (0 + eps, 5)]))
(8)                     assert not a.intersects_path(b) and not b.intersects_path(a)
(4)                 for eps in eps_array:
(8)                     a = transform.transform_path(Path([(0, 1), (0, 5)]))
(8)                     b = transform.transform_path(Path([(0, 5 + eps), (0, 7)]))
(8)                     assert not a.intersects_path(b) and not b.intersects_path(a)
(4)                 for eps in eps_array:
(8)                     a = transform.transform_path(Path([(0, 1), (0, 5)]))
(8)                     b = transform.transform_path(Path([(0, 5 - eps), (0, 7)]))
(8)                     assert a.intersects_path(b) and b.intersects_path(a)
(4)                 a = transform.transform_path(Path([(0, 1), (0, 5)]))
(4)                 b = transform.transform_path(Path([(0, 1), (0, 2), (0, 5)]))
(4)                 assert a.intersects_path(b) and b.intersects_path(a)
(4)                 a = transform.transform_path(Path([(1, -1), (0, -1)]))
(4)                 b = transform.transform_path(Path([(0, 1), (0.9, 1)]))
(4)                 assert not a.intersects_path(b) and not b.intersects_path(a)
(4)                 a = transform.transform_path(Path([(0., -5.), (1., -5.)]))
(4)                 b = transform.transform_path(Path([(1., 5.), (0., 5.)]))
(4)                 assert not a.intersects_path(b) and not b.intersects_path(a)
(0)             @pytest.mark.parametrize('offset', range(-720, 361, 45))
(0)             def test_full_arc(offset):
(4)                 low = offset
(4)                 high = 360 + offset
(4)                 path = Path.arc(low, high)
(4)                 mins = np.min(path.vertices, axis=0)
(4)                 maxs = np.max(path.vertices, axis=0)
(4)                 np.testing.assert_allclose(mins, -1)
(4)                 np.testing.assert_allclose(maxs, 1)
(0)             def test_disjoint_zero_length_segment():
(4)                 this_path = Path(
(8)                     np.array([
(12)                        [824.85064295, 2056.26489203],
(12)                        [861.69033931, 2041.00539016],
(12)                        [868.57864109, 2057.63522175],
(12)                        [831.73894473, 2072.89472361],
(12)                        [824.85064295, 2056.26489203]]),
(8)                     np.array([1, 2, 2, 2, 79], dtype=Path.code_type))
(4)                 outline_path = Path(
(8)                     np.array([
(12)                        [859.91051028, 2165.38461538],
(12)                        [859.06772495, 2149.30331334],
(12)                        [859.06772495, 2181.46591743],
(12)                        [859.91051028, 2165.38461538],
(12)                        [859.91051028, 2165.38461538]]),
(8)                     np.array([1, 2, 2, 2, 2],
(17)                             dtype=Path.code_type))
(4)                 assert not outline_path.intersects_path(this_path)
(4)                 assert not this_path.intersects_path(outline_path)
(0)             def test_intersect_zero_length_segment():
(4)                 this_path = Path(
(8)                     np.array([
(12)                        [0, 0],
(12)                        [1, 1],
(8)                     ]))
(4)                 outline_path = Path(
(8)                     np.array([
(12)                        [1, 0],
(12)                        [.5, .5],
(12)                        [.5, .5],
(12)                        [0, 1],
(8)                     ]))
(4)                 assert outline_path.intersects_path(this_path)
(4)                 assert this_path.intersects_path(outline_path)
```

```
371: (0)            def test_cleanup_closepoly():
372: (4)                paths = [
373: (8)                    Path([[np.nan, np.nan], [np.nan, np.nan]],
374: (13)                         [Path.MOVETO, Path.CLOSEPOLY]),
375: (8)                    Path([[np.nan, np.nan], [np.nan, np.nan]]),
376: (8)                    Path([[np.nan, np.nan], [np.nan, np.nan], [np.nan, np.nan],
377: (14)                          [np.nan, np.nan]],
378: (13)                         [Path.MOVETO, Path.CURVE3, Path.CURVE3, Path.CLOSEPOLY])
379: (4)                ]
380: (4)                for p in paths:
381: (8)                    cleaned = p.cleaned(remove_nans=True)
382: (8)                    assert len(cleaned) == 1
383: (8)                    assert cleaned.codes[0] == Path.STOP
```

File 201 - test_pickle.py:

```
1: (0)              from io import BytesIO
2: (0)              import ast
3: (0)              import os
4: (0)              import sys
5: (0)              import pickle
6: (0)              import pickletools
7: (0)              import numpy as np
8: (0)              import pytest
9: (0)              import matplotlib as mpl
10: (0)             from matplotlib import cm
11: (0)             from matplotlib.testing import subprocess_run_helper, is_ci_environment
12: (0)             from matplotlib.testing.decorators import check_figures_equal
13: (0)             from matplotlib.dates import rrulewrapper
14: (0)             from matplotlib.lines import VertexSelector
15: (0)             import matplotlib.pyplot as plt
16: (0)             import matplotlib.transforms as mtransforms
17: (0)             import matplotlib.figure as mfigure
18: (0)             from mpl_toolkits.axes_grid1 import axes_divider, parasite_axes  # type:
ignore
19: (0)             def test_simple():
20: (4)                 fig = plt.figure()
21: (4)                 pickle.dump(fig, BytesIO(), pickle.HIGHEST_PROTOCOL)
22: (4)                 ax = plt.subplot(121)
23: (4)                 pickle.dump(ax, BytesIO(), pickle.HIGHEST_PROTOCOL)
24: (4)                 ax = plt.axes(projection='polar')
25: (4)                 plt.plot(np.arange(10), label='foobar')
26: (4)                 plt.legend()
27: (4)                 pickle.dump(ax, BytesIO(), pickle.HIGHEST_PROTOCOL)
28: (4)                 plt.figure()
29: (4)                 plt.bar(x=np.arange(10), height=np.arange(10))
30: (4)                 pickle.dump(plt.gca(), BytesIO(), pickle.HIGHEST_PROTOCOL)
31: (4)                 fig = plt.figure()
32: (4)                 ax = plt.axes()
33: (4)                 plt.plot(np.arange(10))
34: (4)                 ax.set_yscale('log')
35: (4)                 pickle.dump(fig, BytesIO(), pickle.HIGHEST_PROTOCOL)
36: (0)             def _generate_complete_test_figure(fig_ref):
37: (4)                 fig_ref.set_size_inches((10, 6))
38: (4)                 plt.figure(fig_ref)
39: (4)                 plt.suptitle('Can you fit any more in a figure?')
40: (4)                 x, y = np.arange(8), np.arange(10)
41: (4)                 data = u = v = np.linspace(0, 10, 80).reshape(10, 8)
42: (4)                 v = np.sin(v * -0.6)
43: (4)                 plt.subplot(3, 3, 1)
44: (4)                 plt.plot(list(range(10)))
45: (4)                 plt.ylabel("hello")
46: (4)                 plt.subplot(3, 3, 2)
47: (4)                 plt.contourf(data, hatches=['//', 'ooo'])
48: (4)                 plt.colorbar()
49: (4)                 plt.subplot(3, 3, 3)
50: (4)                 plt.pcolormesh(data)
```

```
 51: (4)                 plt.subplot(3, 3, 4)
 52: (4)                 plt.imshow(data)
 53: (4)                 plt.ylabel("hello\nworld!")
 54: (4)                 plt.subplot(3, 3, 5)
 55: (4)                 plt.pcolor(data)
 56: (4)                 ax = plt.subplot(3, 3, 6)
 57: (4)                 ax.set_xlim(0, 7)
 58: (4)                 ax.set_ylim(0, 9)
 59: (4)                 plt.streamplot(x, y, u, v)
 60: (4)                 ax = plt.subplot(3, 3, 7)
 61: (4)                 ax.set_xlim(0, 7)
 62: (4)                 ax.set_ylim(0, 9)
 63: (4)                 plt.quiver(x, y, u, v)
 64: (4)                 plt.subplot(3, 3, 8)
 65: (4)                 plt.scatter(x, x ** 2, label='$x^2$')
 66: (4)                 plt.legend(loc='upper left')
 67: (4)                 plt.subplot(3, 3, 9)
 68: (4)                 plt.errorbar(x, x * -0.5, xerr=0.2, yerr=0.4, label='$-.5 x$')
 69: (4)                 plt.legend(draggable=True)
 70: (4)                 subfigs = fig_ref.subfigures(2)
 71: (4)                 subfigs[0].subplots(1, 2)
 72: (4)                 subfigs[1].subplots(1, 2)
 73: (4)                 fig_ref.align_ylabels()  # Test handling of _align_label_groups Groupers.
 74: (0)             @mpl.style.context("default")
 75: (0)             @check_figures_equal(extensions=["png"])
 76: (0)             def test_complete(fig_test, fig_ref):
 77: (4)                 _generate_complete_test_figure(fig_ref)
 78: (4)                 pkl = pickle.dumps(fig_ref, pickle.HIGHEST_PROTOCOL)
 79: (4)                 assert "FigureCanvasAgg" not in [arg for op, arg, pos in
pickletools.genops(pkl)]
 80: (4)                 loaded = pickle.loads(pkl)
 81: (4)                 loaded.canvas.draw()
 82: (4)                 fig_test.set_size_inches(loaded.get_size_inches())
 83: (4)                 fig_test.figimage(loaded.canvas.renderer.buffer_rgba())
 84: (4)                 plt.close(loaded)
 85: (0)             def _pickle_load_subprocess():
 86: (4)                 import os
 87: (4)                 import pickle
 88: (4)                 path = os.environ['PICKLE_FILE_PATH']
 89: (4)                 with open(path, 'rb') as blob:
 90: (8)                     fig = pickle.load(blob)
 91: (4)                 print(str(pickle.dumps(fig)))
 92: (0)             @mpl.style.context("default")
 93: (0)             @check_figures_equal(extensions=['png'])
 94: (0)             def test_pickle_load_from_subprocess(fig_test, fig_ref, tmp_path):
 95: (4)                 _generate_complete_test_figure(fig_ref)
 96: (4)                 fp = tmp_path / 'sinus.pickle'
 97: (4)                 assert not fp.exists()
 98: (4)                 with fp.open('wb') as file:
 99: (8)                     pickle.dump(fig_ref, file, pickle.HIGHEST_PROTOCOL)
100: (4)                 assert fp.exists()
101: (4)                 proc = subprocess_run_helper(
102: (8)                     _pickle_load_subprocess,
103: (8)                     timeout=60,
104: (8)                     extra_env={'PICKLE_FILE_PATH': str(fp), 'MPLBACKEND': 'Agg'}
105: (4)                 )
106: (4)                 loaded_fig = pickle.loads(ast.literal_eval(proc.stdout))
107: (4)                 loaded_fig.canvas.draw()
108: (4)                 fig_test.set_size_inches(loaded_fig.get_size_inches())
109: (4)                 fig_test.figimage(loaded_fig.canvas.renderer.buffer_rgba())
110: (4)                 plt.close(loaded_fig)
111: (0)             def test_gcf():
112: (4)                 fig = plt.figure("a label")
113: (4)                 buf = BytesIO()
114: (4)                 pickle.dump(fig, buf, pickle.HIGHEST_PROTOCOL)
115: (4)                 plt.close("all")
116: (4)                 assert plt._pylab_helpers.Gcf.figs == {}  # No figures must be left.
117: (4)                 fig = pickle.loads(buf.getbuffer())
118: (4)                 assert plt._pylab_helpers.Gcf.figs != {}  # A manager is there again.
```

```
(4)                 assert fig.get_label() == "a label"
(0)             def test_no_pyplot():
(4)                 from matplotlib.backends.backend_pdf import FigureCanvasPdf
(4)                 fig = mfigure.Figure()
(4)                 _ = FigureCanvasPdf(fig)
(4)                 ax = fig.add_subplot(1, 1, 1)
(4)                 ax.plot([1, 2, 3], [1, 2, 3])
(4)                 pickle.dump(fig, BytesIO(), pickle.HIGHEST_PROTOCOL)
(0)             def test_renderer():
(4)                 from matplotlib.backends.backend_agg import RendererAgg
(4)                 renderer = RendererAgg(10, 20, 30)
(4)                 pickle.dump(renderer, BytesIO())
(0)             def test_image():
(4)                 from matplotlib.backends.backend_agg import new_figure_manager
(4)                 manager = new_figure_manager(1000)
(4)                 fig = manager.canvas.figure
(4)                 ax = fig.add_subplot(1, 1, 1)
(4)                 ax.imshow(np.arange(12).reshape(3, 4))
(4)                 manager.canvas.draw()
(4)                 pickle.dump(fig, BytesIO())
(0)             def test_polar():
(4)                 plt.subplot(polar=True)
(4)                 fig = plt.gcf()
(4)                 pf = pickle.dumps(fig)
(4)                 pickle.loads(pf)
(4)                 plt.draw()
(0)             class TransformBlob:
(4)                 def __init__(self):
(8)                     self.identity = mtransforms.IdentityTransform()
(8)                     self.identity2 = mtransforms.IdentityTransform()
(8)                     self.composite = mtransforms.CompositeGenericTransform(
(12)                        self.identity,
(12)                        self.identity2)
(8)                     self.wrapper = mtransforms.TransformWrapper(self.composite)
(8)                     self.composite2 = mtransforms.CompositeGenericTransform(
(12)                        self.wrapper,
(12)                        self.identity)
(0)             def test_transform():
(4)                 obj = TransformBlob()
(4)                 pf = pickle.dumps(obj)
(4)                 del obj
(4)                 obj = pickle.loads(pf)
(4)                 assert obj.wrapper._child == obj.composite
(4)                 assert [v() for v in obj.wrapper._parents.values()] == [obj.composite2]
(4)                 assert obj.wrapper.input_dims == obj.composite.input_dims
(4)                 assert obj.wrapper.output_dims == obj.composite.output_dims
(0)             def test_rrulewrapper():
(4)                 r = rrulewrapper(2)
(4)                 try:
(8)                     pickle.loads(pickle.dumps(r))
(4)                 except RecursionError:
(8)                     print('rrulewrapper pickling test failed')
(8)                     raise
(0)             def test_shared():
(4)                 fig, axs = plt.subplots(2, sharex=True)
(4)                 fig = pickle.loads(pickle.dumps(fig))
(4)                 fig.axes[0].set_xlim(10, 20)
(4)                 assert fig.axes[1].get_xlim() == (10, 20)
(0)             def test_inset_and_secondary():
(4)                 fig, ax = plt.subplots()
(4)                 ax.inset_axes([.1, .1, .3, .3])
(4)                 ax.secondary_xaxis("top", functions=(np.square, np.sqrt))
(4)                 pickle.loads(pickle.dumps(fig))
(0)             @pytest.mark.parametrize("cmap", cm._colormaps.values())
(0)             def test_cmap(cmap):
(4)                 pickle.dumps(cmap)
(0)             def test_unpickle_canvas():
(4)                 fig = mfigure.Figure()
(4)                 assert fig.canvas is not None
```

```
188: (4)                out = BytesIO()
189: (4)                pickle.dump(fig, out)
190: (4)                out.seek(0)
191: (4)                fig2 = pickle.load(out)
192: (4)                assert fig2.canvas is not None
193: (0)            def test_mpl_toolkits():
194: (4)                ax = parasite_axes.host_axes([0, 0, 1, 1])
195: (4)                axes_divider.make_axes_area_auto_adjustable(ax)
196: (4)                assert type(pickle.loads(pickle.dumps(ax))) == parasite_axes.HostAxes
197: (0)            def test_standard_norm():
198: (4)                assert type(pickle.loads(pickle.dumps(mpl.colors.LogNorm()))) \
199: (8)                    == mpl.colors.LogNorm
200: (0)            def test_dynamic_norm():
201: (4)                logit_norm_instance = mpl.colors.make_norm_from_scale(
202: (8)                    mpl.scale.LogitScale, mpl.colors.Normalize)()
203: (4)                assert type(pickle.loads(pickle.dumps(logit_norm_instance))) \
204: (8)                    == type(logit_norm_instance)
205: (0)            def test_vertexselector():
206: (4)                line, = plt.plot([0, 1], picker=True)
207: (4)                pickle.loads(pickle.dumps(VertexSelector(line)))
208: (0)            def test_cycler():
209: (4)                ax = plt.figure().add_subplot()
210: (4)                ax.set_prop_cycle(c=["c", "m", "y", "k"])
211: (4)                ax.plot([1, 2])
212: (4)                ax = pickle.loads(pickle.dumps(ax))
213: (4)                l, = ax.plot([3, 4])
214: (4)                assert l.get_color() == "m"
215: (0)            def _test_axeswidget_interactive():
216: (4)                ax = plt.figure().add_subplot()
217: (4)                pickle.dumps(mpl.widgets.Button(ax, "button"))
218: (0)            @pytest.mark.xfail(  # https://github.com/actions/setup-python/issues/649
219: (8)                    ('TF_BUILD' in os.environ or 'GITHUB_ACTION' in os.environ) and
220: (8)                    sys.platform == 'darwin' and sys.version_info[:2] < (3, 11),
221: (8)                    reason='Tk version mismatch on Azure macOS CI'
222: (4)                )
223: (0)            def test_axeswidget_interactive():
224: (4)                subprocess_run_helper(
225: (8)                    _test_axeswidget_interactive,
226: (8)                    timeout=120 if is_ci_environment() else 20,
227: (8)                    extra_env={'MPLBACKEND': 'tkagg'}
228: (4)                )
```

File 202 - test_png.py:

```
1: (0)              from io import BytesIO
2: (0)              from pathlib import Path
3: (0)              import pytest
4: (0)              from matplotlib.testing.decorators import image_comparison
5: (0)              from matplotlib import cm, pyplot as plt
6: (0)              @image_comparison(['pngsuite.png'], tol=0.03)
7: (0)              def test_pngsuite():
8: (4)                  files = sorted(
9: (8)                      (Path(__file__).parent /
"baseline_images/pngsuite").glob("basn*.png"))
10: (4)                 plt.figure(figsize=(len(files), 2))
11: (4)                 for i, fname in enumerate(files):
12: (8)                     data = plt.imread(fname)
13: (8)                     cmap = None  # use default colormap
14: (8)                     if data.ndim == 2:
15: (12)                        cmap = cm.gray
16: (8)                     plt.imshow(data, extent=(i, i + 1, 0, 1), cmap=cmap)
17: (4)                 plt.gca().patch.set_facecolor("#ddffff")
18: (4)                 plt.gca().set_xlim(0, len(files))
19: (0)             def test_truncated_file(tmp_path):
20: (4)                 path = tmp_path / 'test.png'
21: (4)                 path_t = tmp_path / 'test_truncated.png'
22: (4)                 plt.savefig(path)
```

```
23: (4)                 with open(path, 'rb') as fin:
24: (8)                     buf = fin.read()
25: (4)                 with open(path_t, 'wb') as fout:
26: (8)                     fout.write(buf[:20])
27: (4)                 with pytest.raises(Exception):
28: (8)                     plt.imread(path_t)
29: (0)             def test_truncated_buffer():
30: (4)                 b = BytesIO()
31: (4)                 plt.savefig(b)
32: (4)                 b.seek(0)
33: (4)                 b2 = BytesIO(b.read(20))
34: (4)                 b2.seek(0)
35: (4)                 with pytest.raises(Exception):
36: (8)                     plt.imread(b2)
```

File 203 - test_polar.py:

```
1: (0)              import numpy as np
2: (0)              from numpy.testing import assert_allclose
3: (0)              import pytest
4: (0)              import matplotlib as mpl
5: (0)              from matplotlib import pyplot as plt
6: (0)              from matplotlib.testing.decorators import image_comparison,
check_figures_equal
7: (0)              @image_comparison(['polar_axes'], style='default', tol=0.012)
8: (0)              def test_polar_annotations():
9: (4)                  r = np.arange(0.0, 1.0, 0.001)
10: (4)                 theta = 2.0 * 2.0 * np.pi * r
11: (4)                 fig = plt.figure()
12: (4)                 ax = fig.add_subplot(polar=True)
13: (4)                 line, = ax.plot(theta, r, color='#ee8d18', lw=3)
14: (4)                 line, = ax.plot((0, 0), (0, 1), color="#0000ff", lw=1)
15: (4)                 ind = 800
16: (4)                 thisr, thistheta = r[ind], theta[ind]
17: (4)                 ax.plot([thistheta], [thisr], 'o')
18: (4)                 ax.annotate('a polar annotation',
19: (16)                            xy=(thistheta, thisr),  # theta, radius
20: (16)                            xytext=(0.05, 0.05),    # fraction, fraction
21: (16)                            textcoords='figure fraction',
22: (16)                            arrowprops=dict(facecolor='black', shrink=0.05),
23: (16)                            horizontalalignment='left',
24: (16)                            verticalalignment='baseline',
25: (16)                            )
26: (4)                 ax.tick_params(axis='x', tick1On=True, tick2On=True, direction='out')
27: (0)             @image_comparison(['polar_coords'], style='default', remove_text=True,
28: (18)                              tol=0.014)
29: (0)             def test_polar_coord_annotations():
30: (4)                 el = mpl.patches.Ellipse((0, 0), 10, 20, facecolor='r', alpha=0.5)
31: (4)                 fig = plt.figure()
32: (4)                 ax = fig.add_subplot(aspect='equal')
33: (4)                 ax.add_artist(el)
34: (4)                 el.set_clip_box(ax.bbox)
35: (4)                 ax.annotate('the top',
36: (16)                            xy=(np.pi/2., 10.),      # theta, radius
37: (16)                            xytext=(np.pi/3, 20.),   # theta, radius
38: (16)                            xycoords='polar',
39: (16)                            textcoords='polar',
40: (16)                            arrowprops=dict(facecolor='black', shrink=0.05),
41: (16)                            horizontalalignment='left',
42: (16)                            verticalalignment='baseline',
43: (16)                            clip_on=True,  # clip to the axes bounding box
44: (16)                            )
45: (4)                 ax.set_xlim(-20, 20)
46: (4)                 ax.set_ylim(-20, 20)
47: (0)             @image_comparison(['polar_alignment.png'])
48: (0)             def test_polar_alignment():
49: (4)                 angles = np.arange(0, 360, 90)
```

```
(4)                 grid_values = [0, 0.2, 0.4, 0.6, 0.8, 1]
(4)                 fig = plt.figure()
(4)                 rect = [0.1, 0.1, 0.8, 0.8]
(4)                 horizontal = fig.add_axes(rect, polar=True, label='horizontal')
(4)                 horizontal.set_thetagrids(angles)
(4)                 vertical = fig.add_axes(rect, polar=True, label='vertical')
(4)                 vertical.patch.set_visible(False)
(4)                 for i in range(2):
(8)                     fig.axes[i].set_rgrids(
(12)                        grid_values, angle=angles[i],
(12)                        horizontalalignment='left', verticalalignment='top')
(0)             def test_polar_twice():
(4)                 fig = plt.figure()
(4)                 plt.polar([1, 2], [.1, .2])
(4)                 plt.polar([3, 4], [.3, .4])
(4)                 assert len(fig.axes) == 1, 'More than one polar Axes created.'
(0)             @check_figures_equal()
(0)             def test_polar_wrap(fig_test, fig_ref):
(4)                 ax = fig_test.add_subplot(projection="polar")
(4)                 ax.plot(np.deg2rad([179, -179]), [0.2, 0.1])
(4)                 ax.plot(np.deg2rad([2, -2]), [0.2, 0.1])
(4)                 ax = fig_ref.add_subplot(projection="polar")
(4)                 ax.plot(np.deg2rad([179, 181]), [0.2, 0.1])
(4)                 ax.plot(np.deg2rad([2, 358]), [0.2, 0.1])
(0)             @check_figures_equal()
(0)             def test_polar_units_1(fig_test, fig_ref):
(4)                 import matplotlib.testing.jpl_units as units
(4)                 units.register()
(4)                 xs = [30.0, 45.0, 60.0, 90.0]
(4)                 ys = [1.0, 2.0, 3.0, 4.0]
(4)                 plt.figure(fig_test.number)
(4)                 plt.polar([x * units.deg for x in xs], ys)
(4)                 ax = fig_ref.add_subplot(projection="polar")
(4)                 ax.plot(np.deg2rad(xs), ys)
(4)                 ax.set(xlabel="deg")
(0)             @check_figures_equal()
(0)             def test_polar_units_2(fig_test, fig_ref):
(4)                 import matplotlib.testing.jpl_units as units
(4)                 units.register()
(4)                 xs = [30.0, 45.0, 60.0, 90.0]
(4)                 xs_deg = [x * units.deg for x in xs]
(4)                 ys = [1.0, 2.0, 3.0, 4.0]
(4)                 ys_km = [y * units.km for y in ys]
(4)                 plt.figure(fig_test.number)
(4)                 plt.polar(xs_deg, ys_km, thetaunits="rad", runits="km")
(4)                 assert isinstance(plt.gca().xaxis.get_major_formatter(),
(22)                                  units.UnitDblFormatter)
(4)                 ax = fig_ref.add_subplot(projection="polar")
(4)                 ax.plot(np.deg2rad(xs), ys)
(4)                 ax.xaxis.set_major_formatter(mpl.ticker.FuncFormatter("{:.12}".format))
(4)                 ax.set(xlabel="rad", ylabel="km")
(0)             @image_comparison(['polar_rmin'], style='default')
(0)             def test_polar_rmin():
(4)                 r = np.arange(0, 3.0, 0.01)
(4)                 theta = 2*np.pi*r
(4)                 fig = plt.figure()
(4)                 ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar=True)
(4)                 ax.plot(theta, r)
(4)                 ax.set_rmax(2.0)
(4)                 ax.set_rmin(0.5)
(0)             @image_comparison(['polar_negative_rmin'], style='default')
(0)             def test_polar_negative_rmin():
(4)                 r = np.arange(-3.0, 0.0, 0.01)
(4)                 theta = 2*np.pi*r
(4)                 fig = plt.figure()
(4)                 ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar=True)
(4)                 ax.plot(theta, r)
(4)                 ax.set_rmax(0.0)
(4)                 ax.set_rmin(-3.0)
```

```
(0)             @image_comparison(['polar_rorigin'], style='default')
(0)             def test_polar_rorigin():
(4)                 r = np.arange(0, 3.0, 0.01)
(4)                 theta = 2*np.pi*r
(4)                 fig = plt.figure()
(4)                 ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar=True)
(4)                 ax.plot(theta, r)
(4)                 ax.set_rmax(2.0)
(4)                 ax.set_rmin(0.5)
(4)                 ax.set_rorigin(0.0)
(0)             @image_comparison(['polar_invertedylim.png'], style='default')
(0)             def test_polar_invertedylim():
(4)                 fig = plt.figure()
(4)                 ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar=True)
(4)                 ax.set_ylim(2, 0)
(0)             @image_comparison(['polar_invertedylim_rorigin.png'], style='default')
(0)             def test_polar_invertedylim_rorigin():
(4)                 fig = plt.figure()
(4)                 ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar=True)
(4)                 ax.yaxis.set_inverted(True)
(4)                 ax.plot([0, 0], [0, 2], c="none")
(4)                 ax.margins(0)
(4)                 ax.set_rorigin(3)
(0)             @image_comparison(['polar_theta_position'], style='default')
(0)             def test_polar_theta_position():
(4)                 r = np.arange(0, 3.0, 0.01)
(4)                 theta = 2*np.pi*r
(4)                 fig = plt.figure()
(4)                 ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar=True)
(4)                 ax.plot(theta, r)
(4)                 ax.set_theta_zero_location("NW", 30)
(4)                 ax.set_theta_direction('clockwise')
(0)             @image_comparison(['polar_rlabel_position'], style='default')
(0)             def test_polar_rlabel_position():
(4)                 fig = plt.figure()
(4)                 ax = fig.add_subplot(projection='polar')
(4)                 ax.set_rlabel_position(315)
(4)                 ax.tick_params(rotation='auto')
(0)             @image_comparison(['polar_theta_wedge'], style='default')
(0)             def test_polar_theta_limits():
(4)                 r = np.arange(0, 3.0, 0.01)
(4)                 theta = 2*np.pi*r
(4)                 theta_mins = np.arange(15.0, 361.0, 90.0)
(4)                 theta_maxs = np.arange(50.0, 361.0, 90.0)
(4)                 DIRECTIONS = ('out', 'in', 'inout')
(4)                 fig, axs = plt.subplots(len(theta_mins), len(theta_maxs),
(28)                                        subplot_kw={'polar': True},
(28)                                        figsize=(8, 6))
(4)                 for i, start in enumerate(theta_mins):
(8)                     for j, end in enumerate(theta_maxs):
(12)                        ax = axs[i, j]
(12)                        ax.plot(theta, r)
(12)                        if start < end:
(16)                            ax.set_thetamin(start)
(16)                            ax.set_thetamax(end)
(12)                        else:
(16)                            ax.set_thetamin(end)
(16)                            ax.set_thetamax(start)
(16)                            ax.set_theta_direction('clockwise')
(12)                        ax.tick_params(tick1On=True, tick2On=True,
(27)                                       direction=DIRECTIONS[i % len(DIRECTIONS)],
(27)                                       rotation='auto')
(12)                        ax.yaxis.set_tick_params(label2On=True, rotation='auto')
(12)                        ax.xaxis.get_major_locator().base.set_params(  # backcompat
(16)                            steps=[1, 2, 2.5, 5, 10])
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_polar_rlim(fig_test, fig_ref):
(4)                 ax = fig_test.subplots(subplot_kw={'polar': True})
(4)                 ax.set_rlim(top=10)
```

```
188: (4)                ax.set_rlim(bottom=.5)
189: (4)                ax = fig_ref.subplots(subplot_kw={'polar': True})
190: (4)                ax.set_rmax(10.)
191: (4)                ax.set_rmin(.5)
192: (0)            @check_figures_equal(extensions=["png"])
193: (0)            def test_polar_rlim_bottom(fig_test, fig_ref):
194: (4)                ax = fig_test.subplots(subplot_kw={'polar': True})
195: (4)                ax.set_rlim(bottom=[.5, 10])
196: (4)                ax = fig_ref.subplots(subplot_kw={'polar': True})
197: (4)                ax.set_rmax(10.)
198: (4)                ax.set_rmin(.5)
199: (0)            def test_polar_rlim_zero():
200: (4)                ax = plt.figure().add_subplot(projection='polar')
201: (4)                ax.plot(np.arange(10), np.arange(10) + .01)
202: (4)                assert ax.get_ylim()[0] == 0
203: (0)            def test_polar_no_data():
204: (4)                plt.subplot(projection="polar")
205: (4)                ax = plt.gca()
206: (4)                assert ax.get_rmin() == 0 and ax.get_rmax() == 1
207: (4)                plt.close("all")
208: (4)                plt.polar()
209: (4)                ax = plt.gca()
210: (4)                assert ax.get_rmin() == 0 and ax.get_rmax() == 1
211: (0)            def test_polar_default_log_lims():
212: (4)                plt.subplot(projection='polar')
213: (4)                ax = plt.gca()
214: (4)                ax.set_rscale('log')
215: (4)                assert ax.get_rmin() > 0
216: (0)            def test_polar_not_datalim_adjustable():
217: (4)                ax = plt.figure().add_subplot(projection="polar")
218: (4)                with pytest.raises(ValueError):
219: (8)                    ax.set_adjustable("datalim")
220: (0)            def test_polar_gridlines():
221: (4)                fig = plt.figure()
222: (4)                ax = fig.add_subplot(polar=True)
223: (4)                ax.grid(alpha=0.2)
224: (4)                plt.setp(ax.yaxis.get_ticklabels(), visible=False)
225: (4)                fig.canvas.draw()
226: (4)                assert ax.xaxis.majorTicks[0].gridline.get_alpha() == .2
227: (4)                assert ax.yaxis.majorTicks[0].gridline.get_alpha() == .2
228: (0)            def test_get_tightbbox_polar():
229: (4)                fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
230: (4)                fig.canvas.draw()
231: (4)                bb = ax.get_tightbbox(fig.canvas.get_renderer())
232: (4)                assert_allclose(
233: (8)                    bb.extents, [107.7778,  29.2778, 539.7847, 450.7222], rtol=1e-03)
234: (0)            @check_figures_equal(extensions=["png"])
235: (0)            def test_polar_interpolation_steps_constant_r(fig_test, fig_ref):
236: (4)                p1 = (fig_test.add_subplot(121, projection="polar")
237: (10)                     .bar([0], [1], 3*np.pi, edgecolor="none", antialiased=False))
238: (4)                p2 = (fig_test.add_subplot(122, projection="polar")
239: (10)                     .bar([0], [1], -3*np.pi, edgecolor="none", antialiased=False))
240: (4)                p3 = (fig_ref.add_subplot(121, projection="polar")
241: (10)                     .bar([0], [1], 2*np.pi, edgecolor="none", antialiased=False))
242: (4)                p4 = (fig_ref.add_subplot(122, projection="polar")
243: (10)                     .bar([0], [1], -2*np.pi, edgecolor="none", antialiased=False))
244: (0)            @check_figures_equal(extensions=["png"])
245: (0)            def test_polar_interpolation_steps_variable_r(fig_test, fig_ref):
246: (4)                l, = fig_test.add_subplot(projection="polar").plot([0, np.pi/2], [1, 2])
247: (4)                l.get_path()._interpolation_steps = 100
248: (4)                fig_ref.add_subplot(projection="polar").plot(
249: (8)                    np.linspace(0, np.pi/2, 101), np.linspace(1, 2, 101))
250: (0)            def test_thetalim_valid_invalid():
251: (4)                ax = plt.subplot(projection='polar')
252: (4)                ax.set_thetalim(0, 2 * np.pi)  # doesn't raise.
253: (4)                ax.set_thetalim(thetamin=800, thetamax=440)  # doesn't raise.
254: (4)                with pytest.raises(ValueError,
255: (23)                                   match='angle range must be less than a full circle'):
256: (8)                    ax.set_thetalim(0, 3 * np.pi)
```

```
(4)                 with pytest.raises(ValueError,
(23)                                   match='angle range must be less than a full circle'):
(8)                     ax.set_thetalim(thetamin=800, thetamax=400)
(0)             def test_thetalim_args():
(4)                 ax = plt.subplot(projection='polar')
(4)                 ax.set_thetalim(0, 1)
(4)                 assert tuple(np.radians((ax.get_thetamin(), ax.get_thetamax()))) == (0, 1)
(4)                 ax.set_thetalim((2, 3))
(4)                 assert tuple(np.radians((ax.get_thetamin(), ax.get_thetamax()))) == (2, 3)
(0)             def test_default_thetalocator():
(4)                 fig, axs = plt.subplot_mosaic(
(8)                     "AAAABB.", subplot_kw={"projection": "polar"})
(4)                 for ax in axs.values():
(8)                     ax.set_thetalim(0, np.pi)
(4)                 for ax in axs.values():
(8)                     ticklocs = np.degrees(ax.xaxis.get_majorticklocs()).tolist()
(8)                     assert pytest.approx(90) in ticklocs
(8)                     assert pytest.approx(100) not in ticklocs
(0)             def test_axvspan():
(4)                 ax = plt.subplot(projection="polar")
(4)                 span = ax.axvspan(0, np.pi/4)
(4)                 assert span.get_path()._interpolation_steps > 1
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_remove_shared_polar(fig_ref, fig_test):
(4)                 axs = fig_ref.subplots(
(8)                     2, 2, sharex=True, subplot_kw={"projection": "polar"})
(4)                 for i in [0, 1, 3]:
(8)                     axs.flat[i].remove()
(4)                 axs = fig_test.subplots(
(8)                     2, 2, sharey=True, subplot_kw={"projection": "polar"})
(4)                 for i in [0, 1, 3]:
(8)                     axs.flat[i].remove()
(0)             def test_shared_polar_keeps_ticklabels():
(4)                 fig, axs = plt.subplots(
(8)                     2, 2, subplot_kw={"projection": "polar"}, sharex=True, sharey=True)
(4)                 fig.canvas.draw()
(4)                 assert axs[0, 1].xaxis.majorTicks[0].get_visible()
(4)                 assert axs[0, 1].yaxis.majorTicks[0].get_visible()
(4)                 fig, axs = plt.subplot_mosaic(
(8)                     "ab\ncd", subplot_kw={"projection": "polar"}, sharex=True,
sharey=True)
(4)                 fig.canvas.draw()
(4)                 assert axs["b"].xaxis.majorTicks[0].get_visible()
(4)                 assert axs["b"].yaxis.majorTicks[0].get_visible()
(0)             def test_axvline_axvspan_do_not_modify_rlims():
(4)                 ax = plt.subplot(projection="polar")
(4)                 ax.axvspan(0, 1)
(4)                 ax.axvline(.5)
(4)                 ax.plot([.1, .2])
(4)                 assert ax.get_ylim() == (0, .2)
(0)             def test_cursor_precision():
(4)                 ax = plt.subplot(projection="polar")
(4)                 assert ax.format_coord(0, 0.005) == "θ=0.0π (0°), r=0.005"
(4)                 assert ax.format_coord(0, .1) == "θ=0.00π (0°), r=0.100"
(4)                 assert ax.format_coord(0, 1) == "θ=0.000π (0.0°), r=1.000"
(4)                 assert ax.format_coord(1, 0.005) == "θ=0.3π (57°), r=0.005"
(4)                 assert ax.format_coord(1, .1) == "θ=0.32π (57°), r=0.100"
(4)                 assert ax.format_coord(1, 1) == "θ=0.318π (57.3°), r=1.000"
(4)                 assert ax.format_coord(2, 0.005) == "θ=0.6π (115°), r=0.005"
(4)                 assert ax.format_coord(2, .1) == "θ=0.64π (115°), r=0.100"
(4)                 assert ax.format_coord(2, 1) == "θ=0.637π (114.6°), r=1.000"
(0)             @image_comparison(['polar_log.png'], style='default')
(0)             def test_polar_log():
(4)                 fig = plt.figure()
(4)                 ax = fig.add_subplot(polar=True)
(4)                 ax.set_rscale('log')
(4)                 ax.set_rlim(1, 1000)
(4)                 n = 100
(4)                 ax.plot(np.linspace(0, 2 * np.pi, n), np.logspace(0, 2, n))
```

```
325: (0)            def test_polar_neg_theta_lims():
326: (4)                fig = plt.figure()
327: (4)                ax = fig.add_subplot(projection='polar')
328: (4)                ax.set_thetalim(-np.pi, np.pi)
329: (4)                labels = [l.get_text() for l in ax.xaxis.get_ticklabels()]
330: (4)                assert labels == ['-180°', '-135°', '-90°', '-45°', '0°', '45°', '90°',
'135°']
```

File 204 - test_preprocess_data.py:

```
1: (0)              import re
2: (0)              import sys
3: (0)              import numpy as np
4: (0)              import pytest
5: (0)              from matplotlib import _preprocess_data
6: (0)              from matplotlib.axes import Axes
7: (0)              from matplotlib.testing import subprocess_run_for_testing
8: (0)              from matplotlib.testing.decorators import check_figures_equal
9: (0)              @_preprocess_data(replace_names=["x", "y"], label_namer="y")
10: (0)             def plot_func(ax, x, y, ls="x", label=None, w="xyz"):
11: (4)                 return f"x: {list(x)}, y: {list(y)}, ls: {ls}, w: {w}, label: {label}"
12: (0)             all_funcs = [plot_func]
13: (0)             all_func_ids = ['plot_func']
14: (0)             def test_compiletime_checks():
15: (4)                 """Test decorator invocations -> no replacements."""
16: (4)                 def func(ax, x, y): pass
17: (4)                 def func_args(ax, x, y, *args): pass
18: (4)                 def func_kwargs(ax, x, y, **kwargs): pass
19: (4)                 def func_no_ax_args(*args, **kwargs): pass
20: (4)                 _preprocess_data(replace_names=["x", "y"])(func)
21: (4)                 _preprocess_data(replace_names=["x", "y"])(func_kwargs)
22: (4)                 _preprocess_data(replace_names=["x", "y"])(func_args)
23: (4)                 with pytest.raises(AssertionError):
24: (8)                     _preprocess_data(replace_names=["x", "y", "z"])(func_args)
25: (4)                 _preprocess_data(replace_names=[], label_namer=None)(func)
26: (4)                 _preprocess_data(replace_names=[], label_namer=None)(func_args)
27: (4)                 _preprocess_data(replace_names=[], label_namer=None)(func_kwargs)
28: (4)                 _preprocess_data(replace_names=[], label_namer=None)(func_no_ax_args)
29: (4)                 with pytest.raises(AssertionError):
30: (8)                     _preprocess_data(label_namer="z")(func)
31: (4)                 with pytest.raises(AssertionError):
32: (8)                     _preprocess_data(label_namer="z")(func_args)
33: (0)             @pytest.mark.parametrize('func', all_funcs, ids=all_func_ids)
34: (0)             def test_function_call_without_data(func):
35: (4)                 """Test without data -> no replacements."""
36: (4)                 assert (func(None, "x", "y") ==
37: (12)                        "x: ['x'], y: ['y'], ls: x, w: xyz, label: None")
38: (4)                 assert (func(None, x="x", y="y") ==
39: (12)                        "x: ['x'], y: ['y'], ls: x, w: xyz, label: None")
40: (4)                 assert (func(None, "x", "y", label="") ==
41: (12)                        "x: ['x'], y: ['y'], ls: x, w: xyz, label: ")
42: (4)                 assert (func(None, "x", "y", label="text") ==
43: (12)                        "x: ['x'], y: ['y'], ls: x, w: xyz, label: text")
44: (4)                 assert (func(None, x="x", y="y", label="") ==
45: (12)                        "x: ['x'], y: ['y'], ls: x, w: xyz, label: ")
46: (4)                 assert (func(None, x="x", y="y", label="text") ==
47: (12)                        "x: ['x'], y: ['y'], ls: x, w: xyz, label: text")
48: (0)             @pytest.mark.parametrize('func', all_funcs, ids=all_func_ids)
49: (0)             def test_function_call_with_dict_input(func):
50: (4)                 """Tests with dict input, unpacking via preprocess_pipeline"""
51: (4)                 data = {'a': 1, 'b': 2}
52: (4)                 assert (func(None, data.keys(), data.values()) ==
53: (12)                        "x: ['a', 'b'], y: [1, 2], ls: x, w: xyz, label: None")
54: (0)             @pytest.mark.parametrize('func', all_funcs, ids=all_func_ids)
55: (0)             def test_function_call_with_dict_data(func):
56: (4)                 """Test with dict data -> label comes from the value of 'x' parameter."""
57: (4)                 data = {"a": [1, 2], "b": [8, 9], "w": "NOT"}
```

```
58: (4)                 assert (func(None, "a", "b", data=data) ==
59: (12)                        "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: b")
60: (4)                 assert (func(None, x="a", y="b", data=data) ==
61: (12)                        "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: b")
62: (4)                 assert (func(None, "a", "b", label="", data=data) ==
63: (12)                        "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: ")
64: (4)                 assert (func(None, "a", "b", label="text", data=data) ==
65: (12)                        "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: text")
66: (4)                 assert (func(None, x="a", y="b", label="", data=data) ==
67: (12)                        "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: ")
68: (4)                 assert (func(None, x="a", y="b", label="text", data=data) ==
69: (12)                        "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: text")
70: (0)             @pytest.mark.parametrize('func', all_funcs, ids=all_func_ids)
71: (0)             def test_function_call_with_dict_data_not_in_data(func):
72: (4)                 """Test the case that one var is not in data -> half replaces, half
kept"""
73: (4)                 data = {"a": [1, 2], "w": "NOT"}
74: (4)                 assert (func(None, "a", "b", data=data) ==
75: (12)                        "x: [1, 2], y: ['b'], ls: x, w: xyz, label: b")
76: (4)                 assert (func(None, x="a", y="b", data=data) ==
77: (12)                        "x: [1, 2], y: ['b'], ls: x, w: xyz, label: b")
78: (4)                 assert (func(None, "a", "b", label="", data=data) ==
79: (12)                        "x: [1, 2], y: ['b'], ls: x, w: xyz, label: ")
80: (4)                 assert (func(None, "a", "b", label="text", data=data) ==
81: (12)                        "x: [1, 2], y: ['b'], ls: x, w: xyz, label: text")
82: (4)                 assert (func(None, x="a", y="b", label="", data=data) ==
83: (12)                        "x: [1, 2], y: ['b'], ls: x, w: xyz, label: ")
84: (4)                 assert (func(None, x="a", y="b", label="text", data=data) ==
85: (12)                        "x: [1, 2], y: ['b'], ls: x, w: xyz, label: text")
86: (0)             @pytest.mark.parametrize('func', all_funcs, ids=all_func_ids)
87: (0)             def test_function_call_with_pandas_data(func, pd):
88: (4)                 """Test with pandas dataframe -> label comes from ``data["col"].name``."""
89: (4)                 data = pd.DataFrame({"a": np.array([1, 2], dtype=np.int32),
90: (25)                                     "b": np.array([8, 9], dtype=np.int32),
91: (25)                                     "w": ["NOT", "NOT"]})
92: (4)                 assert (func(None, "a", "b", data=data) ==
93: (12)                        "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: b")
94: (4)                 assert (func(None, x="a", y="b", data=data) ==
95: (12)                        "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: b")
96: (4)                 assert (func(None, "a", "b", label="", data=data) ==
97: (12)                        "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: ")
98: (4)                 assert (func(None, "a", "b", label="text", data=data) ==
99: (12)                        "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: text")
100: (4)                assert (func(None, x="a", y="b", label="", data=data) ==
101: (12)                       "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: ")
102: (4)                assert (func(None, x="a", y="b", label="text", data=data) ==
103: (12)                       "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: text")
104: (0)            def test_function_call_replace_all():
105: (4)                """Test without a "replace_names" argument, all vars should be
replaced."""
106: (4)                data = {"a": [1, 2], "b": [8, 9], "x": "xyz"}
107: (4)                @_preprocess_data(label_namer="y")
108: (4)                def func_replace_all(ax, x, y, ls="x", label=None, w="NOT"):
109: (8)                    return f"x: {list(x)}, y: {list(y)}, ls: {ls}, w: {w}, label: {label}"
110: (4)                assert (func_replace_all(None, "a", "b", w="x", data=data) ==
111: (12)                       "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: b")
112: (4)                assert (func_replace_all(None, x="a", y="b", w="x", data=data) ==
113: (12)                       "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: b")
114: (4)                assert (func_replace_all(None, "a", "b", w="x", label="", data=data) ==
115: (12)                       "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: ")
116: (4)                assert (
117: (8)                    func_replace_all(None, "a", "b", w="x", label="text", data=data) ==
118: (8)                    "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: text")
119: (4)                assert (
120: (8)                    func_replace_all(None, x="a", y="b", w="x", label="", data=data) ==
121: (8)                    "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: ")
122: (4)                assert (
123: (8)                    func_replace_all(None, x="a", y="b", w="x", label="text", data=data)
==
```

```
124: (8)                    "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: text")
125: (0)            def test_no_label_replacements():
126: (4)                """Test with "label_namer=None" -> no label replacement at all."""
127: (4)                @_preprocess_data(replace_names=["x", "y"], label_namer=None)
128: (4)                def func_no_label(ax, x, y, ls="x", label=None, w="xyz"):
129: (8)                    return f"x: {list(x)}, y: {list(y)}, ls: {ls}, w: {w}, label: {label}"
130: (4)                data = {"a": [1, 2], "b": [8, 9], "w": "NOT"}
131: (4)                assert (func_no_label(None, "a", "b", data=data) ==
132: (12)                       "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: None")
133: (4)                assert (func_no_label(None, x="a", y="b", data=data) ==
134: (12)                       "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: None")
135: (4)                assert (func_no_label(None, "a", "b", label="", data=data) ==
136: (12)                       "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: ")
137: (4)                assert (func_no_label(None, "a", "b", label="text", data=data) ==
138: (12)                       "x: [1, 2], y: [8, 9], ls: x, w: xyz, label: text")
139: (0)            def test_more_args_than_pos_parameter():
140: (4)                @_preprocess_data(replace_names=["x", "y"], label_namer="y")
141: (4)                def func(ax, x, y, z=1):
142: (8)                    pass
143: (4)                data = {"a": [1, 2], "b": [8, 9], "w": "NOT"}
144: (4)                with pytest.raises(TypeError):
145: (8)                    func(None, "a", "b", "z", "z", data=data)
146: (0)            def test_docstring_addition():
147: (4)                @_preprocess_data()
148: (4)                def funcy(ax, *args, **kwargs):
149: (8)                    """
150: (8)                    Parameters
151: (8)                    ----------
152: (8)                    data : indexable object, optional
153: (12)                       DATA_PARAMETER_PLACEHOLDER
154: (8)                    """
155: (4)                assert re.search(r"all parameters also accept a string", funcy.__doc__)
156: (4)                assert not re.search(r"the following parameters", funcy.__doc__)
157: (4)                @_preprocess_data(replace_names=[])
158: (4)                def funcy(ax, x, y, z, bar=None):
159: (8)                    """
160: (8)                    Parameters
161: (8)                    ----------
162: (8)                    data : indexable object, optional
163: (12)                       DATA_PARAMETER_PLACEHOLDER
164: (8)                    """
165: (4)                assert not re.search(r"all parameters also accept a string",
funcy.__doc__)
166: (4)                assert not re.search(r"the following parameters", funcy.__doc__)
167: (4)                @_preprocess_data(replace_names=["bar"])
168: (4)                def funcy(ax, x, y, z, bar=None):
169: (8)                    """
170: (8)                    Parameters
171: (8)                    ----------
172: (8)                    data : indexable object, optional
173: (12)                       DATA_PARAMETER_PLACEHOLDER
174: (8)                    """
175: (4)                assert not re.search(r"all parameters also accept a string",
funcy.__doc__)
176: (4)                assert not re.search(r"the following parameters .*: \*bar\*\.",
177: (25)                                    funcy.__doc__)
178: (4)                @_preprocess_data(replace_names=["x", "t"])
179: (4)                def funcy(ax, x, y, z, t=None):
180: (8)                    """
181: (8)                    Parameters
182: (8)                    ----------
183: (8)                    data : indexable object, optional
184: (12)                       DATA_PARAMETER_PLACEHOLDER
185: (8)                    """
186: (4)                assert not re.search(r"all parameters also accept a string",
funcy.__doc__)
187: (4)                assert not re.search(r"the following parameters .*: \*x\*, \*t\*\.",
188: (25)                                    funcy.__doc__)
189: (0)            def test_data_parameter_replacement():
```

```
190: (4)                """
191: (4)                Test that the docstring contains the correct *data* parameter stub
192: (4)                for all methods that we run _preprocess_data() on.
193: (4)                """
194: (4)                program = (
195: (8)                    "import logging; "
196: (8)                    "logging.basicConfig(level=logging.DEBUG); "
197: (8)                    "import matplotlib.pyplot as plt"
198: (4)                )
199: (4)                cmd = [sys.executable, "-c", program]
200: (4)                completed_proc = subprocess_run_for_testing(
201: (8)                    cmd, text=True, capture_output=True
202: (4)                )
203: (4)                assert 'data parameter docstring error' not in completed_proc.stderr
204: (0)            class TestPlotTypes:
205: (4)                plotters = [Axes.scatter, Axes.bar, Axes.plot]
206: (4)                @pytest.mark.parametrize('plotter', plotters)
207: (4)                @check_figures_equal(extensions=['png'])
208: (4)                def test_dict_unpack(self, plotter, fig_test, fig_ref):
209: (8)                    x = [1, 2, 3]
210: (8)                    y = [4, 5, 6]
211: (8)                    ddict = dict(zip(x, y))
212: (8)                    plotter(fig_test.subplots(),
213: (16)                           ddict.keys(), ddict.values())
214: (8)                    plotter(fig_ref.subplots(), x, y)
215: (4)                @pytest.mark.parametrize('plotter', plotters)
216: (4)                @check_figures_equal(extensions=['png'])
217: (4)                def test_data_kwarg(self, plotter, fig_test, fig_ref):
218: (8)                    x = [1, 2, 3]
219: (8)                    y = [4, 5, 6]
220: (8)                    plotter(fig_test.subplots(), 'xval', 'yval',
221: (16)                           data={'xval': x, 'yval': y})
222: (8)                    plotter(fig_ref.subplots(), x, y)
```

File 205 - test_pyplot.py:

```
1: (0)              import difflib
2: (0)              import numpy as np
3: (0)              import sys
4: (0)              from pathlib import Path
5: (0)              import pytest
6: (0)              import matplotlib as mpl
7: (0)              from matplotlib.testing import subprocess_run_for_testing
8: (0)              from matplotlib import pyplot as plt
9: (0)              def test_pyplot_up_to_date(tmp_path):
10: (4)                 pytest.importorskip("black")
11: (4)                 gen_script = Path(mpl.__file__).parents[2] / "tools/boilerplate.py"
12: (4)                 if not gen_script.exists():
13: (8)                     pytest.skip("boilerplate.py not found")
14: (4)                 orig_contents = Path(plt.__file__).read_text()
15: (4)                 plt_file = tmp_path / 'pyplot.py'
16: (4)                 plt_file.write_text(orig_contents, 'utf-8')
17: (4)                 subprocess_run_for_testing(
18: (8)                     [sys.executable, str(gen_script), str(plt_file)],
19: (8)                     check=True)
20: (4)                 new_contents = plt_file.read_text('utf-8')
21: (4)                 if orig_contents != new_contents:
22: (8)                     diff_msg = '\n'.join(
23: (12)                        difflib.unified_diff(
24: (16)                            orig_contents.split('\n'), new_contents.split('\n'),
25: (16)                            fromfile='found pyplot.py',
26: (16)                            tofile='expected pyplot.py',
27: (16)                            n=0, lineterm=''))
28: (8)                     pytest.fail(
29: (12)                        "pyplot.py is not up-to-date. Please run "
30: (12)                        "'python tools/boilerplate.py' to update pyplot.py. "
31: (12)                        "This needs to be done from an environment where your "
```

```
(12)                        "current working copy is installed (e.g. 'pip install -e'd). "
(12)                        "Here is a diff of unexpected differences:\n%s" % diff_msg
(8)                     )
(0)             def test_copy_docstring_and_deprecators(recwarn):
(4)                 @mpl._api.rename_parameter(mpl.__version__, "old", "new")
(4)                 @mpl._api.make_keyword_only(mpl.__version__, "kwo")
(4)                 def func(new, kwo=None):
(8)                     pass
(4)                 @plt._copy_docstring_and_deprecators(func)
(4)                 def wrapper_func(new, kwo=None):
(8)                     pass
(4)                 wrapper_func(None)
(4)                 wrapper_func(new=None)
(4)                 wrapper_func(None, kwo=None)
(4)                 wrapper_func(new=None, kwo=None)
(4)                 assert not recwarn
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning):
(8)                     wrapper_func(old=None)
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning):
(8)                     wrapper_func(None, None)
(0)             def test_pyplot_box():
(4)                 fig, ax = plt.subplots()
(4)                 plt.box(False)
(4)                 assert not ax.get_frame_on()
(4)                 plt.box(True)
(4)                 assert ax.get_frame_on()
(4)                 plt.box()
(4)                 assert not ax.get_frame_on()
(4)                 plt.box()
(4)                 assert ax.get_frame_on()
(0)             def test_stackplot_smoke():
(4)                 plt.stackplot([1, 2, 3], [1, 2, 3])
(0)             def test_nrows_error():
(4)                 with pytest.raises(TypeError):
(8)                     plt.subplot(nrows=1)
(4)                 with pytest.raises(TypeError):
(8)                     plt.subplot(ncols=1)
(0)             def test_ioff():
(4)                 plt.ion()
(4)                 assert mpl.is_interactive()
(4)                 with plt.ioff():
(8)                     assert not mpl.is_interactive()
(4)                 assert mpl.is_interactive()
(4)                 plt.ioff()
(4)                 assert not mpl.is_interactive()
(4)                 with plt.ioff():
(8)                     assert not mpl.is_interactive()
(4)                 assert not mpl.is_interactive()
(0)             def test_ion():
(4)                 plt.ioff()
(4)                 assert not mpl.is_interactive()
(4)                 with plt.ion():
(8)                     assert mpl.is_interactive()
(4)                 assert not mpl.is_interactive()
(4)                 plt.ion()
(4)                 assert mpl.is_interactive()
(4)                 with plt.ion():
(8)                     assert mpl.is_interactive()
(4)                 assert mpl.is_interactive()
(0)             def test_nested_ion_ioff():
(4)                 plt.ion()
(4)                 with plt.ioff():
(8)                     assert not mpl.is_interactive()
(8)                     with plt.ion():
(12)                        assert mpl.is_interactive()
(8)                     assert not mpl.is_interactive()
(4)                 assert mpl.is_interactive()
(4)                 with plt.ioff():
(8)                     with plt.ioff():
```

```
(12)                        assert not mpl.is_interactive()
(4)                 assert mpl.is_interactive()
(4)                 with plt.ion():
(8)                     plt.ioff()
(4)                 assert mpl.is_interactive()
(4)                 plt.ioff()
(4)                 with plt.ion():
(8)                     assert mpl.is_interactive()
(8)                     with plt.ioff():
(12)                        assert not mpl.is_interactive()
(8)                     assert mpl.is_interactive()
(4)                 assert not mpl.is_interactive()
(4)                 with plt.ion():
(8)                     with plt.ion():
(12)                        assert mpl.is_interactive()
(4)                 assert not mpl.is_interactive()
(4)                 with plt.ioff():
(8)                     plt.ion()
(4)                 assert not mpl.is_interactive()
(0)             def test_close():
(4)                 try:
(8)                     plt.close(1.1)
(4)                 except TypeError as e:
(8)                     assert str(e) == "close() argument must be a Figure, an int, " \
(25)                                     "a string, or None, not <class 'float'>"
(0)             def test_subplot_reuse():
(4)                 ax1 = plt.subplot(121)
(4)                 assert ax1 is plt.gca()
(4)                 ax2 = plt.subplot(122)
(4)                 assert ax2 is plt.gca()
(4)                 ax3 = plt.subplot(121)
(4)                 assert ax1 is plt.gca()
(4)                 assert ax1 is ax3
(0)             def test_axes_kwargs():
(4)                 plt.figure()
(4)                 ax = plt.axes()
(4)                 ax1 = plt.axes()
(4)                 assert ax is not None
(4)                 assert ax1 is not ax
(4)                 plt.close()
(4)                 plt.figure()
(4)                 ax = plt.axes(projection='polar')
(4)                 ax1 = plt.axes(projection='polar')
(4)                 assert ax is not None
(4)                 assert ax1 is not ax
(4)                 plt.close()
(4)                 plt.figure()
(4)                 ax = plt.axes(projection='polar')
(4)                 ax1 = plt.axes()
(4)                 assert ax is not None
(4)                 assert ax1.name == 'rectilinear'
(4)                 assert ax1 is not ax
(4)                 plt.close()
(0)             def test_subplot_replace_projection():
(4)                 fig = plt.figure()
(4)                 ax = plt.subplot(1, 2, 1)
(4)                 ax1 = plt.subplot(1, 2, 1)
(4)                 ax2 = plt.subplot(1, 2, 2)
(4)                 ax3 = plt.subplot(1, 2, 1, projection='polar')
(4)                 ax4 = plt.subplot(1, 2, 1, projection='polar')
(4)                 assert ax is not None
(4)                 assert ax1 is ax
(4)                 assert ax2 is not ax
(4)                 assert ax3 is not ax
(4)                 assert ax3 is ax4
(4)                 assert ax in fig.axes
(4)                 assert ax2 in fig.axes
(4)                 assert ax3 in fig.axes
(4)                 assert ax.name == 'rectilinear'
```

```
170: (4)                assert ax2.name == 'rectilinear'
171: (4)                assert ax3.name == 'polar'
172: (0)            def test_subplot_kwarg_collision():
173: (4)                ax1 = plt.subplot(projection='polar', theta_offset=0)
174: (4)                ax2 = plt.subplot(projection='polar', theta_offset=0)
175: (4)                assert ax1 is ax2
176: (4)                ax1.remove()
177: (4)                ax3 = plt.subplot(projection='polar', theta_offset=1)
178: (4)                assert ax1 is not ax3
179: (4)                assert ax1 not in plt.gcf().axes
180: (0)            def test_gca():
181: (4)                plt.figure()
182: (4)                ax = plt.gca()
183: (4)                ax1 = plt.gca()
184: (4)                assert ax is not None
185: (4)                assert ax1 is ax
186: (4)                plt.close()
187: (0)            def test_subplot_projection_reuse():
188: (4)                ax1 = plt.subplot(111)
189: (4)                assert ax1 is plt.gca()
190: (4)                assert ax1 is plt.subplot(111)
191: (4)                ax1.remove()
192: (4)                ax2 = plt.subplot(111, projection='polar')
193: (4)                assert ax2 is plt.gca()
194: (4)                assert ax1 not in plt.gcf().axes
195: (4)                assert ax2 is plt.subplot(111)
196: (4)                ax2.remove()
197: (4)                ax3 = plt.subplot(111, projection='rectilinear')
198: (4)                assert ax3 is plt.gca()
199: (4)                assert ax3 is not ax2
200: (4)                assert ax2 not in plt.gcf().axes
201: (0)            def test_subplot_polar_normalization():
202: (4)                ax1 = plt.subplot(111, projection='polar')
203: (4)                ax2 = plt.subplot(111, polar=True)
204: (4)                ax3 = plt.subplot(111, polar=True, projection='polar')
205: (4)                assert ax1 is ax2
206: (4)                assert ax1 is ax3
207: (4)                with pytest.raises(ValueError,
208: (23)                                   match="polar=True, yet projection='3d'"):
209: (8)                    ax2 = plt.subplot(111, polar=True, projection='3d')
210: (0)            def test_subplot_change_projection():
211: (4)                created_axes = set()
212: (4)                ax = plt.subplot()
213: (4)                created_axes.add(ax)
214: (4)                projections = ('aitoff', 'hammer', 'lambert', 'mollweide',
215: (19)                              'polar', 'rectilinear', '3d')
216: (4)                for proj in projections:
217: (8)                    ax.remove()
218: (8)                    ax = plt.subplot(projection=proj)
219: (8)                    assert ax is plt.subplot()
220: (8)                    assert ax.name == proj
221: (8)                    created_axes.add(ax)
222: (4)                assert len(created_axes) == 1 + len(projections)
223: (0)            def test_polar_second_call():
224: (4)                ln1, = plt.polar(0., 1., 'ro')
225: (4)                assert isinstance(ln1, mpl.lines.Line2D)
226: (4)                ln2, = plt.polar(1.57, .5, 'bo')
227: (4)                assert isinstance(ln2, mpl.lines.Line2D)
228: (4)                assert ln1.axes is ln2.axes
229: (0)            def test_fallback_position():
230: (4)                axref = plt.axes([0.2, 0.2, 0.5, 0.5])
231: (4)                axtest = plt.axes(position=[0.2, 0.2, 0.5, 0.5])
232: (4)                np.testing.assert_allclose(axtest.bbox.get_points(),
233: (31)                                          axref.bbox.get_points())
234: (4)                axref = plt.axes([0.2, 0.2, 0.5, 0.5])
235: (4)                axtest = plt.axes([0.2, 0.2, 0.5, 0.5], position=[0.1, 0.1, 0.8, 0.8])
236: (4)                np.testing.assert_allclose(axtest.bbox.get_points(),
237: (31)                                          axref.bbox.get_points())
238: (0)            def test_set_current_figure_via_subfigure():
```

```
239: (4)                fig1 = plt.figure()
240: (4)                subfigs = fig1.subfigures(2)
241: (4)                plt.figure()
242: (4)                assert plt.gcf() != fig1
243: (4)                current = plt.figure(subfigs[1])
244: (4)                assert plt.gcf() == fig1
245: (4)                assert current == fig1
246: (0)            def test_set_current_axes_on_subfigure():
247: (4)                fig = plt.figure()
248: (4)                subfigs = fig.subfigures(2)
249: (4)                ax = subfigs[0].subplots(1, squeeze=True)
250: (4)                subfigs[1].subplots(1, squeeze=True)
251: (4)                assert plt.gca() != ax
252: (4)                plt.sca(ax)
253: (4)                assert plt.gca() == ax
254: (0)            def test_pylab_integration():
255: (4)                IPython = pytest.importorskip("IPython")
256: (4)                mpl.testing.subprocess_run_helper(
257: (8)                    IPython.start_ipython,
258: (8)                    "--pylab",
259: (8)                    "-c",
260: (8)                    ";".join((
261: (12)                       "import matplotlib.pyplot as plt",
262: (12)                       "assert plt._REPL_DISPLAYHOOK == plt._ReplDisplayHook.IPYTHON",
263: (8)                    )),
264: (8)                    timeout=60,
265: (4)                )
266: (0)            def test_doc_pyplot_summary():
267: (4)                """Test that pyplot_summary lists all the plot functions."""
268: (4)                pyplot_docs = Path(__file__).parent /
'../../../doc/api/pyplot_summary.rst'
269: (4)                if not pyplot_docs.exists():
270: (8)                    pytest.skip("Documentation sources not available")
271: (4)                def extract_documented_functions(lines):
272: (8)                    """
273: (8)                    Return a list of all the functions that are mentioned in the
274: (8)                    autosummary blocks contained in *lines*.
275: (8)                    An autosummary block looks like this::
276: (12)                       .. autosummary::
277: (15)                          :toctree: _as_gen
278: (15)                          :template: autosummary.rst
279: (15)                          :nosignatures:
280: (15)                          plot
281: (15)                          plot_date
282: (8)                    """
283: (8)                    functions = []
284: (8)                    in_autosummary = False
285: (8)                    for line in lines:
286: (12)                       if not in_autosummary:
287: (16)                           if line.startswith(".. autosummary::"):
288: (20)                               in_autosummary = True
289: (12)                       else:
290: (16)                           if not line or line.startswith("   :"):
291: (20)                               continue
292: (16)                           if not line[0].isspace():
293: (20)                               in_autosummary = False
294: (20)                               continue
295: (16)                           functions.append(line.strip())
296: (8)                    return functions
297: (4)                lines = pyplot_docs.read_text().split("\n")
298: (4)                doc_functions = set(extract_documented_functions(lines))
299: (4)                plot_commands = set(plt._get_pyplot_commands())
300: (4)                missing = plot_commands.difference(doc_functions)
301: (4)                if missing:
302: (8)                    raise AssertionError(
303: (12)                       f"The following pyplot functions are not listed in the "
304: (12)                       f"documentation. Please add them to doc/api/pyplot_summary.rst: "
305: (12)                       f"{missing!r}")
306: (4)                extra = doc_functions.difference(plot_commands)
```

```
307: (4)                if extra:
308: (8)                    raise AssertionError(
309: (12)                       f"The following functions are listed in the pyplot documentation,
"
310: (12)                       f"but they do not exist in pyplot. "
311: (12)                       f"Please remove them from doc/api/pyplot_summary.rst: {extra!r}")
312: (0)            def test_minor_ticks():
313: (4)                plt.figure()
314: (4)                plt.plot(np.arange(1, 10))
315: (4)                tick_pos, tick_labels = plt.xticks(minor=True)
316: (4)                assert np.all(tick_labels == np.array([], dtype=np.float64))
317: (4)                assert tick_labels == []
318: (4)                plt.yticks(ticks=[3.5, 6.5], labels=["a", "b"], minor=True)
319: (4)                ax = plt.gca()
320: (4)                tick_pos = ax.get_yticks(minor=True)
321: (4)                tick_labels = ax.get_yticklabels(minor=True)
322: (4)                assert np.all(tick_pos == np.array([3.5, 6.5]))
323: (4)                assert [l.get_text() for l in tick_labels] == ['a', 'b']
324: (0)            def test_switch_backend_no_close():
325: (4)                plt.switch_backend('agg')
326: (4)                fig = plt.figure()
327: (4)                fig = plt.figure()
328: (4)                assert len(plt.get_fignums()) == 2
329: (4)                plt.switch_backend('agg')
330: (4)                assert len(plt.get_fignums()) == 2
331: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning):
332: (8)                    plt.switch_backend('svg')
333: (4)                assert len(plt.get_fignums()) == 0
334: (0)            def figure_hook_example(figure):
335: (4)                figure._test_was_here = True
336: (0)            def test_figure_hook():
337: (4)                test_rc = {
338: (8)                    'figure.hooks': ['matplotlib.tests.test_pyplot:figure_hook_example']
339: (4)                }
340: (4)                with mpl.rc_context(test_rc):
341: (8)                    fig = plt.figure()
342: (4)                assert fig._test_was_here
```

File 206 - test_quiver.py:

```
1: (0)              import platform
2: (0)              import sys
3: (0)              import numpy as np
4: (0)              import pytest
5: (0)              from matplotlib import pyplot as plt
6: (0)              from matplotlib.testing.decorators import image_comparison
7: (0)              def draw_quiver(ax, **kwargs):
8: (4)                  X, Y = np.meshgrid(np.arange(0, 2 * np.pi, 1),
9: (23)                                    np.arange(0, 2 * np.pi, 1))
10: (4)                 U = np.cos(X)
11: (4)                 V = np.sin(Y)
12: (4)                 Q = ax.quiver(U, V, **kwargs)
13: (4)                 return Q
14: (0)             @pytest.mark.skipif(platform.python_implementation() != 'CPython',
15: (20)                                reason='Requires CPython')
16: (0)             def test_quiver_memory_leak():
17: (4)                 fig, ax = plt.subplots()
18: (4)                 Q = draw_quiver(ax)
19: (4)                 ttX = Q.X
20: (4)                 Q.remove()
21: (4)                 del Q
22: (4)                 assert sys.getrefcount(ttX) == 2
23: (0)             @pytest.mark.skipif(platform.python_implementation() != 'CPython',
24: (20)                                reason='Requires CPython')
25: (0)             def test_quiver_key_memory_leak():
26: (4)                 fig, ax = plt.subplots()
27: (4)                 Q = draw_quiver(ax)
```

```
(4)                 qk = ax.quiverkey(Q, 0.5, 0.92, 2, r'$2 \frac{m}{s}$',
(22)                                  labelpos='W',
(22)                                  fontproperties={'weight': 'bold'})
(4)                 assert sys.getrefcount(qk) == 3
(4)                 qk.remove()
(4)                 assert sys.getrefcount(qk) == 2
(0)             def test_quiver_number_of_args():
(4)                 X = [1, 2]
(4)                 with pytest.raises(
(12)                        TypeError,
(12)                        match='takes from 2 to 5 positional arguments but 1 were given'):
(8)                     plt.quiver(X)
(4)                 with pytest.raises(
(12)                        TypeError,
(12)                        match='takes from 2 to 5 positional arguments but 6 were given'):
(8)                     plt.quiver(X, X, X, X, X, X)
(0)             def test_quiver_arg_sizes():
(4)                 X2 = [1, 2]
(4)                 X3 = [1, 2, 3]
(4)                 with pytest.raises(
(12)                        ValueError, match=('X and Y must be the same size, but '
(31)                                           'X.size is 2 and Y.size is 3.')):
(8)                     plt.quiver(X2, X3, X2, X2)
(4)                 with pytest.raises(
(12)                        ValueError, match=('Argument U has a size 3 which does not match '
(31)                                           '2, the number of arrow positions')):
(8)                     plt.quiver(X2, X2, X3, X2)
(4)                 with pytest.raises(
(12)                        ValueError, match=('Argument V has a size 3 which does not match '
(31)                                           '2, the number of arrow positions')):
(8)                     plt.quiver(X2, X2, X2, X3)
(4)                 with pytest.raises(
(12)                        ValueError, match=('Argument C has a size 3 which does not match '
(31)                                           '2, the number of arrow positions')):
(8)                     plt.quiver(X2, X2, X2, X2, X3)
(0)             def test_no_warnings():
(4)                 fig, ax = plt.subplots()
(4)                 X, Y = np.meshgrid(np.arange(15), np.arange(10))
(4)                 U = V = np.ones_like(X)
(4)                 phi = (np.random.rand(15, 10) - .5) * 150
(4)                 ax.quiver(X, Y, U, V, angles=phi)
(4)                 fig.canvas.draw()  # Check that no warning is emitted.
(0)             def test_zero_headlength():
(4)                 fig, ax = plt.subplots()
(4)                 X, Y = np.meshgrid(np.arange(10), np.arange(10))
(4)                 U, V = np.cos(X), np.sin(Y)
(4)                 ax.quiver(U, V, headlength=0, headaxislength=0)
(4)                 fig.canvas.draw()  # Check that no warning is emitted.
(0)             @image_comparison(['quiver_animated_test_image.png'])
(0)             def test_quiver_animate():
(4)                 fig, ax = plt.subplots()
(4)                 Q = draw_quiver(ax, animated=True)
(4)                 ax.quiverkey(Q, 0.5, 0.92, 2, r'$2 \frac{m}{s}$',
(17)                             labelpos='W', fontproperties={'weight': 'bold'})
(0)             @image_comparison(['quiver_with_key_test_image.png'])
(0)             def test_quiver_with_key():
(4)                 fig, ax = plt.subplots()
(4)                 ax.margins(0.1)
(4)                 Q = draw_quiver(ax)
(4)                 ax.quiverkey(Q, 0.5, 0.95, 2,
(17)                             r'$2\, \mathrm{m}\, \mathrm{s}^{-1}$',
(17)                             angle=-10,
(17)                             coordinates='figure',
(17)                             labelpos='W',
(17)                             fontproperties={'weight': 'bold', 'size': 'large'})
(0)             @image_comparison(['quiver_single_test_image.png'], remove_text=True)
(0)             def test_quiver_single():
(4)                 fig, ax = plt.subplots()
(4)                 ax.margins(0.1)
```

```
97: (4)                 ax.quiver([1], [1], [2], [2])
98: (0)             def test_quiver_copy():
99: (4)                 fig, ax = plt.subplots()
100: (4)                uv = dict(u=np.array([1.1]), v=np.array([2.0]))
101: (4)                q0 = ax.quiver([1], [1], uv['u'], uv['v'])
102: (4)                uv['v'][0] = 0
103: (4)                assert q0.V[0] == 2.0
104: (0)            @image_comparison(['quiver_key_pivot.png'], remove_text=True)
105: (0)            def test_quiver_key_pivot():
106: (4)                fig, ax = plt.subplots()
107: (4)                u, v = np.mgrid[0:2*np.pi:10j, 0:2*np.pi:10j]
108: (4)                q = ax.quiver(np.sin(u), np.cos(v))
109: (4)                ax.set_xlim(-2, 11)
110: (4)                ax.set_ylim(-2, 11)
111: (4)                ax.quiverkey(q, 0.5, 1, 1, 'N', labelpos='N')
112: (4)                ax.quiverkey(q, 1, 0.5, 1, 'E', labelpos='E')
113: (4)                ax.quiverkey(q, 0.5, 0, 1, 'S', labelpos='S')
114: (4)                ax.quiverkey(q, 0, 0.5, 1, 'W', labelpos='W')
115: (0)            @image_comparison(['quiver_key_xy.png'], remove_text=True)
116: (0)            def test_quiver_key_xy():
117: (4)                X = np.arange(8)
118: (4)                Y = np.zeros(8)
119: (4)                angles = X * (np.pi / 4)
120: (4)                uv = np.exp(1j * angles)
121: (4)                U = uv.real
122: (4)                V = uv.imag
123: (4)                fig, axs = plt.subplots(2)
124: (4)                for ax, angle_str in zip(axs, ('uv', 'xy')):
125: (8)                    ax.set_xlim(-1, 8)
126: (8)                    ax.set_ylim(-0.2, 0.2)
127: (8)                    q = ax.quiver(X, Y, U, V, pivot='middle',
128: (22)                                  units='xy', width=0.05,
129: (22)                                  scale=2, scale_units='xy',
130: (22)                                  angles=angle_str)
131: (8)                    for x, angle in zip((0.2, 0.5, 0.8), (0, 45, 90)):
132: (12)                       ax.quiverkey(q, X=x, Y=0.8, U=1, angle=angle, label='', color='b')
133: (0)            @image_comparison(['barbs_test_image.png'], remove_text=True)
134: (0)            def test_barbs():
135: (4)                x = np.linspace(-5, 5, 5)
136: (4)                X, Y = np.meshgrid(x, x)
137: (4)                U, V = 12*X, 12*Y
138: (4)                fig, ax = plt.subplots()
139: (4)                ax.barbs(X, Y, U, V, np.hypot(U, V), fill_empty=True, rounding=False,
140: (13)                         sizes=dict(emptybarb=0.25, spacing=0.2, height=0.3),
141: (13)                         cmap='viridis')
142: (0)            @image_comparison(['barbs_pivot_test_image.png'], remove_text=True)
143: (0)            def test_barbs_pivot():
144: (4)                x = np.linspace(-5, 5, 5)
145: (4)                X, Y = np.meshgrid(x, x)
146: (4)                U, V = 12*X, 12*Y
147: (4)                fig, ax = plt.subplots()
148: (4)                ax.barbs(X, Y, U, V, fill_empty=True, rounding=False, pivot=1.7,
149: (13)                         sizes=dict(emptybarb=0.25, spacing=0.2, height=0.3))
150: (4)                ax.scatter(X, Y, s=49, c='black')
151: (0)            @image_comparison(['barbs_test_flip.png'], remove_text=True)
152: (0)            def test_barbs_flip():
153: (4)                """Test barbs with an array for flip_barb."""
154: (4)                x = np.linspace(-5, 5, 5)
155: (4)                X, Y = np.meshgrid(x, x)
156: (4)                U, V = 12*X, 12*Y
157: (4)                fig, ax = plt.subplots()
158: (4)                ax.barbs(X, Y, U, V, fill_empty=True, rounding=False, pivot=1.7,
159: (13)                         sizes=dict(emptybarb=0.25, spacing=0.2, height=0.3),
160: (13)                         flip_barb=Y < 0)
161: (0)            def test_barb_copy():
162: (4)                fig, ax = plt.subplots()
163: (4)                u = np.array([1.1])
164: (4)                v = np.array([2.2])
165: (4)                b0 = ax.barbs([1], [1], u, v)
```

```
166: (4)                u[0] = 0
167: (4)                assert b0.u[0] == 1.1
168: (4)                v[0] = 0
169: (4)                assert b0.v[0] == 2.2
170: (0)            def test_bad_masked_sizes():
171: (4)                """Test error handling when given differing sized masked arrays."""
172: (4)                x = np.arange(3)
173: (4)                y = np.arange(3)
174: (4)                u = np.ma.array(15. * np.ones((4,)))
175: (4)                v = np.ma.array(15. * np.ones_like(u))
176: (4)                u[1] = np.ma.masked
177: (4)                v[1] = np.ma.masked
178: (4)                fig, ax = plt.subplots()
179: (4)                with pytest.raises(ValueError):
180: (8)                    ax.barbs(x, y, u, v)
181: (0)            def test_angles_and_scale():
182: (4)                fig, ax = plt.subplots()
183: (4)                X, Y = np.meshgrid(np.arange(15), np.arange(10))
184: (4)                U = V = np.ones_like(X)
185: (4)                phi = (np.random.rand(15, 10) - .5) * 150
186: (4)                ax.quiver(X, Y, U, V, angles=phi, scale_units='xy')
187: (0)            @image_comparison(['quiver_xy.png'], remove_text=True)
188: (0)            def test_quiver_xy():
189: (4)                fig, ax = plt.subplots(subplot_kw=dict(aspect='equal'))
190: (4)                ax.quiver(0, 0, 1, 1, angles='xy', scale_units='xy', scale=1)
191: (4)                ax.set_xlim(0, 1.1)
192: (4)                ax.set_ylim(0, 1.1)
193: (4)                ax.grid()
194: (0)            def test_quiverkey_angles():
195: (4)                fig, ax = plt.subplots()
196: (4)                X, Y = np.meshgrid(np.arange(2), np.arange(2))
197: (4)                U = V = angles = np.ones_like(X)
198: (4)                q = ax.quiver(X, Y, U, V, angles=angles)
199: (4)                qk = ax.quiverkey(q, 1, 1, 2, 'Label')
200: (4)                fig.canvas.draw()
201: (4)                assert len(qk.verts) == 1
202: (0)            def test_quiverkey_angles_xy_aitoff():
203: (4)                kwargs_list = [
204: (8)                    {'angles': 'xy'},
205: (8)                    {'angles': 'xy', 'scale_units': 'xy'},
206: (8)                    {'scale_units': 'xy'}
207: (4)                ]
208: (4)                for kwargs_dict in kwargs_list:
209: (8)                    x = np.linspace(-np.pi, np.pi, 11)
210: (8)                    y = np.ones_like(x) * np.pi / 6
211: (8)                    vx = np.zeros_like(x)
212: (8)                    vy = np.ones_like(x)
213: (8)                    fig = plt.figure()
214: (8)                    ax = fig.add_subplot(projection='aitoff')
215: (8)                    q = ax.quiver(x, y, vx, vy, **kwargs_dict)
216: (8)                    qk = ax.quiverkey(q, 0, 0, 1, '1 units')
217: (8)                    fig.canvas.draw()
218: (8)                    assert len(qk.verts) == 1
219: (0)            def test_quiverkey_angles_scale_units_cartesian():
220: (4)                kwargs_list = [
221: (8)                    {'angles': 'xy'},
222: (8)                    {'angles': 'xy', 'scale_units': 'xy'},
223: (8)                    {'scale_units': 'xy'}
224: (4)                ]
225: (4)                for kwargs_dict in kwargs_list:
226: (8)                    X = [0, -1, 0]
227: (8)                    Y = [0, -1, 0]
228: (8)                    U = [1, -1, 1]
229: (8)                    V = [1, -1, 0]
230: (8)                    fig, ax = plt.subplots()
231: (8)                    q = ax.quiver(X, Y, U, V, **kwargs_dict)
232: (8)                    ax.quiverkey(q, X=0.3, Y=1.1, U=1,
233: (21)                                 label='Quiver key, length = 1', labelpos='E')
234: (8)                    qk = ax.quiverkey(q, 0, 0, 1, '1 units')
```

```
235: (8)                    fig.canvas.draw()
236: (8)                    assert len(qk.verts) == 1
237: (0)            def test_quiver_setuvc_numbers():
238: (4)                """Check that it is possible to set all arrow UVC to the same numbers"""
239: (4)                fig, ax = plt.subplots()
240: (4)                X, Y = np.meshgrid(np.arange(2), np.arange(2))
241: (4)                U = V = np.ones_like(X)
242: (4)                q = ax.quiver(X, Y, U, V)
243: (4)                q.set_UVC(0, 1)
```

File 207 - test_rcparams.py:

```
1: (0)              import copy
2: (0)              import os
3: (0)              import subprocess
4: (0)              import sys
5: (0)              from unittest import mock
6: (0)              from cycler import cycler, Cycler
7: (0)              import pytest
8: (0)              import matplotlib as mpl
9: (0)              from matplotlib import _api, _c_internal_utils
10: (0)             import matplotlib.pyplot as plt
11: (0)             import matplotlib.colors as mcolors
12: (0)             import numpy as np
13: (0)             from matplotlib.rcsetup import (
14: (4)                 validate_bool,
15: (4)                 validate_color,
16: (4)                 validate_colorlist,
17: (4)                 _validate_color_or_linecolor,
18: (4)                 validate_cycler,
19: (4)                 validate_float,
20: (4)                 validate_fontstretch,
21: (4)                 validate_fontweight,
22: (4)                 validate_hatch,
23: (4)                 validate_hist_bins,
24: (4)                 validate_int,
25: (4)                 validate_markevery,
26: (4)                 validate_stringlist,
27: (4)                 validate_sketch,
28: (4)                 _validate_linestyle,
29: (4)                 _listify_validator)
30: (0)             from matplotlib.testing import subprocess_run_for_testing
31: (0)             def test_rcparams(tmp_path):
32: (4)                 mpl.rc('text', usetex=False)
33: (4)                 mpl.rc('lines', linewidth=22)
34: (4)                 usetex = mpl.rcParams['text.usetex']
35: (4)                 linewidth = mpl.rcParams['lines.linewidth']
36: (4)                 rcpath = tmp_path / 'test_rcparams.rc'
37: (4)                 rcpath.write_text('lines.linewidth: 33', encoding='utf-8')
38: (4)                 with mpl.rc_context(rc={'text.usetex': not usetex}):
39: (8)                     assert mpl.rcParams['text.usetex'] == (not usetex)
40: (4)                 assert mpl.rcParams['text.usetex'] == usetex
41: (4)                 with mpl.rc_context(fname=rcpath):
42: (8)                     assert mpl.rcParams['lines.linewidth'] == 33
43: (4)                 assert mpl.rcParams['lines.linewidth'] == linewidth
44: (4)                 with mpl.rc_context(fname=rcpath, rc={'lines.linewidth': 44}):
45: (8)                     assert mpl.rcParams['lines.linewidth'] == 44
46: (4)                 assert mpl.rcParams['lines.linewidth'] == linewidth
47: (4)                 @mpl.rc_context({'lines.linewidth': 44})
48: (4)                 def func():
49: (8)                     assert mpl.rcParams['lines.linewidth'] == 44
50: (4)                 func()
51: (4)                 func()
52: (4)                 mpl.rc_file(rcpath)
53: (4)                 assert mpl.rcParams['lines.linewidth'] == 33
54: (0)             def test_RcParams_class():
55: (4)                 rc = mpl.RcParams({'font.cursive': ['Apple Chancery',
```

```
 56: (40)                                                 'Textile',
 57: (40)                                                 'Zapf Chancery',
 58: (40)                                                 'cursive'],
 59: (23)                                     'font.family': 'sans-serif',
 60: (23)                                     'font.weight': 'normal',
 61: (23)                                     'font.size': 12})
 62: (4)                 expected_repr = """
 63: (0)             RcParams({'font.cursive': ['Apple Chancery',
 64: (27)                                       'Textile',
 65: (27)                                       'Zapf Chancery',
 66: (27)                                       'cursive'],
 67: (10)                      'font.family': ['sans-serif'],
 68: (10)                      'font.size': 12.0,
 69: (10)                      'font.weight': 'normal'})""".lstrip()
 70: (4)                 assert expected_repr == repr(rc)
 71: (4)                 expected_str = """
 72: (0)             font.cursive: ['Apple Chancery', 'Textile', 'Zapf Chancery', 'cursive']
 73: (0)             font.family: ['sans-serif']
 74: (0)             font.size: 12.0
 75: (0)             font.weight: normal""".lstrip()
 76: (4)                 assert expected_str == str(rc)
 77: (4)                 assert ['font.cursive', 'font.size'] == sorted(rc.find_all('i[vz]'))
 78: (4)                 assert ['font.family'] == list(rc.find_all('family'))
 79: (0)             def test_rcparams_update():
 80: (4)                 rc = mpl.RcParams({'figure.figsize': (3.5, 42)})
 81: (4)                 bad_dict = {'figure.figsize': (3.5, 42, 1)}
 82: (4)                 with pytest.raises(ValueError):
 83: (8)                     rc.update(bad_dict)
 84: (0)             def test_rcparams_init():
 85: (4)                 with pytest.raises(ValueError):
 86: (8)                     mpl.RcParams({'figure.figsize': (3.5, 42, 1)})
 87: (0)             def test_nargs_cycler():
 88: (4)                 from matplotlib.rcsetup import cycler as ccl
 89: (4)                 with pytest.raises(TypeError, match='3 were given'):
 90: (8)                     ccl(ccl(color=list('rgb')), 2, 3)
 91: (0)             def test_Bug_2543():
 92: (4)                 with _api.suppress_matplotlib_deprecation_warning():
 93: (8)                     with mpl.rc_context():
 94: (12)                        _copy = mpl.rcParams.copy()
 95: (12)                        for key in _copy:
 96: (16)                            mpl.rcParams[key] = _copy[key]
 97: (8)                     with mpl.rc_context():
 98: (12)                        copy.deepcopy(mpl.rcParams)
 99: (4)                 with pytest.raises(ValueError):
100: (8)                     validate_bool(None)
101: (4)                 with pytest.raises(ValueError):
102: (8)                     with mpl.rc_context():
103: (12)                        mpl.rcParams['svg.fonttype'] = True
104: (0)             legend_color_tests = [
105: (4)                 ('face', {'color': 'r'}, mcolors.to_rgba('r')),
106: (4)                 ('face', {'color': 'inherit', 'axes.facecolor': 'r'},
107: (5)                  mcolors.to_rgba('r')),
108: (4)                 ('face', {'color': 'g', 'axes.facecolor': 'r'}, mcolors.to_rgba('g')),
109: (4)                 ('edge', {'color': 'r'}, mcolors.to_rgba('r')),
110: (4)                 ('edge', {'color': 'inherit', 'axes.edgecolor': 'r'},
111: (5)                  mcolors.to_rgba('r')),
112: (4)                 ('edge', {'color': 'g', 'axes.facecolor': 'r'}, mcolors.to_rgba('g'))
113: (0)             ]
114: (0)             legend_color_test_ids = [
115: (4)                 'same facecolor',
116: (4)                 'inherited facecolor',
117: (4)                 'different facecolor',
118: (4)                 'same edgecolor',
119: (4)                 'inherited edgecolor',
120: (4)                 'different facecolor',
121: (0)             ]
122: (0)             @pytest.mark.parametrize('color_type, param_dict, target', legend_color_tests,
123: (25)                                     ids=legend_color_test_ids)
124: (0)             def test_legend_colors(color_type, param_dict, target):
```

```
(4)                 param_dict[f'legend.{color_type}color'] = param_dict.pop('color')
(4)                 get_func = f'get_{color_type}color'
(4)                 with mpl.rc_context(param_dict):
(8)                     _, ax = plt.subplots()
(8)                     ax.plot(range(3), label='test')
(8)                     leg = ax.legend()
(8)                     assert getattr(leg.legendPatch, get_func)() == target
(0)             def test_mfc_rcparams():
(4)                 mpl.rcParams['lines.markerfacecolor'] = 'r'
(4)                 ln = mpl.lines.Line2D([1, 2], [1, 2])
(4)                 assert ln.get_markerfacecolor() == 'r'
(0)             def test_mec_rcparams():
(4)                 mpl.rcParams['lines.markeredgecolor'] = 'r'
(4)                 ln = mpl.lines.Line2D([1, 2], [1, 2])
(4)                 assert ln.get_markeredgecolor() == 'r'
(0)             def test_axes_titlecolor_rcparams():
(4)                 mpl.rcParams['axes.titlecolor'] = 'r'
(4)                 _, ax = plt.subplots()
(4)                 title = ax.set_title("Title")
(4)                 assert title.get_color() == 'r'
(0)             def test_Issue_1713(tmp_path):
(4)                 rcpath = tmp_path / 'test_rcparams.rc'
(4)                 rcpath.write_text('timezone: UTC', encoding='utf-8')
(4)                 with mock.patch('locale.getpreferredencoding', return_value='UTF-32-BE'):
(8)                     rc = mpl.rc_params_from_file(rcpath, True, False)
(4)                 assert rc.get('timezone') == 'UTC'
(0)             def test_animation_frame_formats():
(4)                 for fmt in ['png', 'jpeg', 'tiff', 'raw', 'rgba', 'ppm',
(16)                            'sgi', 'bmp', 'pbm', 'svg']:
(8)                     mpl.rcParams['animation.frame_format'] = fmt
(0)             def generate_validator_testcases(valid):
(4)                 validation_tests = (
(8)                     {'validator': validate_bool,
(9)                      'success': (*((_, True) for _ in
(23)                                    ('t', 'y', 'yes', 'on', 'true', '1', 1, True)),
(21)                                  *((_, False) for _ in
(23)                                    ('f', 'n', 'no', 'off', 'false', '0', 0, False))),
(9)                      'fail': ((_, ValueError)
(18)                               for _ in ('aardvark', 2, -1, [], ))
(9)                      },
(8)                     {'validator': validate_stringlist,
(9)                      'success': (('', []),
(21)                                 ('a,b', ['a', 'b']),
(21)                                 ('aardvark', ['aardvark']),
(21)                                 ('aardvark, ', ['aardvark']),
(21)                                 ('aardvark, ,', ['aardvark']),
(21)                                 (['a', 'b'], ['a', 'b']),
(21)                                 (('a', 'b'), ['a', 'b']),
(21)                                 (iter(['a', 'b']), ['a', 'b']),
(21)                                 (np.array(['a', 'b']), ['a', 'b']),
(21)                                 ),
(9)                      'fail': ((set(), ValueError),
(18)                               (1, ValueError),
(18)                               )
(9)                      },
(8)                     {'validator': _listify_validator(validate_int, n=2),
(9)                      'success': ((_, [1, 2])
(21)                                 for _ in ('1, 2', [1.5, 2.5], [1, 2],
(31)                                           (1, 2), np.array((1, 2)))),
(9)                      'fail': ((_, ValueError)
(18)                               for _ in ('aardvark', ('a', 1),
(28)                                         (1, 2, 3)
(28)                                         ))
(9)                      },
(8)                     {'validator': _listify_validator(validate_float, n=2),
(9)                      'success': ((_, [1.5, 2.5])
(21)                                 for _ in ('1.5, 2.5', [1.5, 2.5], [1.5, 2.5],
(31)                                           (1.5, 2.5), np.array((1.5, 2.5)))),
(9)                      'fail': ((_, ValueError)
```

```
194: (18)                                for _ in ('aardvark', ('a', 1), (1, 2, 3), (None, ), None))
195: (9)                         },
196: (8)                        {'validator': validate_cycler,
197: (9)                         'success': (('cycler("color", "rgb")',
198: (22)                                     cycler("color", 'rgb')),
199: (21)                                    (cycler('linestyle', ['-', '--']),
200: (22)                                     cycler('linestyle', ['-', '--'])),
201: (21)                                    ("""(cycler("color", ["r", "g", "b"]) +
202: (26)                                         cycler("mew", [2, 3, 5]))""",
203: (22)                                     (cycler("color", 'rgb') +
204: (23)                                      cycler("markeredgewidth", [2, 3, 5]))),
205: (21)                                    ("cycler(c='rgb', lw=[1, 2, 3])",
206: (22)                                     cycler('color', 'rgb') + cycler('linewidth', [1, 2,
3])),
207: (21)                                    ("cycler('c', 'rgb') * cycler('linestyle', ['-', '--'])",
208: (22)                                     (cycler('color', 'rgb') *
209: (23)                                      cycler('linestyle', ['-', '--'])),
210: (21)                                    (cycler('ls', ['-', '--']),
211: (22)                                     cycler('linestyle', ['-', '--'])),
212: (21)                                    (cycler(mew=[2, 5]),
213: (22)                                     cycler('markeredgewidth', [2, 5])),
214: (21)                                    ),
215: (9)                         'fail': ((4, ValueError),  # Gotta be a string or Cycler object
216: (18)                                 ('cycler("bleh, [])', ValueError),  # syntax error
217: (18)                                 ('Cycler("linewidth", [1, 2, 3])',
218: (19)                                  ValueError),  # only 'cycler()' function is allowed
219: (18)                                 ("cycler('c', [j.__class__(j) for j in ['r', 'b']])",
220: (19)                                  ValueError),
221: (18)                                 ("cycler('c', [j. __class__(j) for j in ['r', 'b']])",
222: (19)                                  ValueError),
223: (18)                                 ("cycler('c', [j.\t__class__(j) for j in ['r', 'b']])",
224: (19)                                  ValueError),
225: (18)                                 ("cycler('c', [j.\u000c__class__(j) for j in ['r', 'b']])",
226: (19)                                  ValueError),
227: (18)                                 ("cycler('c', [j.__class__(j).lower() for j in ['r',
'b']])",
228: (19)                                  ValueError),
229: (18)                                 ('1 + 2', ValueError),  # doesn't produce a Cycler object
230: (18)                                 ('os.system("echo Gotcha")', ValueError),  # os not
available
231: (18)                                 ('import os', ValueError),  # should not be able to import
232: (18)                                 ('def badjuju(a): return a; badjuju(cycler("color",
"rgb"))',
233: (19)                                  ValueError),  # Should not be able to define anything
234: (18)                                 ('cycler("waka", [1, 2, 3])', ValueError),  # not a property
235: (18)                                 ('cycler(c=[1, 2, 3])', ValueError),  # invalid values
236: (18)                                 ("cycler(lw=['a', 'b', 'c'])", ValueError),  # invalid
values
237: (18)                                 (cycler('waka', [1, 3, 5]), ValueError),  # not a property
238: (18)                                 (cycler('color', ['C1', 'r', 'g']), ValueError)  # no CN
239: (18)                                 )
240: (9)                         },
241: (8)                        {'validator': validate_hatch,
242: (9)                         'success': (('--|', '--|'), ('\\oO', '\\oO'),
243: (21)                                    ('/+*/.x', '/+*/.x'), ('', '')),
244: (9)                         'fail': (('--_', ValueError),
245: (18)                                 (8, ValueError),
246: (18)                                 ('X', ValueError)),
247: (9)                         },
248: (8)                        {'validator': validate_colorlist,
249: (9)                         'success': (('r,g,b', ['r', 'g', 'b']),
250: (21)                                    (['r', 'g', 'b'], ['r', 'g', 'b']),
251: (21)                                    ('r, ,', ['r']),
252: (21)                                    ('', 'g', 'blue'], ['g', 'blue']),
253: (21)                                    ([np.array([1, 0, 0]), np.array([0, 1, 0])],
254: (21)                                    np.array([[1, 0, 0], [0, 1, 0]])),
255: (21)                                    (np.array([[1, 0, 0], [0, 1, 0]]),
256: (21)                                    np.array([[1, 0, 0], [0, 1, 0]])),
257: (21)                                    ),
```

```
 (9)                     'fail': (('fish', ValueError),
 (18)                             ),
 (9)                     },
 (8)                    {'validator': validate_color,
 (9)                     'success': (('None', 'none'),
 (21)                                ('none', 'none'),
 (21)                                ('AABBCC', '#AABBCC'),  # RGB hex code
 (21)                                ('AABBCC00', '#AABBCC00'),  # RGBA hex code
 (21)                                ('tab:blue', 'tab:blue'),  # named color
 (21)                                ('C12', 'C12'),  # color from cycle
 (21)                                ('(0, 1, 0)', (0.0, 1.0, 0.0)),  # RGB tuple
 (21)                                ((0, 1, 0), (0, 1, 0)),  # non-string version
 (21)                                ('(0, 1, 0, 1)', (0.0, 1.0, 0.0, 1.0)),  # RGBA tuple
 (21)                                ((0, 1, 0, 1), (0, 1, 0, 1)),  # non-string version
 (21)                                ),
 (9)                     'fail': (('tab:veryblue', ValueError),  # invalid name
 (18)                             ('(0, 1)', ValueError),  # tuple with length < 3
 (18)                             ('(0, 1, 0, 1, 0)', ValueError),  # tuple with length > 4
 (18)                             ('(0, 1, none)', ValueError),  # cannot cast none to float
 (18)                             ('(0, 1, "0.5")', ValueError),  # last one not a float
 (18)                             ),
 (9)                     },
 (8)                    {'validator': _validate_color_or_linecolor,
 (9)                     'success': (('linecolor', 'linecolor'),
 (21)                                ('markerfacecolor', 'markerfacecolor'),
 (21)                                ('mfc', 'markerfacecolor'),
 (21)                                ('markeredgecolor', 'markeredgecolor'),
 (21)                                ('mec', 'markeredgecolor')
 (21)                                ),
 (9)                     'fail': (('line', ValueError),
 (18)                             ('marker', ValueError)
 (18)                             )
 (9)                     },
 (8)                    {'validator': validate_hist_bins,
 (9)                     'success': (('auto', 'auto'),
 (21)                                ('fd', 'fd'),
 (21)                                ('10', 10),
 (21)                                ('1, 2, 3', [1, 2, 3]),
 (21)                                ([1, 2, 3], [1, 2, 3]),
 (21)                                (np.arange(15), np.arange(15))
 (21)                                ),
 (9)                     'fail': (('aardvark', ValueError),
 (18)                             )
 (9)                     },
 (8)                    {'validator': validate_markevery,
 (9)                     'success': ((None, None),
 (21)                                (1, 1),
 (21)                                (0.1, 0.1),
 (21)                                ((1, 1), (1, 1)),
 (21)                                ((0.1, 0.1), (0.1, 0.1)),
 (21)                                ([1, 2, 3], [1, 2, 3]),
 (21)                                (slice(2), slice(None, 2, None)),
 (21)                                (slice(1, 2, 3), slice(1, 2, 3))
 (21)                                ),
 (9)                     'fail': (((1, 2, 3), TypeError),
 (18)                             ([1, 2, 0.3], TypeError),
 (18)                             (['a', 2, 3], TypeError),
 (18)                             ([1, 2, 'a'], TypeError),
 (18)                             ((0.1, 0.2, 0.3), TypeError),
 (18)                             ((0.1, 2, 3), TypeError),
 (18)                             ((1, 0.2, 0.3), TypeError),
 (18)                             ((1, 0.1), TypeError),
 (18)                             ((0.1, 1), TypeError),
 (18)                             (('abc'), TypeError),
 (18)                             ((1, 'a'), TypeError),
 (18)                             ((0.1, 'b'), TypeError),
 (18)                             (('a', 1), TypeError),
 (18)                             (('a', 0.1), TypeError),
 (18)                             ('abc', TypeError),
```

```
(18)                            ('a', TypeError),
(18)                            (object(), TypeError)
(18)                            )
(9)                      },
(8)                     {'validator': _validate_linestyle,
(9)                      'success': (('-', '-'), ('solid', 'solid'),
(21)                                 ('--', '--'), ('dashed', 'dashed'),
(21)                                 ('-.', '-.'), ('dashdot', 'dashdot'),
(21)                                 (':', ':'), ('dotted', 'dotted'),
(21)                                 ('', ''), (' ', ' '),
(21)                                 ('None', 'none'), ('none', 'none'),
(21)                                 ('DoTtEd', 'dotted'),  # case-insensitive
(21)                                 ('1, 3', (0, (1, 3))),
(21)                                 ([1.23, 456], (0, [1.23, 456.0])),
(21)                                 ([1, 2, 3, 4], (0, [1.0, 2.0, 3.0, 4.0])),
(21)                                 ((0, [1, 2]), (0, [1, 2])),
(21)                                 ((-1, [1, 2]), (-1, [1, 2])),
(21)                                 ),
(9)                      'fail': (('aardvark', ValueError),  # not a valid string
(18)                            (b'dotted', ValueError),
(18)                            ('dotted'.encode('utf-16'), ValueError),
(18)                            ([1, 2, 3], ValueError),  # sequence with odd length
(18)                            (1.23, ValueError),  # not a sequence
(18)                            (("a", [1, 2]), ValueError),  # wrong explicit offset
(18)                            ((None, [1, 2]), ValueError),  # wrong explicit offset
(18)                            ((1, [1, 2, 3]), ValueError),  # odd length sequence
(18)                            (([1, 2], 1), ValueError),  # inverted offset/onoff
(18)                            )
(9)                      },
(4)                 )
(4)                 for validator_dict in validation_tests:
(8)                     validator = validator_dict['validator']
(8)                     if valid:
(12)                        for arg, target in validator_dict['success']:
(16)                            yield validator, arg, target
(8)                     else:
(12)                        for arg, error_type in validator_dict['fail']:
(16)                            yield validator, arg, error_type
(0)             @pytest.mark.parametrize('validator, arg, target',
(25)                                     generate_validator_testcases(True))
(0)             def test_validator_valid(validator, arg, target):
(4)                 res = validator(arg)
(4)                 if isinstance(target, np.ndarray):
(8)                     np.testing.assert_equal(res, target)
(4)                 elif not isinstance(target, Cycler):
(8)                     assert res == target
(4)                 else:
(8)                     assert list(res) == list(target)
(0)             @pytest.mark.parametrize('validator, arg, exception_type',
(25)                                     generate_validator_testcases(False))
(0)             def test_validator_invalid(validator, arg, exception_type):
(4)                 with pytest.raises(exception_type):
(8)                     validator(arg)
(0)             @pytest.mark.parametrize('weight, parsed_weight', [
(4)                 ('bold', 'bold'),
(4)                 ('BOLD', ValueError),  # weight is case-sensitive
(4)                 (100, 100),
(4)                 ('100', 100),
(4)                 (np.array(100), 100),
(4)                 (20.6, 20),
(4)                 ('20.6', ValueError),
(4)                 ([100], ValueError),
(0)             ])
(0)             def test_validate_fontweight(weight, parsed_weight):
(4)                 if parsed_weight is ValueError:
(8)                     with pytest.raises(ValueError):
(12)                        validate_fontweight(weight)
(4)                 else:
(8)                     assert validate_fontweight(weight) == parsed_weight
```

```
(0)  @pytest.mark.parametrize('stretch, parsed_stretch', [
(4)      ('expanded', 'expanded'),
(4)      ('EXPANDED', ValueError),  # stretch is case-sensitive
(4)      (100, 100),
(4)      ('100', 100),
(4)      (np.array(100), 100),
(4)      (20.6, 20),
(4)      ('20.6', ValueError),
(4)      ([100], ValueError),
(0)  ])
(0)  def test_validate_fontstretch(stretch, parsed_stretch):
(4)      if parsed_stretch is ValueError:
(8)          with pytest.raises(ValueError):
(12)             validate_fontstretch(stretch)
(4)      else:
(8)          assert validate_fontstretch(stretch) == parsed_stretch
(0)  def test_keymaps():
(4)      key_list = [k for k in mpl.rcParams if 'keymap' in k]
(4)      for k in key_list:
(8)          assert isinstance(mpl.rcParams[k], list)
(0)  def test_no_backend_reset_rccontext():
(4)      assert mpl.rcParams['backend'] != 'module://aardvark'
(4)      with mpl.rc_context():
(8)          mpl.rcParams['backend'] = 'module://aardvark'
(4)      assert mpl.rcParams['backend'] == 'module://aardvark'
(0)  def test_rcparams_reset_after_fail():
(4)      with mpl.rc_context(rc={'text.usetex': False}):
(8)          assert mpl.rcParams['text.usetex'] is False
(8)          with pytest.raises(KeyError):
(12)             with mpl.rc_context(rc={'text.usetex': True, 'test.blah': True}):
(16)                 pass
(8)          assert mpl.rcParams['text.usetex'] is False
(0)  @pytest.mark.skipif(sys.platform != "linux", reason="Linux only")
(0)  def test_backend_fallback_headless(tmp_path):
(4)      env = {**os.environ,
(11)            "DISPLAY": "", "WAYLAND_DISPLAY": "",
(11)            "MPLBACKEND": "", "MPLCONFIGDIR": str(tmp_path)}
(4)      with pytest.raises(subprocess.CalledProcessError):
(8)          subprocess_run_for_testing(
(12)             [sys.executable, "-c",
(13)              "import matplotlib;"
(13)              "matplotlib.use('tkagg');"
(13)              "import matplotlib.pyplot;"
(13)              "matplotlib.pyplot.plot(42);"
(13)              ],
(12)             env=env, check=True, stderr=subprocess.DEVNULL)
(0)  @pytest.mark.skipif(
(4)      sys.platform == "linux" and not _c_internal_utils.display_is_valid(),
(4)      reason="headless")
(0)  def test_backend_fallback_headful(tmp_path):
(4)      pytest.importorskip("tkinter")
(4)      env = {**os.environ, "MPLBACKEND": "", "MPLCONFIGDIR": str(tmp_path)}
(4)      backend = subprocess_run_for_testing(
(8)          [sys.executable, "-c",
(9)           "import matplotlib as mpl; "
(9)           "sentinel = mpl.rcsetup._auto_backend_sentinel; "
(9)           "assert mpl.RcParams({'backend': sentinel})['backend'] == sentinel; "
(9)           "assert mpl.rcParams._get('backend') == sentinel; "
(9)           "import matplotlib.pyplot; "
(9)           "print(matplotlib.get_backend())"],
(8)          env=env, text=True, check=True, capture_output=True).stdout
(4)      assert backend.strip().lower() != "agg"
(0)  def test_deprecation(monkeypatch):
(4)      monkeypatch.setitem(
(8)          mpl._deprecated_map, "patch.linewidth",
(8)          ("0.0", "axes.linewidth", lambda old: 2 * old, lambda new: new / 2))
(4)      with pytest.warns(mpl.MatplotlibDeprecationWarning):
(8)          assert mpl.rcParams["patch.linewidth"] \
(12)             == mpl.rcParams["axes.linewidth"] / 2
```

```
465: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning):
466: (8)                    mpl.rcParams["patch.linewidth"] = 1
467: (4)                assert mpl.rcParams["axes.linewidth"] == 2
468: (4)                monkeypatch.setitem(
469: (8)                    mpl._deprecated_ignore_map, "patch.edgecolor",
470: (8)                    ("0.0", "axes.edgecolor"))
471: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning):
472: (8)                    assert mpl.rcParams["patch.edgecolor"] \
473: (12)                       == mpl.rcParams["axes.edgecolor"]
474: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning):
475: (8)                    mpl.rcParams["patch.edgecolor"] = "#abcd"
476: (4)                assert mpl.rcParams["axes.edgecolor"] != "#abcd"
477: (4)                monkeypatch.setitem(
478: (8)                    mpl._deprecated_ignore_map, "patch.force_edgecolor",
479: (8)                    ("0.0", None))
480: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning):
481: (8)                    assert mpl.rcParams["patch.force_edgecolor"] is None
482: (4)                monkeypatch.setitem(
483: (8)                    mpl._deprecated_remain_as_none, "svg.hashsalt",
484: (8)                    ("0.0",))
485: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning):
486: (8)                    mpl.rcParams["svg.hashsalt"] = "foobar"
487: (4)                assert mpl.rcParams["svg.hashsalt"] == "foobar"  # Doesn't warn.
488: (4)                mpl.rcParams["svg.hashsalt"] = None  # Doesn't warn.
489: (4)                mpl.rcParams.update(mpl.rcParams.copy())  # Doesn't warn.
490: (0)            @pytest.mark.parametrize("value", [
491: (4)                "best",
492: (4)                1,
493: (4)                "1",
494: (4)                (0.9, .7),
495: (4)                (-0.9, .7),
496: (4)                "(0.9, .7)"
497: (0)            ])
498: (0)            def test_rcparams_legend_loc(value):
499: (4)                mpl.rcParams["legend.loc"] = value
500: (0)            @pytest.mark.parametrize("value", [
501: (4)                "best",
502: (4)                1,
503: (4)                (0.9, .7),
504: (4)                (-0.9, .7),
505: (0)            ])
506: (0)            def test_rcparams_legend_loc_from_file(tmp_path, value):
507: (4)                rc_path = tmp_path / "matplotlibrc"
508: (4)                rc_path.write_text(f"legend.loc: {value}")
509: (4)                with mpl.rc_context(fname=rc_path):
510: (8)                    assert mpl.rcParams["legend.loc"] == value
511: (0)            @pytest.mark.parametrize("value", [(1, 2, 3), '1, 2, 3', '(1, 2, 3)'])
512: (0)            def test_validate_sketch(value):
513: (4)                mpl.rcParams["path.sketch"] = value
514: (4)                assert mpl.rcParams["path.sketch"] == (1, 2, 3)
515: (4)                assert validate_sketch(value) == (1, 2, 3)
516: (0)            @pytest.mark.parametrize("value", [1, '1', '1 2 3'])
517: (0)            def test_validate_sketch_error(value):
518: (4)                with pytest.raises(ValueError, match="scale, length, randomness"):
519: (8)                    validate_sketch(value)
520: (4)                with pytest.raises(ValueError, match="scale, length, randomness"):
521: (8)                    mpl.rcParams["path.sketch"] = value
522: (0)            @pytest.mark.parametrize("value", ['1, 2, 3', '(1,2,3)'])
523: (0)            def test_rcparams_path_sketch_from_file(tmp_path, value):
524: (4)                rc_path = tmp_path / "matplotlibrc"
525: (4)                rc_path.write_text(f"path.sketch: {value}")
526: (4)                with mpl.rc_context(fname=rc_path):
527: (8)                    assert mpl.rcParams["path.sketch"] == (1, 2, 3)

----------------------------------------

File 208 - test_sankey.py:

1: (0)              import pytest
```

```
(0)  from numpy.testing import assert_allclose, assert_array_equal
(0)  from matplotlib.sankey import Sankey
(0)  from matplotlib.testing.decorators import check_figures_equal
(0)  def test_sankey():
(4)      sankey = Sankey()
(4)      sankey.add()
(0)  def test_label():
(4)      s = Sankey(flows=[0.25], labels=['First'], orientations=[-1])
(4)      assert s.diagrams[0].texts[0].get_text() == 'First\n0.25'
(0)  def test_format_using_callable():
(4)      def show_three_decimal_places(value):
(8)          return f'{value:.3f}'
(4)      s = Sankey(flows=[0.25], labels=['First'], orientations=[-1],
(15)                 format=show_three_decimal_places)
(4)      assert s.diagrams[0].texts[0].get_text() == 'First\n0.250'
(0)  @pytest.mark.parametrize('kwargs, msg', (
(4)      ({'gap': -1}, "'gap' is negative"),
(4)      ({'gap': 1, 'radius': 2}, "'radius' is greater than 'gap'"),
(4)      ({'head_angle': -1}, "'head_angle' is negative"),
(4)      ({'tolerance': -1}, "'tolerance' is negative"),
(4)      ({'flows': [1, -1], 'orientations': [-1, 0, 1]},
(5)       r"The shapes of 'flows' \(2,\) and 'orientations'"),
(4)      ({'flows': [1, -1], 'labels': ['a', 'b', 'c']},
(5)       r"The shapes of 'flows' \(2,\) and 'labels'"),
(4)      ))
(0)  def test_sankey_errors(kwargs, msg):
(4)      with pytest.raises(ValueError, match=msg):
(8)          Sankey(**kwargs)
(0)  @pytest.mark.parametrize('kwargs, msg', (
(4)      ({'trunklength': -1}, "'trunklength' is negative"),
(4)      ({'flows': [0.2, 0.3], 'prior': 0}, "The scaled sum of the connected"),
(4)      ({'prior': -1}, "The index of the prior diagram is negative"),
(4)      ({'prior': 1}, "The index of the prior diagram is 1"),
(4)      ({'connect': (-1, 1), 'prior': 0}, "At least one of the connection"),
(4)      ({'connect': (2, 1), 'prior': 0}, "The connection index to the source"),
(4)      ({'connect': (1, 3), 'prior': 0}, "The connection index to this dia"),
(4)      ({'connect': (1, 1), 'prior': 0, 'flows': [-0.2, 0.2],
(6)        'orientations': [2]}, "The value of orientations"),
(4)      ({'connect': (1, 1), 'prior': 0, 'flows': [-0.2, 0.2],
(6)        'pathlengths': [2]}, "The lengths of 'flows'"),
(4)      ))
(0)  def test_sankey_add_errors(kwargs, msg):
(4)      sankey = Sankey()
(4)      with pytest.raises(ValueError, match=msg):
(8)          sankey.add(flows=[0.2, -0.2])
(8)          sankey.add(**kwargs)
(0)  def test_sankey2():
(4)      s = Sankey(flows=[0.25, -0.25, 0.5, -0.5], labels=['Foo'],
(15)                 orientations=[-1], unit='Bar')
(4)      sf = s.finish()
(4)      assert_array_equal(sf[0].flows, [0.25, -0.25, 0.5, -0.5])
(4)      assert sf[0].angles == [1, 3, 1, 3]
(4)      assert all([text.get_text()[0:3] == 'Foo' for text in sf[0].texts])
(4)      assert all([text.get_text()[-3:] == 'Bar' for text in sf[0].texts])
(4)      assert sf[0].text.get_text() == ''
(4)      assert_allclose(sf[0].tips,
(20)                     [(-1.375, -0.52011255),
(21)                      (1.375, -0.75506044),
(21)                      (-0.75, -0.41522509),
(21)                      (0.75, -0.8599479)])
(4)      s = Sankey(flows=[0.25, -0.25, 0, 0.5, -0.5], labels=['Foo'],
(15)                 orientations=[-1], unit='Bar')
(4)      sf = s.finish()
(4)      assert_array_equal(sf[0].flows, [0.25, -0.25, 0, 0.5, -0.5])
(4)      assert sf[0].angles == [1, 3, None, 1, 3]
(4)      assert_allclose(sf[0].tips,
(20)                     [(-1.375, -0.52011255),
(21)                      (1.375, -0.75506044),
(21)                      (0, 0),
```

```
71: (21)                                (-0.75, -0.41522509),
72: (21)                                (0.75, -0.8599479)])
73: (0)             @check_figures_equal(extensions=['png'])
74: (0)             def test_sankey3(fig_test, fig_ref):
75: (4)                 ax_test = fig_test.gca()
76: (4)                 s_test = Sankey(ax=ax_test, flows=[0.25, -0.25, -0.25, 0.25, 0.5, -0.5],
77: (20)                                orientations=[1, -1, 1, -1, 0, 0])
78: (4)                 s_test.finish()
79: (4)                 ax_ref = fig_ref.gca()
80: (4)                 s_ref = Sankey(ax=ax_ref)
81: (4)                 s_ref.add(flows=[0.25, -0.25, -0.25, 0.25, 0.5, -0.5],
82: (14)                          orientations=[1, -1, 1, -1, 0, 0])
83: (4)                 s_ref.finish()

----------------------------------------

File 209 - test_scale.py:

1: (0)              import copy
2: (0)              import matplotlib.pyplot as plt
3: (0)              from matplotlib.scale import (
4: (4)                  AsinhScale, AsinhTransform,
5: (4)                  LogTransform, InvertedLogTransform,
6: (4)                  SymmetricalLogTransform)
7: (0)              import matplotlib.scale as mscale
8: (0)              from matplotlib.ticker import AsinhLocator, LogFormatterSciNotation
9: (0)              from matplotlib.testing.decorators import check_figures_equal,
image_comparison
10: (0)             import numpy as np
11: (0)             from numpy.testing import assert_allclose
12: (0)             import io
13: (0)             import pytest
14: (0)             @check_figures_equal()
15: (0)             def test_log_scales(fig_test, fig_ref):
16: (4)                 ax_test = fig_test.add_subplot(122, yscale='log', xscale='symlog')
17: (4)                 ax_test.axvline(24.1)
18: (4)                 ax_test.axhline(24.1)
19: (4)                 xlim = ax_test.get_xlim()
20: (4)                 ylim = ax_test.get_ylim()
21: (4)                 ax_ref = fig_ref.add_subplot(122, yscale='log', xscale='symlog')
22: (4)                 ax_ref.set(xlim=xlim, ylim=ylim)
23: (4)                 ax_ref.plot([24.1, 24.1], ylim, 'b')
24: (4)                 ax_ref.plot(xlim, [24.1, 24.1], 'b')
25: (0)             def test_symlog_mask_nan():
26: (4)                 slt = SymmetricalLogTransform(10, 2, 1)
27: (4)                 slti = slt.inverted()
28: (4)                 x = np.arange(-1.5, 5, 0.5)
29: (4)                 out = slti.transform_non_affine(slt.transform_non_affine(x))
30: (4)                 assert_allclose(out, x)
31: (4)                 assert type(out) is type(x)
32: (4)                 x[4] = np.nan
33: (4)                 out = slti.transform_non_affine(slt.transform_non_affine(x))
34: (4)                 assert_allclose(out, x)
35: (4)                 assert type(out) is type(x)
36: (4)                 x = np.ma.array(x)
37: (4)                 out = slti.transform_non_affine(slt.transform_non_affine(x))
38: (4)                 assert_allclose(out, x)
39: (4)                 assert type(out) is type(x)
40: (4)                 x[3] = np.ma.masked
41: (4)                 out = slti.transform_non_affine(slt.transform_non_affine(x))
42: (4)                 assert_allclose(out, x)
43: (4)                 assert type(out) is type(x)
44: (0)             @image_comparison(['logit_scales.png'], remove_text=True)
45: (0)             def test_logit_scales():
46: (4)                 fig, ax = plt.subplots()
47: (4)                 x = np.array([0.001, 0.003, 0.01, 0.03, 0.1, 0.2, 0.3, 0.4, 0.5,
48: (18)                              0.6, 0.7, 0.8, 0.9, 0.97, 0.99, 0.997, 0.999])
49: (4)                 y = 1.0 / x
50: (4)                 ax.plot(x, y)
```

```
(4)                 ax.set_xscale('logit')
(4)                 ax.grid(True)
(4)                 bbox = ax.get_tightbbox(fig.canvas.get_renderer())
(4)                 assert np.isfinite(bbox.x0)
(4)                 assert np.isfinite(bbox.y0)
(0)             def test_log_scatter():
(4)                 """Issue #1799"""
(4)                 fig, ax = plt.subplots(1)
(4)                 x = np.arange(10)
(4)                 y = np.arange(10) - 1
(4)                 ax.scatter(x, y)
(4)                 buf = io.BytesIO()
(4)                 fig.savefig(buf, format='pdf')
(4)                 buf = io.BytesIO()
(4)                 fig.savefig(buf, format='eps')
(4)                 buf = io.BytesIO()
(4)                 fig.savefig(buf, format='svg')
(0)             def test_logscale_subs():
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_yscale('log', subs=np.array([2, 3, 4]))
(4)                 fig.canvas.draw()
(0)             @image_comparison(['logscale_mask.png'], remove_text=True)
(0)             def test_logscale_mask():
(4)                 xs = np.linspace(0, 50, 1001)
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(np.exp(-xs**2))
(4)                 fig.canvas.draw()
(4)                 ax.set(yscale="log")
(0)             def test_extra_kwargs_raise():
(4)                 fig, ax = plt.subplots()
(4)                 for scale in ['linear', 'log', 'symlog']:
(8)                     with pytest.raises(TypeError):
(12)                        ax.set_yscale(scale, foo='mask')
(0)             def test_logscale_invert_transform():
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_yscale('log')
(4)                 tform = (ax.transAxes + ax.transData.inverted()).inverted()
(4)                 inverted_transform = LogTransform(base=2).inverted()
(4)                 assert isinstance(inverted_transform, InvertedLogTransform)
(4)                 assert inverted_transform.base == 2
(0)             def test_logscale_transform_repr():
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_yscale('log')
(4)                 repr(ax.transData)
(4)                 repr(LogTransform(10, nonpositive='clip'))
(0)             @image_comparison(['logscale_nonpos_values.png'],
(18)                              remove_text=True, tol=0.02, style='mpl20')
(0)             def test_logscale_nonpos_values():
(4)                 np.random.seed(19680801)
(4)                 xs = np.random.normal(size=int(1e3))
(4)                 fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
(4)                 ax1.hist(xs, range=(-5, 5), bins=10)
(4)                 ax1.set_yscale('log')
(4)                 ax2.hist(xs, range=(-5, 5), bins=10)
(4)                 ax2.set_yscale('log', nonpositive='mask')
(4)                 xdata = np.arange(0, 10, 0.01)
(4)                 ydata = np.exp(-xdata)
(4)                 edata = 0.2*(10-xdata)*np.cos(5*xdata)*np.exp(-xdata)
(4)                 ax3.fill_between(xdata, ydata - edata, ydata + edata)
(4)                 ax3.set_yscale('log')
(4)                 x = np.logspace(-1, 1)
(4)                 y = x ** 3
(4)                 yerr = x**2
(4)                 ax4.errorbar(x, y, yerr=yerr)
(4)                 ax4.set_yscale('log')
(4)                 ax4.set_xscale('log')
(0)             def test_invalid_log_lims():
(4)                 fig, ax = plt.subplots()
(4)                 ax.scatter(range(0, 4), range(0, 4))
```

```
(4)                 ax.set_xscale('log')
(4)                 original_xlim = ax.get_xlim()
(4)                 with pytest.warns(UserWarning):
(8)                     ax.set_xlim(left=0)
(4)                 assert ax.get_xlim() == original_xlim
(4)                 with pytest.warns(UserWarning):
(8)                     ax.set_xlim(right=-1)
(4)                 assert ax.get_xlim() == original_xlim
(4)                 ax.set_yscale('log')
(4)                 original_ylim = ax.get_ylim()
(4)                 with pytest.warns(UserWarning):
(8)                     ax.set_ylim(bottom=0)
(4)                 assert ax.get_ylim() == original_ylim
(4)                 with pytest.warns(UserWarning):
(8)                     ax.set_ylim(top=-1)
(4)                 assert ax.get_ylim() == original_ylim
(0)             @image_comparison(['function_scales.png'], remove_text=True, style='mpl20')
(0)             def test_function_scale():
(4)                 def inverse(x):
(8)                     return x**2
(4)                 def forward(x):
(8)                     return x**(1/2)
(4)                 fig, ax = plt.subplots()
(4)                 x = np.arange(1, 1000)
(4)                 ax.plot(x, x)
(4)                 ax.set_xscale('function', functions=(forward, inverse))
(4)                 ax.set_xlim(1, 1000)
(0)             def test_pass_scale():
(4)                 fig, ax = plt.subplots()
(4)                 scale = mscale.LogScale(axis=None)
(4)                 ax.set_xscale(scale)
(4)                 scale = mscale.LogScale(axis=None)
(4)                 ax.set_yscale(scale)
(4)                 assert ax.xaxis.get_scale() == 'log'
(4)                 assert ax.yaxis.get_scale() == 'log'
(0)             def test_scale_deepcopy():
(4)                 sc = mscale.LogScale(axis='x', base=10)
(4)                 sc2 = copy.deepcopy(sc)
(4)                 assert str(sc.get_transform()) == str(sc2.get_transform())
(4)                 assert sc._transform is not sc2._transform
(0)             class TestAsinhScale:
(4)                 def test_transforms(self):
(8)                     a0 = 17.0
(8)                     a = np.linspace(-50, 50, 100)
(8)                     forward = AsinhTransform(a0)
(8)                     inverse = forward.inverted()
(8)                     invinv = inverse.inverted()
(8)                     a_forward = forward.transform_non_affine(a)
(8)                     a_inverted = inverse.transform_non_affine(a_forward)
(8)                     assert_allclose(a_inverted, a)
(8)                     a_invinv = invinv.transform_non_affine(a)
(8)                     assert_allclose(a_invinv, a0 * np.arcsinh(a / a0))
(4)                 def test_init(self):
(8)                     fig, ax = plt.subplots()
(8)                     s = AsinhScale(axis=None, linear_width=23.0)
(8)                     assert s.linear_width == 23
(8)                     assert s._base == 10
(8)                     assert s._subs == (2, 5)
(8)                     tx = s.get_transform()
(8)                     assert isinstance(tx, AsinhTransform)
(8)                     assert tx.linear_width == s.linear_width
(4)                 def test_base_init(self):
(8)                     fig, ax = plt.subplots()
(8)                     s3 = AsinhScale(axis=None, base=3)
(8)                     assert s3._base == 3
(8)                     assert s3._subs == (2,)
(8)                     s7 = AsinhScale(axis=None, base=7, subs=(2, 4))
(8)                     assert s7._base == 7
(8)                     assert s7._subs == (2, 4)
```

```
189: (4)                def test_fmtloc(self):
190: (8)                    class DummyAxis:
191: (12)                       def __init__(self):
192: (16)                           self.fields = {}
193: (12)                       def set(self, **kwargs):
194: (16)                           self.fields.update(**kwargs)
195: (12)                       def set_major_formatter(self, f):
196: (16)                           self.fields['major_formatter'] = f
197: (8)                    ax0 = DummyAxis()
198: (8)                    s0 = AsinhScale(axis=ax0, base=0)
199: (8)                    s0.set_default_locators_and_formatters(ax0)
200: (8)                    assert isinstance(ax0.fields['major_locator'], AsinhLocator)
201: (8)                    assert isinstance(ax0.fields['major_formatter'], str)
202: (8)                    ax5 = DummyAxis()
203: (8)                    s7 = AsinhScale(axis=ax5, base=5)
204: (8)                    s7.set_default_locators_and_formatters(ax5)
205: (8)                    assert isinstance(ax5.fields['major_locator'], AsinhLocator)
206: (8)                    assert isinstance(ax5.fields['major_formatter'],
207: (26)                                     LogFormatterSciNotation)
208: (4)                def test_bad_scale(self):
209: (8)                    fig, ax = plt.subplots()
210: (8)                    with pytest.raises(ValueError):
211: (12)                       AsinhScale(axis=None, linear_width=0)
212: (8)                    with pytest.raises(ValueError):
213: (12)                       AsinhScale(axis=None, linear_width=-1)
214: (8)                    s0 = AsinhScale(axis=None, )
215: (8)                    s1 = AsinhScale(axis=None, linear_width=3.0)
```

File 210 - test_simplification.py:

```
1: (0)              import base64
2: (0)              import io
3: (0)              import platform
4: (0)              import numpy as np
5: (0)              from numpy.testing import assert_array_almost_equal, assert_array_equal
6: (0)              import pytest
7: (0)              from matplotlib.testing.decorators import (
8: (4)                  check_figures_equal, image_comparison, remove_ticks_and_titles)
9: (0)              import matplotlib.pyplot as plt
10: (0)             from matplotlib import patches, transforms
11: (0)             from matplotlib.path import Path
12: (0)             @image_comparison(['clipping'], remove_text=True)
13: (0)             def test_clipping():
14: (4)                 t = np.arange(0.0, 2.0, 0.01)
15: (4)                 s = np.sin(2*np.pi*t)
16: (4)                 fig, ax = plt.subplots()
17: (4)                 ax.plot(t, s, linewidth=1.0)
18: (4)                 ax.set_ylim((-0.20, -0.28))
19: (0)             @image_comparison(['overflow'], remove_text=True,
20: (18)                            tol=0.007 if platform.machine() == 'arm64' else 0)
21: (0)             def test_overflow():
22: (4)                 x = np.array([1.0, 2.0, 3.0, 2.0e5])
23: (4)                 y = np.arange(len(x))
24: (4)                 fig, ax = plt.subplots()
25: (4)                 ax.plot(x, y)
26: (4)                 ax.set_xlim(2, 6)
27: (0)             @image_comparison(['clipping_diamond'], remove_text=True)
28: (0)             def test_diamond():
29: (4)                 x = np.array([0.0, 1.0, 0.0, -1.0, 0.0])
30: (4)                 y = np.array([1.0, 0.0, -1.0, 0.0, 1.0])
31: (4)                 fig, ax = plt.subplots()
32: (4)                 ax.plot(x, y)
33: (4)                 ax.set_xlim(-0.6, 0.6)
34: (4)                 ax.set_ylim(-0.6, 0.6)
35: (0)             def test_clipping_out_of_bounds():
36: (4)                 path = Path([(0, 0), (1, 2), (2, 1)])
37: (4)                 simplified = path.cleaned(clip=(10, 10, 20, 20))
```

```
 38: (4)                assert_array_equal(simplified.vertices, [(0, 0)])
 39: (4)                assert simplified.codes == [Path.STOP]
 40: (4)                path = Path([(0, 0), (1, 2), (2, 1)],
 41: (16)                           [Path.MOVETO, Path.LINETO, Path.LINETO])
 42: (4)                simplified = path.cleaned(clip=(10, 10, 20, 20))
 43: (4)                assert_array_equal(simplified.vertices, [(0, 0)])
 44: (4)                assert simplified.codes == [Path.STOP]
 45: (4)                path = Path([(0, 0), (1, 2), (2, 3)],
 46: (16)                           [Path.MOVETO, Path.CURVE3, Path.CURVE3])
 47: (4)                simplified = path.cleaned()
 48: (4)                simplified_clipped = path.cleaned(clip=(10, 10, 20, 20))
 49: (4)                assert_array_equal(simplified.vertices, simplified_clipped.vertices)
 50: (4)                assert_array_equal(simplified.codes, simplified_clipped.codes)
 51: (0)            def test_noise():
 52: (4)                np.random.seed(0)
 53: (4)                x = np.random.uniform(size=50000) * 50
 54: (4)                fig, ax = plt.subplots()
 55: (4)                p1 = ax.plot(x, solid_joinstyle='round', linewidth=2.0)
 56: (4)                fig.canvas.draw()
 57: (4)                path = p1[0].get_path()
 58: (4)                transform = p1[0].get_transform()
 59: (4)                path = transform.transform_path(path)
 60: (4)                simplified = path.cleaned(simplify=True)
 61: (4)                assert simplified.vertices.size == 25512
 62: (0)            def test_antiparallel_simplification():
 63: (4)                def _get_simplified(x, y):
 64: (8)                    fig, ax = plt.subplots()
 65: (8)                    p1 = ax.plot(x, y)
 66: (8)                    path = p1[0].get_path()
 67: (8)                    transform = p1[0].get_transform()
 68: (8)                    path = transform.transform_path(path)
 69: (8)                    simplified = path.cleaned(simplify=True)
 70: (8)                    simplified = transform.inverted().transform_path(simplified)
 71: (8)                    return simplified
 72: (4)                x = [0, 0, 0, 0, 0, 1]
 73: (4)                y = [.5, 1, -1, 1, 2, .5]
 74: (4)                simplified = _get_simplified(x, y)
 75: (4)                assert_array_almost_equal([[0., 0.5],
 76: (31)                                          [0., -1.],
 77: (31)                                          [0., 2.],
 78: (31)                                          [1., 0.5]],
 79: (30)                                         simplified.vertices[:-2, :])
 80: (4)                x = [0, 0,  0, 0, 0, 1]
 81: (4)                y = [.5, 1, -1, 1, -2, .5]
 82: (4)                simplified = _get_simplified(x, y)
 83: (4)                assert_array_almost_equal([[0., 0.5],
 84: (31)                                          [0., 1.],
 85: (31)                                          [0., -2.],
 86: (31)                                          [1., 0.5]],
 87: (30)                                         simplified.vertices[:-2, :])
 88: (4)                x = [0, 0, 0, 0, 0, 1]
 89: (4)                y = [.5, 1, -1, 1, 0, .5]
 90: (4)                simplified = _get_simplified(x, y)
 91: (4)                assert_array_almost_equal([[0., 0.5],
 92: (31)                                          [0., 1.],
 93: (31)                                          [0., -1.],
 94: (31)                                          [0., 0.],
 95: (31)                                          [1., 0.5]],
 96: (30)                                         simplified.vertices[:-2, :])
 97: (4)                x = [0, 0, 0, 0, 0, 1]
 98: (4)                y = [.5, 1, 2, 1, 3, .5]
 99: (4)                simplified = _get_simplified(x, y)
100: (4)                assert_array_almost_equal([[0., 0.5],
101: (31)                                          [0., 3.],
102: (31)                                          [1., 0.5]],
103: (30)                                         simplified.vertices[:-2, :])
104: (4)                x = [0, 0, 0, 0, 0, 1]
105: (4)                y = [.5, 1, 2, 1, 1, .5]
106: (4)                simplified = _get_simplified(x, y)
```

```
(4)                 assert_array_almost_equal([[0., 0.5],
(31)                                           [0., 2.],
(31)                                           [0., 1.],
(31)                                           [1., 0.5]],
(30)                                          simplified.vertices[:-2, :])
(0)             @pytest.mark.parametrize('angle', [0, np.pi/4, np.pi/3, np.pi/2])
(0)             @pytest.mark.parametrize('offset', [0, .5])
(0)             def test_angled_antiparallel(angle, offset):
(4)                 scale = 5
(4)                 np.random.seed(19680801)
(4)                 vert_offsets = (np.random.rand(15) - offset) * scale
(4)                 vert_offsets[0] = 0
(4)                 vert_offsets[1] = 1
(4)                 x = np.sin(angle) * vert_offsets
(4)                 y = np.cos(angle) * vert_offsets
(4)                 x_max = x[1:].max()
(4)                 x_min = x[1:].min()
(4)                 y_max = y[1:].max()
(4)                 y_min = y[1:].min()
(4)                 if offset > 0:
(8)                     p_expected = Path([[0, 0],
(27)                                       [x_max, y_max],
(27)                                       [x_min, y_min],
(27)                                       [x[-1], y[-1]],
(27)                                       [0, 0]],
(26)                                      codes=[1, 2, 2, 2, 0])
(4)                 else:
(8)                     p_expected = Path([[0, 0],
(27)                                       [x_max, y_max],
(27)                                       [x[-1], y[-1]],
(27)                                       [0, 0]],
(26)                                      codes=[1, 2, 2, 0])
(4)                 p = Path(np.vstack([x, y]).T)
(4)                 p2 = p.cleaned(simplify=True)
(4)                 assert_array_almost_equal(p_expected.vertices,
(30)                                          p2.vertices)
(4)                 assert_array_equal(p_expected.codes, p2.codes)
(0)             def test_sine_plus_noise():
(4)                 np.random.seed(0)
(4)                 x = (np.sin(np.linspace(0, np.pi * 2.0, 50000)) +
(9)                      np.random.uniform(size=50000) * 0.01)
(4)                 fig, ax = plt.subplots()
(4)                 p1 = ax.plot(x, solid_joinstyle='round', linewidth=2.0)
(4)                 fig.canvas.draw()
(4)                 path = p1[0].get_path()
(4)                 transform = p1[0].get_transform()
(4)                 path = transform.transform_path(path)
(4)                 simplified = path.cleaned(simplify=True)
(4)                 assert simplified.vertices.size == 25240
(0)             @image_comparison(['simplify_curve'], remove_text=True, tol=0.017)
(0)             def test_simplify_curve():
(4)                 pp1 = patches.PathPatch(
(8)                     Path([(0, 0), (1, 0), (1, 1), (np.nan, 1), (0, 0), (2, 0), (2, 2),
(14)                          (0, 0)],
(13)                         [Path.MOVETO, Path.CURVE3, Path.CURVE3, Path.CURVE3, Path.CURVE3,
(14)                          Path.CURVE3, Path.CURVE3, Path.CLOSEPOLY]),
(8)                     fc="none")
(4)                 fig, ax = plt.subplots()
(4)                 ax.add_patch(pp1)
(4)                 ax.set_xlim((0, 2))
(4)                 ax.set_ylim((0, 2))
(0)             @check_figures_equal()
(0)             def test_closed_path_nan_removal(fig_test, fig_ref):
(4)                 ax_test = fig_test.subplots(2, 2).flatten()
(4)                 ax_ref = fig_ref.subplots(2, 2).flatten()
(4)                 path = Path(
(8)                     [[-3, np.nan], [3, -3], [3, 3], [-3, 3], [-3, -3]],
(8)                     [Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO, Path.CLOSEPOLY])
(4)                 ax_test[0].add_patch(patches.PathPatch(path, facecolor='none'))
```

```
(4)                 path = Path(
(8)                     [[-3, np.nan], [3, -3], [3, 3], [-3, 3], [-3, np.nan]],
(8)                     [Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO, Path.LINETO])
(4)                 ax_ref[0].add_patch(patches.PathPatch(path, facecolor='none'))
(4)                 path = Path(
(8)                     [[-2, -2], [2, -2], [2, 2], [-2, np.nan], [-2, -2]],
(8)                     [Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO, Path.CLOSEPOLY])
(4)                 ax_test[0].add_patch(patches.PathPatch(path, facecolor='none'))
(4)                 path = Path(
(8)                     [[-2, -2], [2, -2], [2, 2], [-2, np.nan], [-2, -2]],
(8)                     [Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO, Path.LINETO])
(4)                 ax_ref[0].add_patch(patches.PathPatch(path, facecolor='none'))
(4)                 path = Path(
(8)                     [[-3, np.nan], [3, -3], [3, 3], [-3, 3], [-3, -3],
(9)                      [-2, -2], [2, -2], [2, 2], [-2, np.nan], [-2, -2]],
(8)                     [Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO, Path.CLOSEPOLY,
(9)                      Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO, Path.CLOSEPOLY])
(4)                 ax_test[1].add_patch(patches.PathPatch(path, facecolor='none'))
(4)                 path = Path(
(8)                     [[-3, np.nan], [3, -3], [3, 3], [-3, 3], [-3, np.nan],
(9)                      [-2, -2], [2, -2], [2, 2], [-2, np.nan], [-2, -2]],
(8)                     [Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO, Path.LINETO,
(9)                      Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO, Path.LINETO])
(4)                 ax_ref[1].add_patch(patches.PathPatch(path, facecolor='none'))
(4)                 path = Path(
(8)                     [[-1, -1], [1, -1], [1, np.nan], [0, 1], [-1, 1], [-1, -1]],
(8)                     [Path.MOVETO, Path.LINETO, Path.CURVE3, Path.CURVE3, Path.LINETO,
(9)                      Path.CLOSEPOLY])
(4)                 ax_test[2].add_patch(patches.PathPatch(path, facecolor='none'))
(4)                 path = Path(
(8)                     [[-1, -1], [1, -1], [1, np.nan], [0, 1], [-1, 1], [-1, -1]],
(8)                     [Path.MOVETO, Path.LINETO, Path.CURVE3, Path.CURVE3, Path.LINETO,
(9)                      Path.CLOSEPOLY])
(4)                 ax_ref[2].add_patch(patches.PathPatch(path, facecolor='none'))
(4)                 path = Path(
(8)                     [[-3, -3], [3, -3], [3, 0], [0, np.nan], [-3, 3], [-3, -3]],
(8)                     [Path.MOVETO, Path.LINETO, Path.CURVE3, Path.CURVE3, Path.LINETO,
(9)                      Path.LINETO])
(4)                 ax_test[2].add_patch(patches.PathPatch(path, facecolor='none'))
(4)                 path = Path(
(8)                     [[-3, -3], [3, -3], [3, 0], [0, np.nan], [-3, 3], [-3, -3]],
(8)                     [Path.MOVETO, Path.LINETO, Path.CURVE3, Path.CURVE3, Path.LINETO,
(9)                      Path.LINETO])
(4)                 ax_ref[2].add_patch(patches.PathPatch(path, facecolor='none'))
(4)                 path = Path(
(8)                     [[-1, -1], [1, -1], [1, np.nan], [0, 0], [0, 1], [-1, 1], [-1, -1]],
(8)                     [Path.MOVETO, Path.LINETO, Path.CURVE4, Path.CURVE4, Path.CURVE4,
(9)                      Path.LINETO, Path.CLOSEPOLY])
(4)                 ax_test[3].add_patch(patches.PathPatch(path, facecolor='none'))
(4)                 path = Path(
(8)                     [[-1, -1], [1, -1], [1, np.nan], [0, 0], [0, 1], [-1, 1], [-1, -1]],
(8)                     [Path.MOVETO, Path.LINETO, Path.CURVE4, Path.CURVE4, Path.CURVE4,
(9)                      Path.LINETO, Path.CLOSEPOLY])
(4)                 ax_ref[3].add_patch(patches.PathPatch(path, facecolor='none'))
(4)                 path = Path(
(8)                     [[-2, -2], [2, -2], [2, 0], [0, np.nan], [0, 2], [-2, 2], [-2, -2]],
(8)                     [Path.MOVETO, Path.LINETO, Path.CURVE4, Path.CURVE4, Path.CURVE4,
(9)                      Path.LINETO, Path.LINETO])
(4)                 ax_test[3].add_patch(patches.PathPatch(path, facecolor='none'))
(4)                 path = Path(
(8)                     [[-2, -2], [2, -2], [2, 0], [0, np.nan], [0, 2], [-2, 2], [-2, -2]],
(8)                     [Path.MOVETO, Path.LINETO, Path.CURVE4, Path.CURVE4, Path.CURVE4,
(9)                      Path.LINETO, Path.LINETO])
(4)                 ax_ref[3].add_patch(patches.PathPatch(path, facecolor='none'))
(4)                 path = Path(
(8)                     [[-3, -3], [3, -3], [3, 0], [0, 0], [0, np.nan], [-3, 3], [-3, -3]],
(8)                     [Path.MOVETO, Path.LINETO, Path.CURVE4, Path.CURVE4, Path.CURVE4,
(9)                      Path.LINETO, Path.LINETO])
(4)                 ax_test[3].add_patch(patches.PathPatch(path, facecolor='none'))
```

```
245: (4)                path = Path(
246: (8)                    [[-3, -3], [3, -3], [3, 0], [0, 0], [0, np.nan], [-3, 3], [-3, -3]],
247: (8)                    [Path.MOVETO, Path.LINETO, Path.CURVE4, Path.CURVE4, Path.CURVE4,
248: (9)                     Path.LINETO, Path.LINETO])
249: (4)                ax_ref[3].add_patch(patches.PathPatch(path, facecolor='none'))
250: (4)                for ax in [*ax_test.flat, *ax_ref.flat]:
251: (8)                    ax.set(xlim=(-3.5, 3.5), ylim=(-3.5, 3.5))
252: (4)                remove_ticks_and_titles(fig_test)
253: (4)                remove_ticks_and_titles(fig_ref)
254: (0)            @check_figures_equal()
255: (0)            def test_closed_path_clipping(fig_test, fig_ref):
256: (4)                vertices = []
257: (4)                for roll in range(8):
258: (8)                    offset = 0.1 * roll + 0.1
259: (8)                    pattern = [
260: (12)                       [-0.5, 1.5], [-0.5, -0.5], [1.5, -0.5], [1.5, 1.5],  # Outer
square
261: (12)                       [1 - offset / 2, 1.5], [1 - offset / 2, offset],
262: (12)                       [offset / 2, offset], [offset / 2, 1.5],
263: (8)                    ]
264: (8)                    pattern = np.roll(pattern, roll, axis=0)
265: (8)                    pattern = np.concatenate((pattern, pattern[:1, :]))
266: (8)                    vertices.append(pattern)
267: (4)                codes = np.full(len(vertices[0]), Path.LINETO)
268: (4)                codes[0] = Path.MOVETO
269: (4)                codes[-1] = Path.CLOSEPOLY
270: (4)                codes = np.tile(codes, len(vertices))
271: (4)                vertices = np.concatenate(vertices)
272: (4)                fig_test.set_size_inches((5, 5))
273: (4)                path = Path(vertices, codes)
274: (4)                fig_test.add_artist(patches.PathPatch(path, facecolor='none'))
275: (4)                fig_ref.set_size_inches((5, 5))
276: (4)                codes = codes.copy()
277: (4)                codes[codes == Path.CLOSEPOLY] = Path.LINETO
278: (4)                path = Path(vertices, codes)
279: (4)                fig_ref.add_artist(patches.PathPatch(path, facecolor='none'))
280: (0)            @image_comparison(['hatch_simplify'], remove_text=True)
281: (0)            def test_hatch():
282: (4)                fig, ax = plt.subplots()
283: (4)                ax.add_patch(plt.Rectangle((0, 0), 1, 1, fill=False, hatch="/"))
284: (4)                ax.set_xlim((0.45, 0.55))
285: (4)                ax.set_ylim((0.45, 0.55))
286: (0)            @image_comparison(['fft_peaks'], remove_text=True)
287: (0)            def test_fft_peaks():
288: (4)                fig, ax = plt.subplots()
289: (4)                t = np.arange(65536)
290: (4)                p1 = ax.plot(abs(np.fft.fft(np.sin(2*np.pi*.01*t)*np.blackman(len(t)))))
291: (4)                fig.canvas.draw()
292: (4)                path = p1[0].get_path()
293: (4)                transform = p1[0].get_transform()
294: (4)                path = transform.transform_path(path)
295: (4)                simplified = path.cleaned(simplify=True)
296: (4)                assert simplified.vertices.size == 36
297: (0)            def test_start_with_moveto():
298: (4)                data = b"""
299: (0)            ZwAAAAku+v9UAQAA+Tj6/z8CAADpQ/r/KAMAANlO+v8QBAAAyVn6//UEAAC6ZPr/2gUAAKpv+v+8
300: (0)            BgAAm3r6/50HAACLhfr/ewgAAHyQ+v9ZCQAAbZv6/zQKAABepvr/DgsAAE+x+v/lCwAAQLz6/7wM
301: (0)            AAAxx/r/kA0AACPS+v9jDgAAFN36/zQPAAAF6Pr/AxAAAPfy+v/QEAAA6f36/5wRAADbCPv/ZhIA
302: (0)            AMwT+/8uEwAAvh77//UTAACwKfv/uRQAAKM0+/98FQAAlT/7/z0WAACHSvv//RYAAHlV+/+7FwAA
303: (0)            bGD7/3cYAABea/v/MRkAAFF2+//pGQAARIH7/6AaAAA3jPv/VRsAACmX+/8JHAAAHKL7/7ocAAAP
304: (0)            rfv/ah0AAAO4+/8YHgAA9sL7/8QeAADpzfv/bx8AANzY+/8YIAAA0OP7/78gAADD7vv/ZCEAALf5
305: (0)            +/8IIgAAqwT8/6kiAACeD/z/SiMAAJIa/P/oIwAAhiX8/4QkAAB6MPz/HyUAAG47/P+4JQAAYkb8
306: (0)            /1AmAABWUfz/5SYAAEpc/P95JwAAPmf8/wsoAAAzcvz/nCgAACd9/P8qKQAAHIj8/7cpAAAQk/z/
307: (0)            QyoAAAWe/P/MKgAA+aj8/1QrAADus/z/2isAAOO+/P9eLAAA2Mn8/+AsAADM1Pz/YS0AAMHf/P/g
308: (0)            LQAAtur8/10uAACr9fz/2C4AAKEA/f9SLwAAlgv9/8ovAACLFv3/QDAAAIAh/f+1MAAAdSz9/ycx
309: (0)            AABrN/3/mDEAAGBC/f8IMgAAVk39/3UyAABLWP3/4TIAAEFj/f9LMwAANm79/7MzAAAsef3/GjQA
310: (0)            ACKE/f9+NAAAF4/9/+E0AAANmv3/QzUAAAOl/f+iNQAA+a/9/wA2AADvuv3/XDYAAOXF/f+2NgAA
311: (0)            29D9/w83AADR2/3/ZjcAAMfm/f+7NwAAvfH9/w44AACz/P3/XzgAAKkH/v+vOAAAnxL+//04AACW
312: (0)            Hf7/SjkAAIwo/v+UOQAAgjP+/905AAB5Pv7/JDoAAG9J/v9pOgAAZVT+/606AABcX/7/7zoAAFJq
```

```
313: (0)            /v8vOwAASXX+/207AAA/gP7/qjsAADaL/v/lOwAALZb+/x48AAAjof7/VTwAABqs/v+LPAAAELf+
314: (0)            /788AAAHwv7/8TwAAP7M/v8hPQAA9df+/1A9AADr4v7/fT0AAOLt/v+oPQAA2fj+/9E9AADQA///
315: (0)            +T0AAMYO//8fPgAAvRn//0M+AAC0JP//ZT4AAKsv//+GPgAAojr//6U+AACZRf//wj4AAJBQ///d
316: (0)            PgAAh1v///c+AAB+Zv//Dz8AAHRx//8lPwAAa3z//zk/AABih///TD8AAFmS//9dPwAAUJ3//2w/
317: (0)            AABHqP//ej8AAD6z//+FPwAANb7//48/AAAsyf//lz8AACPU//+ePwAAGt///6M/AAAR6v//pj8A
318: (0)            AAj1//+nPwAA/////w=="""
319: (4)                verts = np.frombuffer(base64.decodebytes(data), dtype='<i4')
320: (4)                verts = verts.reshape((len(verts) // 2, 2))
321: (4)                path = Path(verts)
322: (4)                segs = path.iter_segments(transforms.IdentityTransform(),
323: (30)                                          clip=(0.0, 0.0, 100.0, 100.0))
324: (4)                segs = list(segs)
325: (4)                assert len(segs) == 1
326: (4)                assert segs[0][1] == Path.MOVETO
327: (0)            def test_throw_rendering_complexity_exceeded():
328: (4)                plt.rcParams['path.simplify'] = False
329: (4)                xx = np.arange(2_000_000)
330: (4)                yy = np.random.rand(2_000_000)
331: (4)                yy[1000] = np.nan
332: (4)                fig, ax = plt.subplots()
333: (4)                ax.plot(xx, yy)
334: (4)                with pytest.raises(OverflowError):
335: (8)                    fig.savefig(io.BytesIO())
336: (0)            @image_comparison(['clipper_edge'], remove_text=True)
337: (0)            def test_clipper():
338: (4)                dat = (0, 1, 0, 2, 0, 3, 0, 4, 0, 5)
339: (4)                fig = plt.figure(figsize=(2, 1))
340: (4)                fig.subplots_adjust(left=0, bottom=0, wspace=0, hspace=0)
341: (4)                ax = fig.add_axes((0, 0, 1.0, 1.0), ylim=(0, 5), autoscale_on=False)
342: (4)                ax.plot(dat)
343: (4)                ax.xaxis.set_major_locator(plt.MultipleLocator(1))
344: (4)                ax.yaxis.set_major_locator(plt.MultipleLocator(1))
345: (4)                ax.xaxis.set_ticks_position('bottom')
346: (4)                ax.yaxis.set_ticks_position('left')
347: (4)                ax.set_xlim(5, 9)
348: (0)            @image_comparison(['para_equal_perp'], remove_text=True)
349: (0)            def test_para_equal_perp():
350: (4)                x = np.array([0, 1, 2, 1, 0, -1, 0, 1] + [1] * 128)
351: (4)                y = np.array([1, 1, 2, 1, 0, -1, 0, 0] + [0] * 128)
352: (4)                fig, ax = plt.subplots()
353: (4)                ax.plot(x + 1, y + 1)
354: (4)                ax.plot(x + 1, y + 1, 'ro')
355: (0)            @image_comparison(['clipping_with_nans'])
356: (0)            def test_clipping_with_nans():
357: (4)                x = np.linspace(0, 3.14 * 2, 3000)
358: (4)                y = np.sin(x)
359: (4)                x[::100] = np.nan
360: (4)                fig, ax = plt.subplots()
361: (4)                ax.plot(x, y)
362: (4)                ax.set_ylim(-0.25, 0.25)
363: (0)            def test_clipping_full():
364: (4)                p = Path([[1e30, 1e30]] * 5)
365: (4)                simplified = list(p.iter_segments(clip=[0, 0, 100, 100]))
366: (4)                assert simplified == []
367: (4)                p = Path([[50, 40], [75, 65]], [1, 2])
368: (4)                simplified = list(p.iter_segments(clip=[0, 0, 100, 100]))
369: (4)                assert ([(list(x), y) for x, y in simplified] ==
370: (12)                       [([50, 40], 1), ([75, 65], 2)])
371: (4)                p = Path([[50, 40]], [1])
372: (4)                simplified = list(p.iter_segments(clip=[0, 0, 100, 100]))
373: (4)                assert ([(list(x), y) for x, y in simplified] ==
374: (12)                       [([50, 40], 1)])

----------------------------------------

File 211 - test_skew.py:

1: (0)              """
2: (0)              Testing that skewed Axes properly work.
```

```
3: (0)              """
4: (0)              from contextlib import ExitStack
5: (0)              import itertools
6: (0)              import platform
7: (0)              import matplotlib.pyplot as plt
8: (0)              from matplotlib.testing.decorators import image_comparison
9: (0)              from matplotlib.axes import Axes
10: (0)             import matplotlib.transforms as transforms
11: (0)             import matplotlib.axis as maxis
12: (0)             import matplotlib.spines as mspines
13: (0)             import matplotlib.patches as mpatch
14: (0)             from matplotlib.projections import register_projection
15: (0)             class SkewXTick(maxis.XTick):
16: (4)                 def draw(self, renderer):
17: (8)                     with ExitStack() as stack:
18: (12)                        for artist in [self.gridline, self.tick1line, self.tick2line,
19: (27)                                       self.label1, self.label2]:
20: (16)                            stack.callback(artist.set_visible, artist.get_visible())
21: (12)                        needs_lower = transforms.interval_contains(
22: (16)                            self.axes.lower_xlim, self.get_loc())
23: (12)                        needs_upper = transforms.interval_contains(
24: (16)                            self.axes.upper_xlim, self.get_loc())
25: (12)                        self.tick1line.set_visible(
26: (16)                            self.tick1line.get_visible() and needs_lower)
27: (12)                        self.label1.set_visible(
28: (16)                            self.label1.get_visible() and needs_lower)
29: (12)                        self.tick2line.set_visible(
30: (16)                            self.tick2line.get_visible() and needs_upper)
31: (12)                        self.label2.set_visible(
32: (16)                            self.label2.get_visible() and needs_upper)
33: (12)                        super().draw(renderer)
34: (4)                 def get_view_interval(self):
35: (8)                     return self.axes.xaxis.get_view_interval()
36: (0)             class SkewXAxis(maxis.XAxis):
37: (4)                 def _get_tick(self, major):
38: (8)                     return SkewXTick(self.axes, None, major=major)
39: (4)                 def get_view_interval(self):
40: (8)                     return self.axes.upper_xlim[0], self.axes.lower_xlim[1]
41: (0)             class SkewSpine(mspines.Spine):
42: (4)                 def _adjust_location(self):
43: (8)                     pts = self._path.vertices
44: (8)                     if self.spine_type == 'top':
45: (12)                        pts[:, 0] = self.axes.upper_xlim
46: (8)                     else:
47: (12)                        pts[:, 0] = self.axes.lower_xlim
48: (0)             class SkewXAxes(Axes):
49: (4)                 name = 'skewx'
50: (4)                 def _init_axis(self):
51: (8)                     self.xaxis = SkewXAxis(self)
52: (8)                     self.spines.top.register_axis(self.xaxis)
53: (8)                     self.spines.bottom.register_axis(self.xaxis)
54: (8)                     self.yaxis = maxis.YAxis(self)
55: (8)                     self.spines.left.register_axis(self.yaxis)
56: (8)                     self.spines.right.register_axis(self.yaxis)
57: (4)                 def _gen_axes_spines(self):
58: (8)                     spines = {'top': SkewSpine.linear_spine(self, 'top'),
59: (18)                              'bottom': mspines.Spine.linear_spine(self, 'bottom'),
60: (18)                              'left': mspines.Spine.linear_spine(self, 'left'),
61: (18)                              'right': mspines.Spine.linear_spine(self, 'right')}
62: (8)                     return spines
63: (4)                 def _set_lim_and_transforms(self):
64: (8)                     """
65: (8)                     This is called once when the plot is created to set up all the
66: (8)                     transforms for the data, text and grids.
67: (8)                     """
68: (8)                     rot = 30
69: (8)                     super()._set_lim_and_transforms()
70: (8)                     self.transDataToAxes = (self.transScale +
71: (32)                                            (self.transLimits +
```

```
72: (33)                                        transforms.Affine2D().skew_deg(rot, 0)))
73: (8)                     self.transData = self.transDataToAxes + self.transAxes
74: (8)                     self._xaxis_transform = (transforms.blended_transform_factory(
75: (12)                        self.transScale + self.transLimits,
76: (12)                        transforms.IdentityTransform()) +
77: (12)                        transforms.Affine2D().skew_deg(rot, 0)) + self.transAxes
78: (4)                 @property
79: (4)                 def lower_xlim(self):
80: (8)                     return self.axes.viewLim.intervalx
81: (4)                 @property
82: (4)                 def upper_xlim(self):
83: (8)                     pts = [[0., 1.], [1., 1.]]
84: (8)                     return self.transDataToAxes.inverted().transform(pts)[:, 0]
85: (0)             register_projection(SkewXAxes)
86: (0)             @image_comparison(['skew_axes'], remove_text=True)
87: (0)             def test_set_line_coll_dash_image():
88: (4)                 fig = plt.figure()
89: (4)                 ax = fig.add_subplot(1, 1, 1, projection='skewx')
90: (4)                 ax.set_xlim(-50, 50)
91: (4)                 ax.set_ylim(50, -50)
92: (4)                 ax.grid(True)
93: (4)                 ax.axvline(0, color='b')
94: (0)             @image_comparison(['skew_rects'], remove_text=True,
95: (18)                            tol=0.009 if platform.machine() == 'arm64' else 0)
96: (0)             def test_skew_rectangle():
97: (4)                 fix, axes = plt.subplots(5, 5, sharex=True, sharey=True, figsize=(8, 8))
98: (4)                 axes = axes.flat
99: (4)                 rotations = list(itertools.product([-3, -1, 0, 1, 3], repeat=2))
100: (4)                axes[0].set_xlim([-3, 3])
101: (4)                axes[0].set_ylim([-3, 3])
102: (4)                axes[0].set_aspect('equal', share=True)
103: (4)                for ax, (xrots, yrots) in zip(axes, rotations):
104: (8)                    xdeg, ydeg = 45 * xrots, 45 * yrots
105: (8)                    t = transforms.Affine2D().skew_deg(xdeg, ydeg)
106: (8)                    ax.set_title(f'Skew of {xdeg} in X and {ydeg} in Y')
107: (8)                    ax.add_patch(mpatch.Rectangle([-1, -1], 2, 2,
108: (38)                                                 transform=t + ax.transData,
109: (38)                                                 alpha=0.5, facecolor='coral'))
110: (4)                plt.subplots_adjust(wspace=0, left=0.01, right=0.99, bottom=0.01,
top=0.99)
```

File 212 - test_sphinxext.py:

```
1: (0)              """Tests for tinypages build using sphinx extensions."""
2: (0)              import filecmp
3: (0)              import os
4: (0)              from pathlib import Path
5: (0)              import shutil
6: (0)              import sys
7: (0)              from matplotlib.testing import subprocess_run_for_testing
8: (0)              import pytest
9: (0)              pytest.importorskip('sphinx',
10: (20)                            minversion=None if sys.version_info < (3, 10) else
'4.1.3')
11: (0)             def build_sphinx_html(source_dir, doctree_dir, html_dir, extra_args=None):
12: (4)                 extra_args = [] if extra_args is None else extra_args
13: (4)                 cmd = [sys.executable, '-msphinx', '-W', '-b', 'html',
14: (11)                       '-d', str(doctree_dir), str(source_dir), str(html_dir),
*extra_args]
15: (4)                 proc = subprocess_run_for_testing(
16: (8)                     cmd, capture_output=True, text=True,
17: (8)                     env={**os.environ, "MPLBACKEND": ""})
18: (4)                 out = proc.stdout
19: (4)                 err = proc.stderr
20: (4)                 assert proc.returncode == 0, \
21: (8)                     f"sphinx build failed with stdout:\n{out}\nstderr:\n{err}\n"
22: (4)                 if err:
```

```
 23: (8)                    pytest.fail(f"sphinx build emitted the following warnings:\n{err}")
 24: (4)                assert html_dir.is_dir()
 25: (0)            def test_tinypages(tmp_path):
 26: (4)                shutil.copytree(Path(__file__).parent / 'tinypages', tmp_path,
 27: (20)                               dirs_exist_ok=True)
 28: (4)                html_dir = tmp_path / '_build' / 'html'
 29: (4)                img_dir = html_dir / '_images'
 30: (4)                doctree_dir = tmp_path / 'doctrees'
 31: (4)                cmd = [sys.executable, '-msphinx', '-W', '-b', 'html',
 32: (11)                      '-d', str(doctree_dir),
 33: (11)                      str(Path(__file__).parent / 'tinypages'), str(html_dir)]
 34: (4)                proc = subprocess_run_for_testing(
 35: (8)                    cmd, capture_output=True, text=True,
 36: (8)                    env={**os.environ, "MPLBACKEND": "", "GCOV_ERROR_FILE": os.devnull}
 37: (4)                )
 38: (4)                out = proc.stdout
 39: (4)                err = proc.stderr
 40: (4)                build_sphinx_html(tmp_path, doctree_dir, html_dir)
 41: (4)                def plot_file(num):
 42: (8)                    return img_dir / f'some_plots-{num}.png'
 43: (4)                def plot_directive_file(num):
 44: (8)                    return doctree_dir.parent / 'plot_directive' / f'some_plots-{num}.png'
 45: (4)                range_10, range_6, range_4 = [plot_file(i) for i in range(1, 4)]
 46: (4)                assert filecmp.cmp(range_6, plot_file(5))
 47: (4)                assert filecmp.cmp(range_4, plot_file(7))
 48: (4)                assert filecmp.cmp(range_10, plot_file(11))
 49: (4)                assert filecmp.cmp(range_10, plot_file('12_00'))
 50: (4)                assert filecmp.cmp(range_6, plot_file('12_01'))
 51: (4)                assert filecmp.cmp(range_4, plot_file(13))
 52: (4)                html_contents = (html_dir / 'some_plots.html').read_bytes()
 53: (4)                assert b'# Only a comment' in html_contents
 54: (4)                assert filecmp.cmp(range_4, img_dir / 'range4.png')
 55: (4)                assert filecmp.cmp(range_6, img_dir / 'range6_range6.png')
 56: (4)                assert b'This is the caption for plot 15.' in html_contents
 57: (4)                assert b'Plot 17 uses the caption option.' in html_contents
 58: (4)                assert b'This is the caption for plot 18.' in html_contents
 59: (4)                assert b'plot-directive my-class my-other-class' in html_contents
 60: (4)                assert html_contents.count(b'This caption applies to both plots.') == 2
 61: (4)                assert filecmp.cmp(range_6, plot_file(17))
 62: (4)                assert filecmp.cmp(range_10, img_dir / 'range6_range10.png')
 63: (4)                contents = (tmp_path / 'included_plot_21.rst').read_bytes()
 64: (4)                contents = contents.replace(b'plt.plot(range(6))', b'plt.plot(range(4))')
 65: (4)                (tmp_path / 'included_plot_21.rst').write_bytes(contents)
 66: (4)                modification_times = [plot_directive_file(i).stat().st_mtime
 67: (26)                                     for i in (1, 2, 3, 5)]
 68: (4)                build_sphinx_html(tmp_path, doctree_dir, html_dir)
 69: (4)                assert filecmp.cmp(range_4, plot_file(17))
 70: (4)                assert plot_directive_file(1).stat().st_mtime == modification_times[0]
 71: (4)                assert plot_directive_file(2).stat().st_mtime == modification_times[1]
 72: (4)                assert plot_directive_file(3).stat().st_mtime == modification_times[2]
 73: (4)                assert filecmp.cmp(range_10, plot_file(1))
 74: (4)                assert filecmp.cmp(range_6, plot_file(2))
 75: (4)                assert filecmp.cmp(range_4, plot_file(3))
 76: (4)                assert plot_directive_file(5).stat().st_mtime > modification_times[3]
 77: (4)                assert filecmp.cmp(range_6, plot_file(5))
 78: (0)            def test_plot_html_show_source_link(tmp_path):
 79: (4)                parent = Path(__file__).parent
 80: (4)                shutil.copyfile(parent / 'tinypages/conf.py', tmp_path / 'conf.py')
 81: (4)                shutil.copytree(parent / 'tinypages/_static', tmp_path / '_static')
 82: (4)                doctree_dir = tmp_path / 'doctrees'
 83: (4)                (tmp_path / 'index.rst').write_text("""
 84: (0)            .. plot::
 85: (4)                plt.plot(range(2))
 86: (0)            """)
 87: (4)                html_dir1 = tmp_path / '_build' / 'html1'
 88: (4)                build_sphinx_html(tmp_path, doctree_dir, html_dir1)
 89: (4)                assert len(list(html_dir1.glob("**/index-1.py"))) == 1
 90: (4)                html_dir2 = tmp_path / '_build' / 'html2'
 91: (4)                build_sphinx_html(tmp_path, doctree_dir, html_dir2,
```

```
 92: (22)                                extra_args=['-D', 'plot_html_show_source_link=0'])
 93: (4)                 assert len(list(html_dir2.glob("**/index-1.py"))) == 0
 94: (0)             @pytest.mark.parametrize('plot_html_show_source_link', [0, 1])
 95: (0)             def test_show_source_link_true(tmp_path, plot_html_show_source_link):
 96: (4)                 parent = Path(__file__).parent
 97: (4)                 shutil.copyfile(parent / 'tinypages/conf.py', tmp_path / 'conf.py')
 98: (4)                 shutil.copytree(parent / 'tinypages/_static', tmp_path / '_static')
 99: (4)                 doctree_dir = tmp_path / 'doctrees'
 100: (4)                (tmp_path / 'index.rst').write_text("""
 101: (0)            .. plot::
 102: (4)                :show-source-link: true
 103: (4)                plt.plot(range(2))
 104: (0)            """)
 105: (4)                html_dir = tmp_path / '_build' / 'html'
 106: (4)                build_sphinx_html(tmp_path, doctree_dir, html_dir, extra_args=[
 107: (8)                    '-D', f'plot_html_show_source_link={plot_html_show_source_link}'])
 108: (4)                assert len(list(html_dir.glob("**/index-1.py"))) == 1
 109: (0)            @pytest.mark.parametrize('plot_html_show_source_link', [0, 1])
 110: (0)            def test_show_source_link_false(tmp_path, plot_html_show_source_link):
 111: (4)                parent = Path(__file__).parent
 112: (4)                shutil.copyfile(parent / 'tinypages/conf.py', tmp_path / 'conf.py')
 113: (4)                shutil.copytree(parent / 'tinypages/_static', tmp_path / '_static')
 114: (4)                doctree_dir = tmp_path / 'doctrees'
 115: (4)                (tmp_path / 'index.rst').write_text("""
 116: (0)            .. plot::
 117: (4)                :show-source-link: false
 118: (4)                plt.plot(range(2))
 119: (0)            """)
 120: (4)                html_dir = tmp_path / '_build' / 'html'
 121: (4)                build_sphinx_html(tmp_path, doctree_dir, html_dir, extra_args=[
 122: (8)                    '-D', f'plot_html_show_source_link={plot_html_show_source_link}'])
 123: (4)                assert len(list(html_dir.glob("**/index-1.py"))) == 0
 124: (0)            def test_srcset_version(tmp_path):
 125: (4)                shutil.copytree(Path(__file__).parent / 'tinypages', tmp_path,
 126: (20)                               dirs_exist_ok=True)
 127: (4)                html_dir = tmp_path / '_build' / 'html'
 128: (4)                img_dir = html_dir / '_images'
 129: (4)                doctree_dir = tmp_path / 'doctrees'
 130: (4)                build_sphinx_html(tmp_path, doctree_dir, html_dir, extra_args=[
 131: (8)                    '-D', 'plot_srcset=2x'])
 132: (4)                def plot_file(num, suff=''):
 133: (8)                    return img_dir / f'some_plots-{num}{suff}.png'
 134: (4)                for ind in [1, 2, 3, 5, 7, 11, 13, 15, 17]:
 135: (8)                    assert plot_file(ind).exists()
 136: (8)                    assert plot_file(ind, suff='.2x').exists()
 137: (4)                assert (img_dir / 'nestedpage-index-1.png').exists()
 138: (4)                assert (img_dir / 'nestedpage-index-1.2x.png').exists()
 139: (4)                assert (img_dir / 'nestedpage-index-2.png').exists()
 140: (4)                assert (img_dir / 'nestedpage-index-2.2x.png').exists()
 141: (4)                assert (img_dir / 'nestedpage2-index-1.png').exists()
 142: (4)                assert (img_dir / 'nestedpage2-index-1.2x.png').exists()
 143: (4)                assert (img_dir / 'nestedpage2-index-2.png').exists()
 144: (4)                assert (img_dir / 'nestedpage2-index-2.2x.png').exists()
 145: (4)                assert ('srcset="_images/some_plots-1.png, _images/some_plots-1.2x.png
 2.00x"'
 146: (12)                       in (html_dir / 'some_plots.html').read_text(encoding='utf-8'))
 147: (4)                st = ('srcset="../_images/nestedpage-index-1.png, '
 148: (10)                     '../_images/nestedpage-index-1.2x.png 2.00x"')
 149: (4)                assert st in (html_dir / 'nestedpage/index.html').read_text(encoding='utf-
 8')
 150: (4)                st = ('srcset="../_images/nestedpage2-index-2.png, '
 151: (10)                     '../_images/nestedpage2-index-2.2x.png 2.00x"')
 152: (4)                assert st in (html_dir /
 'nestedpage2/index.html').read_text(encoding='utf-8')

 ----------------------------------------

 File 213 - test_spines.py:
```

```
import numpy as np
import pytest
import matplotlib.pyplot as plt
from matplotlib.spines import Spines
from matplotlib.testing.decorators import check_figures_equal, image_comparison
def test_spine_class():
    """Test Spines and SpinesProxy in isolation."""
    class SpineMock:
        def __init__(self):
            self.val = None
        def set(self, **kwargs):
            vars(self).update(kwargs)
        def set_val(self, val):
            self.val = val
    spines_dict = {
        'left': SpineMock(),
        'right': SpineMock(),
        'top': SpineMock(),
        'bottom': SpineMock(),
    }
    spines = Spines(**spines_dict)
    assert spines['left'] is spines_dict['left']
    assert spines.left is spines_dict['left']
    spines[['left', 'right']].set_val('x')
    assert spines.left.val == 'x'
    assert spines.right.val == 'x'
    assert spines.top.val is None
    assert spines.bottom.val is None
    spines[:].set_val('y')
    assert all(spine.val == 'y' for spine in spines.values())
    spines[:].set(foo='bar')
    assert all(spine.foo == 'bar' for spine in spines.values())
    with pytest.raises(AttributeError, match='foo'):
        spines.foo
    with pytest.raises(KeyError, match='foo'):
        spines['foo']
    with pytest.raises(KeyError, match='foo, bar'):
        spines[['left', 'foo', 'right', 'bar']]
    with pytest.raises(ValueError, match='single list'):
        spines['left', 'right']
    with pytest.raises(ValueError, match='Spines does not support slicing'):
        spines['left':'right']
    with pytest.raises(ValueError, match='Spines does not support slicing'):
        spines['top':]
@image_comparison(['spines_axes_positions'])
def test_spines_axes_positions():
    fig = plt.figure()
    x = np.linspace(0, 2*np.pi, 100)
    y = 2*np.sin(x)
    ax = fig.add_subplot(1, 1, 1)
    ax.set_title('centered spines')
    ax.plot(x, y)
    ax.spines.right.set_position(('axes', 0.1))
    ax.yaxis.set_ticks_position('right')
    ax.spines.top.set_position(('axes', 0.25))
    ax.xaxis.set_ticks_position('top')
    ax.spines.left.set_color('none')
    ax.spines.bottom.set_color('none')
@image_comparison(['spines_data_positions'])
def test_spines_data_positions():
    fig, ax = plt.subplots()
    ax.spines.left.set_position(('data', -1.5))
    ax.spines.top.set_position(('data', 0.5))
    ax.spines.right.set_position(('data', -0.5))
    ax.spines.bottom.set_position('zero')
    ax.set_xlim([-2, 2])
    ax.set_ylim([-2, 2])
@check_figures_equal(extensions=["png"])
```

```
69: (0)             def test_spine_nonlinear_data_positions(fig_test, fig_ref):
70: (4)                 plt.style.use("default")
71: (4)                 ax = fig_test.add_subplot()
72: (4)                 ax.set(xscale="log", xlim=(.1, 1))
73: (4)                 ax.spines.left.set_position(("data", 1))
74: (4)                 ax.spines.left.set_linewidth(2)
75: (4)                 ax.spines.right.set_position(("data", .1))
76: (4)                 ax.tick_params(axis="y", labelleft=False, direction="in")
77: (4)                 ax = fig_ref.add_subplot()
78: (4)                 ax.set(xscale="log", xlim=(.1, 1))
79: (4)                 ax.spines.right.set_linewidth(2)
80: (4)                 ax.tick_params(axis="y", labelleft=False, left=False, right=True)
81: (0)             @image_comparison(['spines_capstyle'])
82: (0)             def test_spines_capstyle():
83: (4)                 plt.rc('axes', linewidth=20)
84: (4)                 fig, ax = plt.subplots()
85: (4)                 ax.set_xticks([])
86: (4)                 ax.set_yticks([])
87: (0)             def test_label_without_ticks():
88: (4)                 fig, ax = plt.subplots()
89: (4)                 plt.subplots_adjust(left=0.3, bottom=0.3)
90: (4)                 ax.plot(np.arange(10))
91: (4)                 ax.yaxis.set_ticks_position('left')
92: (4)                 ax.spines.left.set_position(('outward', 30))
93: (4)                 ax.spines.right.set_visible(False)
94: (4)                 ax.set_ylabel('y label')
95: (4)                 ax.xaxis.set_ticks_position('bottom')
96: (4)                 ax.spines.bottom.set_position(('outward', 30))
97: (4)                 ax.spines.top.set_visible(False)
98: (4)                 ax.set_xlabel('x label')
99: (4)                 ax.xaxis.set_ticks([])
100: (4)                ax.yaxis.set_ticks([])
101: (4)                plt.draw()
102: (4)                spine = ax.spines.left
103: (4)                spinebbox = spine.get_transform().transform_path(
104: (8)                    spine.get_path()).get_extents()
105: (4)                assert ax.yaxis.label.get_position()[0] < spinebbox.xmin, \
106: (8)                    "Y-Axis label not left of the spine"
107: (4)                spine = ax.spines.bottom
108: (4)                spinebbox = spine.get_transform().transform_path(
109: (8)                    spine.get_path()).get_extents()
110: (4)                assert ax.xaxis.label.get_position()[1] < spinebbox.ymin, \
111: (8)                    "X-Axis label not below the spine"
112: (0)            @image_comparison(['black_axes'])
113: (0)            def test_spines_black_axes():
114: (4)                plt.rcParams["savefig.pad_inches"] = 0
115: (4)                plt.rcParams["savefig.bbox"] = 'tight'
116: (4)                fig = plt.figure(0, figsize=(4, 4))
117: (4)                ax = fig.add_axes((0, 0, 1, 1))
118: (4)                ax.set_xticklabels([])
119: (4)                ax.set_yticklabels([])
120: (4)                ax.set_xticks([])
121: (4)                ax.set_yticks([])
122: (4)                ax.set_facecolor((0, 0, 0))
```

File 214 - test_streamplot.py:

```
1: (0)              import numpy as np
2: (0)              from numpy.testing import assert_array_almost_equal
3: (0)              import pytest
4: (0)              import matplotlib.pyplot as plt
5: (0)              from matplotlib.testing.decorators import image_comparison
6: (0)              import matplotlib.transforms as mtransforms
7: (0)              def velocity_field():
8: (4)                  Y, X = np.mgrid[-3:3:100j, -3:3:200j]
9: (4)                  U = -1 - X**2 + Y
10: (4)                 V = 1 + X - Y**2
```

```
 11: (4)                 return X, Y, U, V
 12: (0)             def swirl_velocity_field():
 13: (4)                 x = np.linspace(-3., 3., 200)
 14: (4)                 y = np.linspace(-3., 3., 100)
 15: (4)                 X, Y = np.meshgrid(x, y)
 16: (4)                 a = 0.1
 17: (4)                 U = np.cos(a) * (-Y) - np.sin(a) * X
 18: (4)                 V = np.sin(a) * (-Y) + np.cos(a) * X
 19: (4)                 return x, y, U, V
 20: (0)             @image_comparison(['streamplot_startpoints'], remove_text=True, style='mpl20',
 21: (18)                              extensions=['png'])
 22: (0)             def test_startpoints():
 23: (4)                 X, Y, U, V = velocity_field()
 24: (4)                 start_x, start_y = np.meshgrid(np.linspace(X.min(), X.max(), 5),
 25: (35)                                               np.linspace(Y.min(), Y.max(), 5))
 26: (4)                 start_points = np.column_stack([start_x.ravel(), start_y.ravel()])
 27: (4)                 plt.streamplot(X, Y, U, V, start_points=start_points)
 28: (4)                 plt.plot(start_x, start_y, 'ok')
 29: (0)             @image_comparison(['streamplot_colormap'], remove_text=True, style='mpl20',
 30: (18)                              tol=0.022)
 31: (0)             def test_colormap():
 32: (4)                 X, Y, U, V = velocity_field()
 33: (4)                 plt.streamplot(X, Y, U, V, color=U, density=0.6, linewidth=2,
 34: (19)                               cmap=plt.cm.autumn)
 35: (4)                 plt.colorbar()
 36: (0)             @image_comparison(['streamplot_linewidth'], remove_text=True, style='mpl20',
 37: (18)                              tol=0.004)
 38: (0)             def test_linewidth():
 39: (4)                 X, Y, U, V = velocity_field()
 40: (4)                 speed = np.hypot(U, V)
 41: (4)                 lw = 5 * speed / speed.max()
 42: (4)                 ax = plt.figure().subplots()
 43: (4)                 ax.streamplot(X, Y, U, V, density=[0.5, 1], color='k', linewidth=lw)
 44: (0)             @image_comparison(['streamplot_masks_and_nans'],
 45: (18)                              remove_text=True, style='mpl20')
 46: (0)             def test_masks_and_nans():
 47: (4)                 X, Y, U, V = velocity_field()
 48: (4)                 mask = np.zeros(U.shape, dtype=bool)
 49: (4)                 mask[40:60, 80:120] = 1
 50: (4)                 U[:20, :40] = np.nan
 51: (4)                 U = np.ma.array(U, mask=mask)
 52: (4)                 ax = plt.figure().subplots()
 53: (4)                 with np.errstate(invalid='ignore'):
 54: (8)                     ax.streamplot(X, Y, U, V, color=U, cmap=plt.cm.Blues)
 55: (0)             @image_comparison(['streamplot_maxlength.png'],
 56: (18)                              remove_text=True, style='mpl20', tol=0.302)
 57: (0)             def test_maxlength():
 58: (4)                 x, y, U, V = swirl_velocity_field()
 59: (4)                 ax = plt.figure().subplots()
 60: (4)                 ax.streamplot(x, y, U, V, maxlength=10., start_points=[[0., 1.5]],
 61: (18)                              linewidth=2, density=2)
 62: (4)                 assert ax.get_xlim()[-1] == ax.get_ylim()[-1] == 3
 63: (4)                 ax.set(xlim=(None, 3.2555988021882305), ylim=(None, 3.078326760195413))
 64: (0)             @image_comparison(['streamplot_maxlength_no_broken.png'],
 65: (18)                              remove_text=True, style='mpl20', tol=0.302)
 66: (0)             def test_maxlength_no_broken():
 67: (4)                 x, y, U, V = swirl_velocity_field()
 68: (4)                 ax = plt.figure().subplots()
 69: (4)                 ax.streamplot(x, y, U, V, maxlength=10., start_points=[[0., 1.5]],
 70: (18)                              linewidth=2, density=2, broken_streamlines=False)
 71: (4)                 assert ax.get_xlim()[-1] == ax.get_ylim()[-1] == 3
 72: (4)                 ax.set(xlim=(None, 3.2555988021882305), ylim=(None, 3.078326760195413))
 73: (0)             @image_comparison(['streamplot_direction.png'],
 74: (18)                              remove_text=True, style='mpl20', tol=0.073)
 75: (0)             def test_direction():
 76: (4)                 x, y, U, V = swirl_velocity_field()
 77: (4)                 plt.streamplot(x, y, U, V, integration_direction='backward',
 78: (19)                               maxlength=1.5, start_points=[[1.5, 0.]],
 79: (19)                               linewidth=2, density=2)
```

```
80: (0)             def test_streamplot_limits():
81: (4)                 ax = plt.axes()
82: (4)                 x = np.linspace(-5, 10, 20)
83: (4)                 y = np.linspace(-2, 4, 10)
84: (4)                 y, x = np.meshgrid(y, x)
85: (4)                 trans = mtransforms.Affine2D().translate(25, 32) + ax.transData
86: (4)                 plt.barbs(x, y, np.sin(x), np.cos(y), transform=trans)
87: (4)                 assert_array_almost_equal(ax.dataLim.bounds, (20, 30, 15, 6),
88: (30)                                          decimal=1)
89: (0)             def test_streamplot_grid():
90: (4)                 u = np.ones((2, 2))
91: (4)                 v = np.zeros((2, 2))
92: (4)                 x = np.array([[10, 20], [10, 30]])
93: (4)                 y = np.array([[10, 10], [20, 20]])
94: (4)                 with pytest.raises(ValueError, match="The rows of 'x' must be equal"):
95: (8)                     plt.streamplot(x, y, u, v)
96: (4)                 x = np.array([[10, 20], [10, 20]])
97: (4)                 y = np.array([[10, 10], [20, 30]])
98: (4)                 with pytest.raises(ValueError, match="The columns of 'y' must be equal"):
99: (8)                     plt.streamplot(x, y, u, v)
100: (4)                x = np.array([[10, 20], [10, 20]])
101: (4)                y = np.array([[10, 10], [20, 20]])
102: (4)                plt.streamplot(x, y, u, v)
103: (4)                x = np.array([0, 10])
104: (4)                y = np.array([[[0, 10]]])
105: (4)                with pytest.raises(ValueError, match="'y' can have at maximum "
106: (41)                                                "2 dimensions"):
107: (8)                    plt.streamplot(x, y, u, v)
108: (4)                u = np.ones((3, 3))
109: (4)                v = np.zeros((3, 3))
110: (4)                x = np.array([0, 10, 20])
111: (4)                y = np.array([0, 10, 30])
112: (4)                with pytest.raises(ValueError, match="'y' values must be equally spaced"):
113: (8)                    plt.streamplot(x, y, u, v)
114: (4)                x = np.array([0, 20, 40])
115: (4)                y = np.array([0, 20, 10])
116: (4)                with pytest.raises(ValueError, match="'y' must be strictly increasing"):
117: (8)                    plt.streamplot(x, y, u, v)
118: (0)            def test_streamplot_inputs():  # test no exception occurs.
119: (4)                plt.streamplot(np.arange(3), np.arange(3),
120: (19)                              np.full((3, 3), np.nan), np.full((3, 3), np.nan),
121: (19)                              color=np.random.rand(3, 3))
122: (4)                plt.streamplot(range(3), range(3),
123: (19)                              np.random.rand(3, 3), np.random.rand(3, 3))
```

File 215 - test_style.py:

```
1: (0)              from contextlib import contextmanager
2: (0)              from pathlib import Path
3: (0)              from tempfile import TemporaryDirectory
4: (0)              import sys
5: (0)              import numpy as np
6: (0)              import pytest
7: (0)              import matplotlib as mpl
8: (0)              from matplotlib import pyplot as plt, style
9: (0)              from matplotlib.style.core import USER_LIBRARY_PATHS, STYLE_EXTENSION
10: (0)             PARAM = 'image.cmap'
11: (0)             VALUE = 'pink'
12: (0)             DUMMY_SETTINGS = {PARAM: VALUE}
13: (0)             @contextmanager
14: (0)             def temp_style(style_name, settings=None):
15: (4)                 """Context manager to create a style sheet in a temporary directory."""
16: (4)                 if not settings:
17: (8)                     settings = DUMMY_SETTINGS
18: (4)                 temp_file = f'{style_name}.{STYLE_EXTENSION}'
19: (4)                 try:
20: (8)                     with TemporaryDirectory() as tmpdir:
```

```
(12)                        Path(tmpdir, temp_file).write_text(
(16)                            "\n".join(f"{k}: {v}" for k, v in settings.items()),
(16)                            encoding="utf-8")
(12)                        USER_LIBRARY_PATHS.append(tmpdir)
(12)                        style.reload_library()
(12)                        yield
(4)                 finally:
(8)                     style.reload_library()
(0)             def test_invalid_rc_warning_includes_filename(caplog):
(4)                 SETTINGS = {'foo': 'bar'}
(4)                 basename = 'basename'
(4)                 with temp_style(basename, SETTINGS):
(8)                     pass
(4)                 assert (len(caplog.records) == 1
(12)                        and basename in caplog.records[0].getMessage())
(0)             def test_available():
(4)                 with temp_style('_test_', DUMMY_SETTINGS):
(8)                     assert '_test_' in style.available
(0)             def test_use():
(4)                 mpl.rcParams[PARAM] = 'gray'
(4)                 with temp_style('test', DUMMY_SETTINGS):
(8)                     with style.context('test'):
(12)                        assert mpl.rcParams[PARAM] == VALUE
(0)             def test_use_url(tmp_path):
(4)                 path = tmp_path / 'file'
(4)                 path.write_text('axes.facecolor: adeade', encoding='utf-8')
(4)                 with temp_style('test', DUMMY_SETTINGS):
(8)                     url = ('file:'
(15)                           + ('///' if sys.platform == 'win32' else '')
(15)                           + path.resolve().as_posix())
(8)                     with style.context(url):
(12)                        assert mpl.rcParams['axes.facecolor'] == "#adeade"
(0)             def test_single_path(tmp_path):
(4)                 mpl.rcParams[PARAM] = 'gray'
(4)                 path = tmp_path / f'text.{STYLE_EXTENSION}'
(4)                 path.write_text(f'{PARAM} : {VALUE}', encoding='utf-8')
(4)                 with style.context(path):
(8)                     assert mpl.rcParams[PARAM] == VALUE
(4)                 assert mpl.rcParams[PARAM] == 'gray'
(0)             def test_context():
(4)                 mpl.rcParams[PARAM] = 'gray'
(4)                 with temp_style('test', DUMMY_SETTINGS):
(8)                     with style.context('test'):
(12)                        assert mpl.rcParams[PARAM] == VALUE
(4)                 assert mpl.rcParams[PARAM] == 'gray'
(0)             def test_context_with_dict():
(4)                 original_value = 'gray'
(4)                 other_value = 'blue'
(4)                 mpl.rcParams[PARAM] = original_value
(4)                 with style.context({PARAM: other_value}):
(8)                     assert mpl.rcParams[PARAM] == other_value
(4)                 assert mpl.rcParams[PARAM] == original_value
(0)             def test_context_with_dict_after_namedstyle():
(4)                 original_value = 'gray'
(4)                 other_value = 'blue'
(4)                 mpl.rcParams[PARAM] = original_value
(4)                 with temp_style('test', DUMMY_SETTINGS):
(8)                     with style.context(['test', {PARAM: other_value}]):
(12)                        assert mpl.rcParams[PARAM] == other_value
(4)                 assert mpl.rcParams[PARAM] == original_value
(0)             def test_context_with_dict_before_namedstyle():
(4)                 original_value = 'gray'
(4)                 other_value = 'blue'
(4)                 mpl.rcParams[PARAM] = original_value
(4)                 with temp_style('test', DUMMY_SETTINGS):
(8)                     with style.context([{PARAM: other_value}, 'test']):
(12)                        assert mpl.rcParams[PARAM] == VALUE
(4)                 assert mpl.rcParams[PARAM] == original_value
(0)             def test_context_with_union_of_dict_and_namedstyle():
```

```
 90: (4)                 original_value = 'gray'
 91: (4)                 other_param = 'text.usetex'
 92: (4)                 other_value = True
 93: (4)                 d = {other_param: other_value}
 94: (4)                 mpl.rcParams[PARAM] = original_value
 95: (4)                 mpl.rcParams[other_param] = (not other_value)
 96: (4)                 with temp_style('test', DUMMY_SETTINGS):
 97: (8)                     with style.context(['test', d]):
 98: (12)                        assert mpl.rcParams[PARAM] == VALUE
 99: (12)                        assert mpl.rcParams[other_param] == other_value
 100: (4)                assert mpl.rcParams[PARAM] == original_value
 101: (4)                assert mpl.rcParams[other_param] == (not other_value)
 102: (0)            def test_context_with_badparam():
 103: (4)                original_value = 'gray'
 104: (4)                other_value = 'blue'
 105: (4)                with style.context({PARAM: other_value}):
 106: (8)                    assert mpl.rcParams[PARAM] == other_value
 107: (8)                    x = style.context({PARAM: original_value, 'badparam': None})
 108: (8)                    with pytest.raises(KeyError):
 109: (12)                       with x:
 110: (16)                           pass
 111: (8)                    assert mpl.rcParams[PARAM] == other_value
 112: (0)            @pytest.mark.parametrize('equiv_styles',
 113: (25)                                   [('mpl20', 'default'),
 114: (26)                                    ('mpl15', 'classic')],
 115: (25)                                   ids=['mpl20', 'mpl15'])
 116: (0)            def test_alias(equiv_styles):
 117: (4)                rc_dicts = []
 118: (4)                for sty in equiv_styles:
 119: (8)                    with style.context(sty):
 120: (12)                       rc_dicts.append(mpl.rcParams.copy())
 121: (4)                rc_base = rc_dicts[0]
 122: (4)                for nm, rc in zip(equiv_styles[1:], rc_dicts[1:]):
 123: (8)                    assert rc_base == rc
 124: (0)            def test_xkcd_no_cm():
 125: (4)                assert mpl.rcParams["path.sketch"] is None
 126: (4)                plt.xkcd()
 127: (4)                assert mpl.rcParams["path.sketch"] == (1, 100, 2)
 128: (4)                np.testing.break_cycles()
 129: (4)                assert mpl.rcParams["path.sketch"] == (1, 100, 2)
 130: (0)            def test_xkcd_cm():
 131: (4)                assert mpl.rcParams["path.sketch"] is None
 132: (4)                with plt.xkcd():
 133: (8)                    assert mpl.rcParams["path.sketch"] == (1, 100, 2)
 134: (4)                assert mpl.rcParams["path.sketch"] is None
 135: (0)            def test_up_to_date_blacklist():
 136: (4)                assert mpl.style.core.STYLE_BLACKLIST <= {*mpl.rcsetup._validators}
 137: (0)            def test_style_from_module(tmp_path, monkeypatch):
 138: (4)                monkeypatch.syspath_prepend(tmp_path)
 139: (4)                monkeypatch.chdir(tmp_path)
 140: (4)                pkg_path = tmp_path / "mpl_test_style_pkg"
 141: (4)                pkg_path.mkdir()
 142: (4)                (pkg_path / "test_style.mplstyle").write_text(
 143: (8)                    "lines.linewidth: 42", encoding="utf-8")
 144: (4)                pkg_path.with_suffix(".mplstyle").write_text(
 145: (8)                    "lines.linewidth: 84", encoding="utf-8")
 146: (4)                mpl.style.use("mpl_test_style_pkg.test_style")
 147: (4)                assert mpl.rcParams["lines.linewidth"] == 42
 148: (4)                mpl.style.use("mpl_test_style_pkg.mplstyle")
 149: (4)                assert mpl.rcParams["lines.linewidth"] == 84
 150: (4)                mpl.style.use("./mpl_test_style_pkg.mplstyle")
 151: (4)                assert mpl.rcParams["lines.linewidth"] == 84

----------------------------------------

File 216 - test_subplots.py:

 1: (0)              import itertools
 2: (0)              import platform
```

```
(0)  import numpy as np
(0)  import pytest
(0)  from matplotlib.axes import Axes, SubplotBase
(0)  import matplotlib.pyplot as plt
(0)  from matplotlib.testing.decorators import check_figures_equal,
image_comparison
(0)  def check_shared(axs, x_shared, y_shared):
(4)      """
(4)      x_shared and y_shared are n x n boolean matrices; entry (i, j) indicates
(4)      whether the x (or y) axes of subplots i and j should be shared.
(4)      """
(4)      for (i1, ax1), (i2, ax2), (i3, (name, shared)) in itertools.product(
(12)             enumerate(axs),
(12)             enumerate(axs),
(12)             enumerate(zip("xy", [x_shared, y_shared]))):
(8)          if i2 <= i1:
(12)             continue
(8)          assert axs[0]._shared_axes[name].joined(ax1, ax2) == shared[i1, i2], \
(12)             "axes %i and %i incorrectly %ssharing %s axis" % (
(16)                 i1, i2, "not " if shared[i1, i2] else "", name)
(0)  def check_ticklabel_visible(axs, x_visible, y_visible):
(4)      """Check that the x and y ticklabel visibility is as specified."""
(4)      for i, (ax, vx, vy) in enumerate(zip(axs, x_visible, y_visible)):
(8)          for l in ax.get_xticklabels() + [ax.xaxis.offsetText]:
(12)             assert l.get_visible() == vx, \
(20)                     f"Visibility of x axis #{i} is incorrectly {vx}"
(8)          for l in ax.get_yticklabels() + [ax.yaxis.offsetText]:
(12)             assert l.get_visible() == vy, \
(20)                     f"Visibility of y axis #{i} is incorrectly {vy}"
(8)          if not vx:
(12)             assert ax.get_xlabel() == ""
(8)          if not vy:
(12)             assert ax.get_ylabel() == ""
(0)  def check_tick1_visible(axs, x_visible, y_visible):
(4)      """
(4)      Check that the x and y tick visibility is as specified.
(4)      Note: This only checks the tick1line, i.e. bottom / left ticks.
(4)      """
(4)      for ax, visible, in zip(axs, x_visible):
(8)          for tick in ax.xaxis.get_major_ticks():
(12)             assert tick.tick1line.get_visible() == visible
(4)      for ax, y_visible, in zip(axs, y_visible):
(8)          for tick in ax.yaxis.get_major_ticks():
(12)             assert tick.tick1line.get_visible() == visible
(0)  def test_shared():
(4)      rdim = (4, 4, 2)
(4)      share = {
(12)             'all': np.ones(rdim[:2], dtype=bool),
(12)             'none': np.zeros(rdim[:2], dtype=bool),
(12)             'row': np.array([
(16)                 [False, True, False, False],
(16)                 [True, False, False, False],
(16)                 [False, False, False, True],
(16)                 [False, False, True, False]]),
(12)             'col': np.array([
(16)                 [False, False, True, False],
(16)                 [False, False, False, True],
(16)                 [True, False, False, False],
(16)                 [False, True, False, False]]),
(12)             }
(4)      visible = {
(12)             'x': {
(16)                 'all': [False, False, True, True],
(16)                 'col': [False, False, True, True],
(16)                 'row': [True] * 4,
(16)                 'none': [True] * 4,
(16)                 False: [True] * 4,
(16)                 True: [False, False, True, True],
(16)                 },
```

```
(12)                        'y': {
(16)                            'all': [True, False, True, False],
(16)                            'col': [True] * 4,
(16)                            'row': [True, False, True, False],
(16)                            'none': [True] * 4,
(16)                            False: [True] * 4,
(16)                            True: [True, False, True, False],
(16)                            },
(12)                        }
(4)                 share[False] = share['none']
(4)                 share[True] = share['all']
(4)                 f, ((a1, a2), (a3, a4)) = plt.subplots(2, 2)
(4)                 axs = [a1, a2, a3, a4]
(4)                 check_shared(axs, share['none'], share['none'])
(4)                 plt.close(f)
(4)                 ops = [False, True, 'all', 'none', 'row', 'col', 0, 1]
(4)                 for xo in ops:
(8)                     for yo in ops:
(12)                        f, ((a1, a2), (a3, a4)) = plt.subplots(2, 2, sharex=xo, sharey=yo)
(12)                        axs = [a1, a2, a3, a4]
(12)                        check_shared(axs, share[xo], share[yo])
(12)                        check_ticklabel_visible(axs, visible['x'][xo], visible['y'][yo])
(12)                        plt.close(f)
(0)             @pytest.mark.parametrize('remove_ticks', [True, False])
(0)             def test_label_outer(remove_ticks):
(4)                 f, axs = plt.subplots(2, 2, sharex=True, sharey=True)
(4)                 for ax in axs.flat:
(8)                     ax.set(xlabel="foo", ylabel="bar")
(8)                     ax.label_outer(remove_inner_ticks=remove_ticks)
(4)                 check_ticklabel_visible(
(8)                     axs.flat, [False, False, True, True], [True, False, True, False])
(4)                 if remove_ticks:
(8)                     check_tick1_visible(
(12)                        axs.flat, [False, False, True, True], [True, False, True, False])
(4)                 else:
(8)                     check_tick1_visible(
(12)                        axs.flat, [True, True, True, True], [True, True, True, True])
(0)             def test_label_outer_span():
(4)                 fig = plt.figure()
(4)                 gs = fig.add_gridspec(3, 3)
(4)                 a1 = fig.add_subplot(gs[0, 0:2])
(4)                 a2 = fig.add_subplot(gs[1:3, 0])
(4)                 a3 = fig.add_subplot(gs[1, 2])
(4)                 a4 = fig.add_subplot(gs[2, 1])
(4)                 for ax in fig.axes:
(8)                     ax.label_outer()
(4)                 check_ticklabel_visible(
(8)                     fig.axes, [False, True, False, True], [True, True, False, False])
(0)             def test_label_outer_non_gridspec():
(4)                 ax = plt.axes((0, 0, 1, 1))
(4)                 ax.label_outer()  # Does nothing.
(4)                 check_ticklabel_visible([ax], [True], [True])
(0)             def test_shared_and_moved():
(4)                 f, (a1, a2) = plt.subplots(1, 2, sharey=True)
(4)                 check_ticklabel_visible([a2], [True], [False])
(4)                 a2.yaxis.tick_left()
(4)                 check_ticklabel_visible([a2], [True], [False])
(4)                 f, (a1, a2) = plt.subplots(2, 1, sharex=True)
(4)                 check_ticklabel_visible([a1], [False], [True])
(4)                 a2.xaxis.tick_bottom()
(4)                 check_ticklabel_visible([a1], [False], [True])
(0)             def test_exceptions():
(4)                 with pytest.raises(ValueError):
(8)                     plt.subplots(2, 2, sharex='blah')
(4)                 with pytest.raises(ValueError):
(8)                     plt.subplots(2, 2, sharey='blah')
(0)             @image_comparison(['subplots_offset_text'],
(18)                              tol=0.028 if platform.machine() == 'arm64' else 0)
(0)             def test_subplots_offsettext():
```

```
140: (4)                x = np.arange(0, 1e10, 1e9)
141: (4)                y = np.arange(0, 100, 10)+1e4
142: (4)                fig, axs = plt.subplots(2, 2, sharex='col', sharey='all')
143: (4)                axs[0, 0].plot(x, x)
144: (4)                axs[1, 0].plot(x, x)
145: (4)                axs[0, 1].plot(y, x)
146: (4)                axs[1, 1].plot(y, x)
147: (0)            @pytest.mark.parametrize("top", [True, False])
148: (0)            @pytest.mark.parametrize("bottom", [True, False])
149: (0)            @pytest.mark.parametrize("left", [True, False])
150: (0)            @pytest.mark.parametrize("right", [True, False])
151: (0)            def test_subplots_hide_ticklabels(top, bottom, left, right):
152: (4)                with plt.rc_context({"xtick.labeltop": top, "xtick.labelbottom": bottom,
153: (25)                                     "ytick.labelleft": left, "ytick.labelright": right}):
154: (8)                    axs = plt.figure().subplots(3, 3, sharex=True, sharey=True)
155: (4)                for (i, j), ax in np.ndenumerate(axs):
156: (8)                    xtop = ax.xaxis._major_tick_kw["label2On"]
157: (8)                    xbottom = ax.xaxis._major_tick_kw["label1On"]
158: (8)                    yleft = ax.yaxis._major_tick_kw["label1On"]
159: (8)                    yright = ax.yaxis._major_tick_kw["label2On"]
160: (8)                    assert xtop == (top and i == 0)
161: (8)                    assert xbottom == (bottom and i == 2)
162: (8)                    assert yleft == (left and j == 0)
163: (8)                    assert yright == (right and j == 2)
164: (0)            @pytest.mark.parametrize("xlabel_position", ["bottom", "top"])
165: (0)            @pytest.mark.parametrize("ylabel_position", ["left", "right"])
166: (0)            def test_subplots_hide_axislabels(xlabel_position, ylabel_position):
167: (4)                axs = plt.figure().subplots(3, 3, sharex=True, sharey=True)
168: (4)                for (i, j), ax in np.ndenumerate(axs):
169: (8)                    ax.set(xlabel="foo", ylabel="bar")
170: (8)                    ax.xaxis.set_label_position(xlabel_position)
171: (8)                    ax.yaxis.set_label_position(ylabel_position)
172: (8)                    ax.label_outer()
173: (8)                    assert bool(ax.get_xlabel()) == (
174: (12)                       xlabel_position == "bottom" and i == 2
175: (12)                       or xlabel_position == "top" and i == 0)
176: (8)                    assert bool(ax.get_ylabel()) == (
177: (12)                       ylabel_position == "left" and j == 0
178: (12)                       or ylabel_position == "right" and j == 2)
179: (0)            def test_get_gridspec():
180: (4)                fig, ax = plt.subplots()
181: (4)                assert ax.get_subplotspec().get_gridspec() == ax.get_gridspec()
182: (0)            def test_dont_mutate_kwargs():
183: (4)                subplot_kw = {'sharex': 'all'}
184: (4)                gridspec_kw = {'width_ratios': [1, 2]}
185: (4)                fig, ax = plt.subplots(1, 2, subplot_kw=subplot_kw,
186: (27)                                      gridspec_kw=gridspec_kw)
187: (4)                assert subplot_kw == {'sharex': 'all'}
188: (4)                assert gridspec_kw == {'width_ratios': [1, 2]}
189: (0)            @pytest.mark.parametrize("width_ratios", [None, [1, 3, 2]])
190: (0)            @pytest.mark.parametrize("height_ratios", [None, [1, 2]])
191: (0)            @check_figures_equal(extensions=['png'])
192: (0)            def test_width_and_height_ratios(fig_test, fig_ref,
193: (33)                                             height_ratios, width_ratios):
194: (4)                fig_test.subplots(2, 3, height_ratios=height_ratios,
195: (22)                                  width_ratios=width_ratios)
196: (4)                fig_ref.subplots(2, 3, gridspec_kw={
197: (21)                                 'height_ratios': height_ratios,
198: (21)                                 'width_ratios': width_ratios})
199: (0)            @pytest.mark.parametrize("width_ratios", [None, [1, 3, 2]])
200: (0)            @pytest.mark.parametrize("height_ratios", [None, [1, 2]])
201: (0)            @check_figures_equal(extensions=['png'])
202: (0)            def test_width_and_height_ratios_mosaic(fig_test, fig_ref,
203: (40)                                                   height_ratios, width_ratios):
204: (4)                mosaic_spec = [['A', 'B', 'B'], ['A', 'C', 'D']]
205: (4)                fig_test.subplot_mosaic(mosaic_spec, height_ratios=height_ratios,
206: (28)                                       width_ratios=width_ratios)
207: (4)                fig_ref.subplot_mosaic(mosaic_spec, gridspec_kw={
208: (27)                                      'height_ratios': height_ratios,
```

```
209: (27)                                    'width_ratios': width_ratios})
210: (0)            @pytest.mark.parametrize('method,args', [
211: (4)                ('subplots', (2, 3)),
212: (4)                ('subplot_mosaic', ('abc;def', ))
213: (4)                ]
214: (0)            )
215: (0)            def test_ratio_overlapping_kws(method, args):
216: (4)                with pytest.raises(ValueError, match='height_ratios'):
217: (8)                    getattr(plt, method)(*args, height_ratios=[1, 2],
218: (29)                                         gridspec_kw={'height_ratios': [1, 2]})
219: (4)                with pytest.raises(ValueError, match='width_ratios'):
220: (8)                    getattr(plt, method)(*args, width_ratios=[1, 2, 3],
221: (29)                                         gridspec_kw={'width_ratios': [1, 2, 3]})
222: (0)            def test_old_subplot_compat():
223: (4)                fig = plt.figure()
224: (4)                assert isinstance(fig.add_subplot(), SubplotBase)
225: (4)                assert not isinstance(fig.add_axes(rect=[0, 0, 1, 1]), SubplotBase)
226: (4)                with pytest.raises(TypeError):
227: (8)                    Axes(fig, [0, 0, 1, 1], rect=[0, 0, 1, 1])

----------------------------------------

File 217 - test_table.py:

1: (0)              import datetime
2: (0)              from unittest.mock import Mock
3: (0)              import numpy as np
4: (0)              import pytest
5: (0)              import matplotlib.pyplot as plt
6: (0)              import matplotlib as mpl
7: (0)              from matplotlib.path import Path
8: (0)              from matplotlib.table import CustomCell, Table
9: (0)              from matplotlib.testing.decorators import image_comparison,
check_figures_equal
10: (0)             from matplotlib.transforms import Bbox
11: (0)             import matplotlib.units as munits
12: (0)             def test_non_square():
13: (4)                 cellcolors = ['b', 'r']
14: (4)                 plt.table(cellColours=cellcolors)
15: (0)             @image_comparison(['table_zorder.png'], remove_text=True)
16: (0)             def test_zorder():
17: (4)                 data = [[66386, 174296],
18: (12)                        [58230, 381139]]
19: (4)                 colLabels = ('Freeze', 'Wind')
20: (4)                 rowLabels = ['%d year' % x for x in (100, 50)]
21: (4)                 cellText = []
22: (4)                 yoff = np.zeros(len(colLabels))
23: (4)                 for row in reversed(data):
24: (8)                     yoff += row
25: (8)                     cellText.append(['%1.1f' % (x/1000.0) for x in yoff])
26: (4)                 t = np.linspace(0, 2*np.pi, 100)
27: (4)                 plt.plot(t, np.cos(t), lw=4, zorder=2)
28: (4)                 plt.table(cellText=cellText,
29: (14)                          rowLabels=rowLabels,
30: (14)                          colLabels=colLabels,
31: (14)                          loc='center',
32: (14)                          zorder=-2,
33: (14)                          )
34: (4)                 plt.table(cellText=cellText,
35: (14)                          rowLabels=rowLabels,
36: (14)                          colLabels=colLabels,
37: (14)                          loc='upper center',
38: (14)                          zorder=4,
39: (14)                          )
40: (4)                 plt.yticks([])
41: (0)             @image_comparison(['table_labels.png'])
42: (0)             def test_label_colours():
43: (4)                 dim = 3
44: (4)                 c = np.linspace(0, 1, dim)
```

```
 45: (4)                 colours = plt.cm.RdYlGn(c)
 46: (4)                 cellText = [['1'] * dim] * dim
 47: (4)                 fig = plt.figure()
 48: (4)                 ax1 = fig.add_subplot(4, 1, 1)
 49: (4)                 ax1.axis('off')
 50: (4)                 ax1.table(cellText=cellText,
 51: (14)                          rowColours=colours,
 52: (14)                          loc='best')
 53: (4)                 ax2 = fig.add_subplot(4, 1, 2)
 54: (4)                 ax2.axis('off')
 55: (4)                 ax2.table(cellText=cellText,
 56: (14)                          rowColours=colours,
 57: (14)                          rowLabels=['Header'] * dim,
 58: (14)                          loc='best')
 59: (4)                 ax3 = fig.add_subplot(4, 1, 3)
 60: (4)                 ax3.axis('off')
 61: (4)                 ax3.table(cellText=cellText,
 62: (14)                          colColours=colours,
 63: (14)                          loc='best')
 64: (4)                 ax4 = fig.add_subplot(4, 1, 4)
 65: (4)                 ax4.axis('off')
 66: (4)                 ax4.table(cellText=cellText,
 67: (14)                          colColours=colours,
 68: (14)                          colLabels=['Header'] * dim,
 69: (14)                          loc='best')
 70: (0)             @image_comparison(['table_cell_manipulation.png'], remove_text=True)
 71: (0)             def test_diff_cell_table():
 72: (4)                 cells = ('horizontal', 'vertical', 'open', 'closed', 'T', 'R', 'B', 'L')
 73: (4)                 cellText = [['1'] * len(cells)] * 2
 74: (4)                 colWidths = [0.1] * len(cells)
 75: (4)                 _, axs = plt.subplots(nrows=len(cells), figsize=(4, len(cells)+1))
 76: (4)                 for ax, cell in zip(axs, cells):
 77: (8)                     ax.table(
 78: (16)                            colWidths=colWidths,
 79: (16)                            cellText=cellText,
 80: (16)                            loc='center',
 81: (16)                            edges=cell,
 82: (16)                            )
 83: (8)                     ax.axis('off')
 84: (4)                 plt.tight_layout()
 85: (0)             def test_customcell():
 86: (4)                 types = ('horizontal', 'vertical', 'open', 'closed', 'T', 'R', 'B', 'L')
 87: (4)                 codes = (
 88: (8)                     (Path.MOVETO, Path.LINETO, Path.MOVETO, Path.LINETO, Path.MOVETO),
 89: (8)                     (Path.MOVETO, Path.MOVETO, Path.LINETO, Path.MOVETO, Path.LINETO),
 90: (8)                     (Path.MOVETO, Path.MOVETO, Path.MOVETO, Path.MOVETO, Path.MOVETO),
 91: (8)                     (Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO, Path.CLOSEPOLY),
 92: (8)                     (Path.MOVETO, Path.MOVETO, Path.MOVETO, Path.LINETO, Path.MOVETO),
 93: (8)                     (Path.MOVETO, Path.MOVETO, Path.LINETO, Path.MOVETO, Path.MOVETO),
 94: (8)                     (Path.MOVETO, Path.LINETO, Path.MOVETO, Path.MOVETO, Path.MOVETO),
 95: (8)                     (Path.MOVETO, Path.MOVETO, Path.MOVETO, Path.MOVETO, Path.LINETO),
 96: (8)                     )
 97: (4)                 for t, c in zip(types, codes):
 98: (8)                     cell = CustomCell((0, 0), visible_edges=t, width=1, height=1)
 99: (8)                     code = tuple(s for _, s in cell.get_path().iter_segments())
100: (8)                     assert c == code
101: (0)             @image_comparison(['table_auto_column.png'])
102: (0)             def test_auto_column():
103: (4)                 fig = plt.figure()
104: (4)                 ax1 = fig.add_subplot(4, 1, 1)
105: (4)                 ax1.axis('off')
106: (4)                 tb1 = ax1.table(
107: (8)                     cellText=[['Fit Text', 2],
108: (18)                              ['very long long text, Longer text than default', 1]],
109: (8)                     rowLabels=["A", "B"],
110: (8)                     colLabels=["Col1", "Col2"],
111: (8)                     loc="center")
112: (4)                 tb1.auto_set_font_size(False)
113: (4)                 tb1.set_fontsize(12)
```

```
(4)                 tb1.auto_set_column_width([-1, 0, 1])
(4)                 ax2 = fig.add_subplot(4, 1, 2)
(4)                 ax2.axis('off')
(4)                 tb2 = ax2.table(
(8)                     cellText=[['Fit Text', 2],
(18)                               ['very long long text, Longer text than default', 1]],
(8)                     rowLabels=["A", "B"],
(8)                     colLabels=["Col1", "Col2"],
(8)                     loc="center")
(4)                 tb2.auto_set_font_size(False)
(4)                 tb2.set_fontsize(12)
(4)                 tb2.auto_set_column_width((-1, 0, 1))
(4)                 ax3 = fig.add_subplot(4, 1, 3)
(4)                 ax3.axis('off')
(4)                 tb3 = ax3.table(
(8)                     cellText=[['Fit Text', 2],
(18)                               ['very long long text, Longer text than default', 1]],
(8)                     rowLabels=["A", "B"],
(8)                     colLabels=["Col1", "Col2"],
(8)                     loc="center")
(4)                 tb3.auto_set_font_size(False)
(4)                 tb3.set_fontsize(12)
(4)                 tb3.auto_set_column_width(-1)
(4)                 tb3.auto_set_column_width(0)
(4)                 tb3.auto_set_column_width(1)
(4)                 ax4 = fig.add_subplot(4, 1, 4)
(4)                 ax4.axis('off')
(4)                 tb4 = ax4.table(
(8)                     cellText=[['Fit Text', 2],
(18)                               ['very long long text, Longer text than default', 1]],
(8)                     rowLabels=["A", "B"],
(8)                     colLabels=["Col1", "Col2"],
(8)                     loc="center")
(4)                 tb4.auto_set_font_size(False)
(4)                 tb4.set_fontsize(12)
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning,
(22)                                  match="'col' must be an int or sequence of ints"):
(8)                     tb4.auto_set_column_width("-101")  # type: ignore [arg-type]
(4)                 with pytest.warns(mpl.MatplotlibDeprecationWarning,
(22)                                  match="'col' must be an int or sequence of ints"):
(8)                     tb4.auto_set_column_width(["-101"])  # type: ignore [list-item]
(0)             def test_table_cells():
(4)                 fig, ax = plt.subplots()
(4)                 table = Table(ax)
(4)                 cell = table.add_cell(1, 2, 1, 1)
(4)                 assert isinstance(cell, CustomCell)
(4)                 assert cell is table[1, 2]
(4)                 cell2 = CustomCell((0, 0), 1, 2, visible_edges=None)
(4)                 table[2, 1] = cell2
(4)                 assert table[2, 1] is cell2
(4)                 table.properties()
(4)                 plt.setp(table)
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_table_bbox(fig_test, fig_ref):
(4)                 data = [[2, 3],
(12)                        [4, 5]]
(4)                 col_labels = ('Foo', 'Bar')
(4)                 row_labels = ('Ada', 'Bob')
(4)                 cell_text = [[f"{x}" for x in row] for row in data]
(4)                 ax_list = fig_test.subplots()
(4)                 ax_list.table(cellText=cell_text,
(18)                              rowLabels=row_labels,
(18)                              colLabels=col_labels,
(18)                              loc='center',
(18)                              bbox=[0.1, 0.2, 0.8, 0.6]
(18)                              )
(4)                 ax_bbox = fig_ref.subplots()
(4)                 ax_bbox.table(cellText=cell_text,
(18)                              rowLabels=row_labels,
```

```
183: (18)                               colLabels=col_labels,
184: (18)                               loc='center',
185: (18)                               bbox=Bbox.from_extents(0.1, 0.2, 0.9, 0.8)
186: (18)                               )
187: (0)            @check_figures_equal(extensions=['png'])
188: (0)            def test_table_unit(fig_test, fig_ref):
189: (4)                class FakeUnit:
190: (8)                    def __init__(self, thing):
191: (12)                       pass
192: (8)                    def __repr__(self):
193: (12)                       return "Hello"
194: (4)                fake_convertor = munits.ConversionInterface()
195: (4)                fake_convertor.convert = Mock(side_effect=lambda v, u, a: 0)
196: (4)                fake_convertor.default_units = Mock(side_effect=lambda v, a: None)
197: (4)                fake_convertor.axisinfo = Mock(side_effect=lambda u, a: munits.AxisInfo())
198: (4)                munits.registry[FakeUnit] = fake_convertor
199: (4)                data = [[FakeUnit("yellow"), FakeUnit(42)],
200: (12)                       [FakeUnit(datetime.datetime(1968, 8, 1)), FakeUnit(True)]]
201: (4)                fig_test.subplots().table(data)
202: (4)                fig_ref.subplots().table([["Hello", "Hello"], ["Hello", "Hello"]])
203: (4)                fig_test.canvas.draw()
204: (4)                fake_convertor.convert.assert_not_called()
205: (4)                munits.registry.pop(FakeUnit)
206: (4)                assert not munits.registry.get_converter(FakeUnit)

----------------------------------------

File 218 - test_testing.py:

1: (0)              import warnings
2: (0)              import pytest
3: (0)              import matplotlib.pyplot as plt
4: (0)              from matplotlib.testing.decorators import check_figures_equal
5: (0)              @pytest.mark.xfail(
6: (4)                  strict=True, reason="testing that warnings fail tests"
7: (0)              )
8: (0)              def test_warn_to_fail():
9: (4)                  warnings.warn("This should fail the test")
10: (0)             @pytest.mark.parametrize("a", [1])
11: (0)             @check_figures_equal(extensions=["png"])
12: (0)             @pytest.mark.parametrize("b", [1])
13: (0)             def test_parametrize_with_check_figure_equal(a, fig_ref, b, fig_test):
14: (4)                 assert a == b
15: (0)             def test_wrap_failure():
16: (4)                 with pytest.raises(ValueError, match="^The decorated function"):
17: (8)                     @check_figures_equal()
18: (8)                     def should_fail(test, ref):
19: (12)                        pass
20: (0)             @pytest.mark.xfail(raises=RuntimeError, strict=True,
21: (19)                            reason='Test for check_figures_equal test creating '
22: (26)                                   'new figures')
23: (0)             @check_figures_equal()
24: (0)             def test_check_figures_equal_extra_fig(fig_test, fig_ref):
25: (4)                 plt.figure()
26: (0)             @check_figures_equal()
27: (0)             def test_check_figures_equal_closed_fig(fig_test, fig_ref):
28: (4)                 fig = plt.figure()
29: (4)                 plt.close(fig)

----------------------------------------

File 219 - test_texmanager.py:

1: (0)              import os
2: (0)              from pathlib import Path
3: (0)              import re
4: (0)              import sys
5: (0)              import pytest
6: (0)              import matplotlib.pyplot as plt
```

```
7: (0)              from matplotlib.testing import subprocess_run_for_testing
8: (0)              from matplotlib.testing._markers import needs_usetex
9: (0)              from matplotlib.texmanager import TexManager
10: (0)             def test_fontconfig_preamble():
11: (4)                 """Test that the preamble is included in the source."""
12: (4)                 plt.rcParams['text.usetex'] = True
13: (4)                 src1 = TexManager()._get_tex_source("", fontsize=12)
14: (4)                 plt.rcParams['text.latex.preamble'] = '\\usepackage{txfonts}'
15: (4)                 src2 = TexManager()._get_tex_source("", fontsize=12)
16: (4)                 assert src1 != src2
17: (0)             @pytest.mark.parametrize(
18: (4)                 "rc, preamble, family", [
19: (8)                     ({"font.family": "sans-serif", "font.sans-serif": "helvetica"},
20: (9)                      r"\usepackage{helvet}", r"\sffamily"),
21: (8)                     ({"font.family": "serif", "font.serif": "palatino"},
22: (9)                      r"\usepackage{mathpazo}", r"\rmfamily"),
23: (8)                     ({"font.family": "cursive", "font.cursive": "zapf chancery"},
24: (9)                      r"\usepackage{chancery}", r"\rmfamily"),
25: (8)                     ({"font.family": "monospace", "font.monospace": "courier"},
26: (9)                      r"\usepackage{courier}", r"\ttfamily"),
27: (8)                     ({"font.family": "helvetica"}, r"\usepackage{helvet}", r"\sffamily"),
28: (8)                     ({"font.family": "palatino"}, r"\usepackage{mathpazo}", r"\rmfamily"),
29: (8)                     ({"font.family": "zapf chancery"},
30: (9)                      r"\usepackage{chancery}", r"\rmfamily"),
31: (8)                     ({"font.family": "courier"}, r"\usepackage{courier}", r"\ttfamily")
32: (4)                 ])
33: (0)             def test_font_selection(rc, preamble, family):
34: (4)                 plt.rcParams.update(rc)
35: (4)                 tm = TexManager()
36: (4)                 src = Path(tm.make_tex("hello, world", fontsize=12)).read_text()
37: (4)                 assert preamble in src
38: (4)                 assert [*re.findall(r"\\\w+family", src)] == [family]
39: (0)             @needs_usetex
40: (0)             def test_unicode_characters():
41: (4)                 plt.rcParams['text.usetex'] = True
42: (4)                 fig, ax = plt.subplots()
43: (4)                 ax.set_ylabel('\\textit{Velocity (\N{DEGREE SIGN}/sec)}')
44: (4)                 ax.set_xlabel('\N{VULGAR FRACTION ONE QUARTER}Öøæ')
45: (4)                 fig.canvas.draw()
46: (4)                 with pytest.raises(RuntimeError):
47: (8)                     ax.set_title('\N{SNOWMAN}')
48: (8)                     fig.canvas.draw()
49: (0)             @needs_usetex
50: (0)             def test_openin_any_paranoid():
51: (4)                 completed = subprocess_run_for_testing(
52: (8)                     [sys.executable, "-c",
53: (9)                      'import matplotlib.pyplot as plt;'
54: (9)                      'plt.rcParams.update({"text.usetex": True});'
55: (9)                      'plt.title("paranoid");'
56: (9)                      'plt.show(block=False);'],
57: (8)                     env={**os.environ, 'openin_any': 'p'}, check=True,
capture_output=True)
58: (4)                 assert completed.stderr == ""
```

File 220 - test_textpath.py:

```
1: (0)              import copy
2: (0)              from matplotlib.textpath import TextPath
3: (0)              def test_copy():
4: (4)                  tp = TextPath((0, 0), ".")
5: (4)                  assert copy.deepcopy(tp).vertices is not tp.vertices
6: (4)                  assert (copy.deepcopy(tp).vertices == tp.vertices).all()
7: (4)                  assert copy.copy(tp).vertices is tp.vertices
```

File 221 - test_text.py:

```
(0)              from datetime import datetime
(0)              import io
(0)              import warnings
(0)              import numpy as np
(0)              from numpy.testing import assert_almost_equal
(0)              from packaging.version import parse as parse_version
(0)              import pyparsing
(0)              import pytest
(0)              import matplotlib as mpl
(0)              from matplotlib.backend_bases import MouseEvent
(0)              from matplotlib.backends.backend_agg import RendererAgg
(0)              from matplotlib.figure import Figure
(0)              from matplotlib.font_manager import FontProperties
(0)              import matplotlib.patches as mpatches
(0)              import matplotlib.pyplot as plt
(0)              from matplotlib.gridspec import GridSpec
(0)              import matplotlib.transforms as mtransforms
(0)              from matplotlib.testing.decorators import check_figures_equal,
image_comparison
(0)              from matplotlib.testing._markers import needs_usetex
(0)              from matplotlib.text import Text, Annotation, OffsetFrom
(0)              pyparsing_version = parse_version(pyparsing.__version__)
(0)              @image_comparison(['font_styles'])
(0)              def test_font_styles():
(4)                  def find_matplotlib_font(**kw):
(8)                      prop = FontProperties(**kw)
(8)                      path = findfont(prop, directory=mpl.get_data_path())
(8)                      return FontProperties(fname=path)
(4)                  from matplotlib.font_manager import FontProperties, findfont
(4)                  warnings.filterwarnings(
(8)                      'ignore',
(8)                      r"findfont: Font family \[u?'Foo'\] not found. Falling back to .",
(8)                      UserWarning,
(8)                      module='matplotlib.font_manager')
(4)                  fig, ax = plt.subplots()
(4)                  normal_font = find_matplotlib_font(
(8)                      family="sans-serif",
(8)                      style="normal",
(8)                      variant="normal",
(8)                      size=14)
(4)                  a = ax.annotate(
(8)                      "Normal Font",
(8)                      (0.1, 0.1),
(8)                      xycoords='axes fraction',
(8)                      fontproperties=normal_font)
(4)                  assert a.get_fontname() == 'DejaVu Sans'
(4)                  assert a.get_fontstyle() == 'normal'
(4)                  assert a.get_fontvariant() == 'normal'
(4)                  assert a.get_weight() == 'normal'
(4)                  assert a.get_stretch() == 'normal'
(4)                  bold_font = find_matplotlib_font(
(8)                      family="Foo",
(8)                      style="normal",
(8)                      variant="normal",
(8)                      weight="bold",
(8)                      stretch=500,
(8)                      size=14)
(4)                  ax.annotate(
(8)                      "Bold Font",
(8)                      (0.1, 0.2),
(8)                      xycoords='axes fraction',
(8)                      fontproperties=bold_font)
(4)                  bold_italic_font = find_matplotlib_font(
(8)                      family="sans serif",
(8)                      style="italic",
(8)                      variant="normal",
(8)                      weight=750,
(8)                      stretch=500,
```

```
 68: (8)                     size=14)
 69: (4)                 ax.annotate(
 70: (8)                     "Bold Italic Font",
 71: (8)                     (0.1, 0.3),
 72: (8)                     xycoords='axes fraction',
 73: (8)                     fontproperties=bold_italic_font)
 74: (4)                 light_font = find_matplotlib_font(
 75: (8)                     family="sans-serif",
 76: (8)                     style="normal",
 77: (8)                     variant="normal",
 78: (8)                     weight=200,
 79: (8)                     stretch=500,
 80: (8)                     size=14)
 81: (4)                 ax.annotate(
 82: (8)                     "Light Font",
 83: (8)                     (0.1, 0.4),
 84: (8)                     xycoords='axes fraction',
 85: (8)                     fontproperties=light_font)
 86: (4)                 condensed_font = find_matplotlib_font(
 87: (8)                     family="sans-serif",
 88: (8)                     style="normal",
 89: (8)                     variant="normal",
 90: (8)                     weight=500,
 91: (8)                     stretch=100,
 92: (8)                     size=14)
 93: (4)                 ax.annotate(
 94: (8)                     "Condensed Font",
 95: (8)                     (0.1, 0.5),
 96: (8)                     xycoords='axes fraction',
 97: (8)                     fontproperties=condensed_font)
 98: (4)                 ax.set_xticks([])
 99: (4)                 ax.set_yticks([])
100: (0)             @image_comparison(['multiline'])
101: (0)             def test_multiline():
102: (4)                 plt.figure()
103: (4)                 ax = plt.subplot(1, 1, 1)
104: (4)                 ax.set_title("multiline\ntext alignment")
105: (4)                 plt.text(
106: (8)                     0.2, 0.5, "TpTpTp\n$M$\nTpTpTp", size=20, ha="center", va="top")
107: (4)                 plt.text(
108: (8)                     0.5, 0.5, "TpTpTp\n$M^{M^{M^{M}}}$\nTpTpTp", size=20,
109: (8)                     ha="center", va="top")
110: (4)                 plt.text(
111: (8)                     0.8, 0.5, "TpTpTp\n$M_{q_{q_{q}}}$\nTpTpTp", size=20,
112: (8)                     ha="center", va="top")
113: (4)                 plt.xlim(0, 1)
114: (4)                 plt.ylim(0, 0.8)
115: (4)                 ax.set_xticks([])
116: (4)                 ax.set_yticks([])
117: (0)             @image_comparison(['multiline2'], style='mpl20')
118: (0)             def test_multiline2():
119: (4)                 plt.rcParams['text.kerning_factor'] = 6
120: (4)                 fig, ax = plt.subplots()
121: (4)                 ax.set_xlim([0, 1.4])
122: (4)                 ax.set_ylim([0, 2])
123: (4)                 ax.axhline(0.5, color='C2', linewidth=0.3)
124: (4)                 sts = ['Line', '2 Lineg\n 2 Lg', '$\\sum_i x $', 'hi $\\sum_i x $\ntest',
125: (11)                       'test\n $\\sum_i x $', '$\\sum_i x $\n $\\sum_i x $']
126: (4)                 renderer = fig.canvas.get_renderer()
127: (4)                 def draw_box(ax, tt):
128: (8)                     r = mpatches.Rectangle((0, 0), 1, 1, clip_on=False,
129: (31)                                            transform=ax.transAxes)
130: (8)                     r.set_bounds(
131: (12)                        tt.get_window_extent(renderer)
132: (12)                        .transformed(ax.transAxes.inverted())
133: (12)                        .bounds)
134: (8)                     ax.add_patch(r)
135: (4)                 horal = 'left'
136: (4)                 for nn, st in enumerate(sts):
```

```
137: (8)                    tt = ax.text(0.2 * nn + 0.1, 0.5, st, horizontalalignment=horal,
138: (21)                                verticalalignment='bottom')
139: (8)                    draw_box(ax, tt)
140: (4)                ax.text(1.2, 0.5, 'Bottom align', color='C2')
141: (4)                ax.axhline(1.3, color='C2', linewidth=0.3)
142: (4)                for nn, st in enumerate(sts):
143: (8)                    tt = ax.text(0.2 * nn + 0.1, 1.3, st, horizontalalignment=horal,
144: (21)                                verticalalignment='top')
145: (8)                    draw_box(ax, tt)
146: (4)                ax.text(1.2, 1.3, 'Top align', color='C2')
147: (4)                ax.axhline(1.8, color='C2', linewidth=0.3)
148: (4)                for nn, st in enumerate(sts):
149: (8)                    tt = ax.text(0.2 * nn + 0.1, 1.8, st, horizontalalignment=horal,
150: (21)                                verticalalignment='baseline')
151: (8)                    draw_box(ax, tt)
152: (4)                ax.text(1.2, 1.8, 'Baseline align', color='C2')
153: (4)                ax.axhline(0.1, color='C2', linewidth=0.3)
154: (4)                for nn, st in enumerate(sts):
155: (8)                    tt = ax.text(0.2 * nn + 0.1, 0.1, st, horizontalalignment=horal,
156: (21)                                verticalalignment='bottom', rotation=20)
157: (8)                    draw_box(ax, tt)
158: (4)                ax.text(1.2, 0.1, 'Bot align, rot20', color='C2')
159: (0)            @image_comparison(['antialiased.png'], style='mpl20')
160: (0)            def test_antialiasing():
161: (4)                mpl.rcParams['text.antialiased'] = False  # Passed arguments should
override.
162: (4)                fig = plt.figure(figsize=(5.25, 0.75))
163: (4)                fig.text(0.3, 0.75, "antialiased", horizontalalignment='center',
164: (13)                        verticalalignment='center', antialiased=True)
165: (4)                fig.text(0.3, 0.25, r"$\sqrt{x}$", horizontalalignment='center',
166: (13)                        verticalalignment='center', antialiased=True)
167: (4)                mpl.rcParams['text.antialiased'] = True  # Passed arguments should
override.
168: (4)                fig.text(0.7, 0.75, "not antialiased", horizontalalignment='center',
169: (13)                        verticalalignment='center', antialiased=False)
170: (4)                fig.text(0.7, 0.25, r"$\sqrt{x}$", horizontalalignment='center',
171: (13)                        verticalalignment='center', antialiased=False)
172: (4)                mpl.rcParams['text.antialiased'] = False  # Should not affect existing
text.
173: (0)            def test_afm_kerning():
174: (4)                fn = mpl.font_manager.findfont("Helvetica", fontext="afm")
175: (4)                with open(fn, 'rb') as fh:
176: (8)                    afm = mpl._afm.AFM(fh)
177: (4)                assert afm.string_width_height('VAVAVAVAVAVA') == (7174.0, 718)
178: (0)            @image_comparison(['text_contains.png'])
179: (0)            def test_contains():
180: (4)                fig = plt.figure()
181: (4)                ax = plt.axes()
182: (4)                mevent = MouseEvent('button_press_event', fig.canvas, 0.5, 0.5, 1, None)
183: (4)                xs = np.linspace(0.25, 0.75, 30)
184: (4)                ys = np.linspace(0.25, 0.75, 30)
185: (4)                xs, ys = np.meshgrid(xs, ys)
186: (4)                txt = plt.text(
187: (8)                    0.5, 0.4, 'hello world', ha='center', fontsize=30, rotation=30)
188: (4)                fig.canvas.draw()
189: (4)                for x, y in zip(xs.flat, ys.flat):
190: (8)                    mevent.x, mevent.y = plt.gca().transAxes.transform([x, y])
191: (8)                    contains, _ = txt.contains(mevent)
192: (8)                    color = 'yellow' if contains else 'red'
193: (8)                    vl = ax.viewLim.frozen()
194: (8)                    ax.plot(x, y, 'o', color=color)
195: (8)                    ax.viewLim.set(vl)
196: (0)            def test_annotation_contains():
197: (4)                fig, ax = plt.subplots()
198: (4)                ann = ax.annotate(
199: (8)                    "hello", xy=(.4, .4), xytext=(.6, .6), arrowprops={"arrowstyle": "-
>"})
200: (4)                fig.canvas.draw()  # Needed for the same reason as in test_contains.
201: (4)                event = MouseEvent(
```

```
(8)                     "button_press_event", fig.canvas, *ax.transData.transform((.5, .6)))
(4)                 assert ann.contains(event) == (False, {})
(0)             @pytest.mark.parametrize('err, xycoords, match', (
(4)                 (TypeError, print, "xycoords callable must return a BboxBase or Transform, not a"),
(4)                 (TypeError, [0, 0], r"'xycoords' must be an instance of str, tuple"),
(4)                 (ValueError, "foo", "'foo' is not a valid coordinate"),
(4)                 (ValueError, "foo bar", "'foo bar' is not a valid coordinate"),
(4)                 (ValueError, "offset foo", "xycoords cannot be an offset coordinate"),
(4)                 (ValueError, "axes foo", "'foo' is not a recognized unit"),
(0)             ))
(0)             def test_annotate_errors(err, xycoords, match):
(4)                 fig, ax = plt.subplots()
(4)                 with pytest.raises(err, match=match):
(8)                     ax.annotate('xy', (0, 0), xytext=(0.5, 0.5), xycoords=xycoords)
(8)                     fig.canvas.draw()
(0)             @image_comparison(['titles'])
(0)             def test_titles():
(4)                 plt.figure()
(4)                 ax = plt.subplot(1, 1, 1)
(4)                 ax.set_title("left title", loc="left")
(4)                 ax.set_title("right title", loc="right")
(4)                 ax.set_xticks([])
(4)                 ax.set_yticks([])
(0)             @image_comparison(['text_alignment'], style='mpl20')
(0)             def test_alignment():
(4)                 plt.figure()
(4)                 ax = plt.subplot(1, 1, 1)
(4)                 x = 0.1
(4)                 for rotation in (0, 30):
(8)                     for alignment in ('top', 'bottom', 'baseline', 'center'):
(12)                        ax.text(
(16)                            x, 0.5, alignment + " Tj", va=alignment, rotation=rotation,
(16)                            bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.5))
(12)                        ax.text(
(16)                            x, 1.0, r'$\sum_{i=0}^{j}$', va=alignment, rotation=rotation)
(12)                        x += 0.1
(4)                 ax.plot([0, 1], [0.5, 0.5])
(4)                 ax.plot([0, 1], [1.0, 1.0])
(4)                 ax.set_xlim(0, 1)
(4)                 ax.set_ylim(0, 1.5)
(4)                 ax.set_xticks([])
(4)                 ax.set_yticks([])
(0)             @image_comparison(['axes_titles.png'])
(0)             def test_axes_titles():
(4)                 plt.figure()
(4)                 ax = plt.subplot(1, 1, 1)
(4)                 ax.set_title('center', loc='center', fontsize=20, fontweight=700)
(4)                 ax.set_title('left', loc='left', fontsize=12, fontweight=400)
(4)                 ax.set_title('right', loc='right', fontsize=12, fontweight=400)
(0)             def test_set_position():
(4)                 fig, ax = plt.subplots()
(4)                 ann = ax.annotate(
(8)                     'test', (0, 0), xytext=(0, 0), textcoords='figure pixels')
(4)                 fig.canvas.draw()
(4)                 init_pos = ann.get_window_extent(fig.canvas.renderer)
(4)                 shift_val = 15
(4)                 ann.set_position((shift_val, shift_val))
(4)                 fig.canvas.draw()
(4)                 post_pos = ann.get_window_extent(fig.canvas.renderer)
(4)                 for a, b in zip(init_pos.min, post_pos.min):
(8)                     assert a + shift_val == b
(4)                 ann = ax.annotate(
(8)                     'test', (0, 0), xytext=(0, 0), textcoords='figure pixels')
(4)                 fig.canvas.draw()
(4)                 init_pos = ann.get_window_extent(fig.canvas.renderer)
(4)                 shift_val = 15
(4)                 ann.xyann = (shift_val, shift_val)
(4)                 fig.canvas.draw()
```

```
(4)                 post_pos = ann.get_window_extent(fig.canvas.renderer)
(4)                 for a, b in zip(init_pos.min, post_pos.min):
(8)                     assert a + shift_val == b
(0)             def test_char_index_at():
(4)                 fig = plt.figure()
(4)                 text = fig.text(0.1, 0.9, "")
(4)                 text.set_text("i")
(4)                 bbox = text.get_window_extent()
(4)                 size_i = bbox.x1 - bbox.x0
(4)                 text.set_text("m")
(4)                 bbox = text.get_window_extent()
(4)                 size_m = bbox.x1 - bbox.x0
(4)                 text.set_text("iiiimmmm")
(4)                 bbox = text.get_window_extent()
(4)                 origin = bbox.x0
(4)                 assert text._char_index_at(origin - size_i) == 0  # left of first char
(4)                 assert text._char_index_at(origin) == 0
(4)                 assert text._char_index_at(origin + 0.499*size_i) == 0
(4)                 assert text._char_index_at(origin + 0.501*size_i) == 1
(4)                 assert text._char_index_at(origin + size_i*3) == 3
(4)                 assert text._char_index_at(origin + size_i*4 + size_m*3) == 7
(4)                 assert text._char_index_at(origin + size_i*4 + size_m*4) == 8
(4)                 assert text._char_index_at(origin + size_i*4 + size_m*10) == 8
(0)             @pytest.mark.parametrize('text', ['', 'O'], ids=['empty', 'non-empty'])
(0)             def test_non_default_dpi(text):
(4)                 fig, ax = plt.subplots()
(4)                 t1 = ax.text(0.5, 0.5, text, ha='left', va='bottom')
(4)                 fig.canvas.draw()
(4)                 dpi = fig.dpi
(4)                 bbox1 = t1.get_window_extent()
(4)                 bbox2 = t1.get_window_extent(dpi=dpi * 10)
(4)                 np.testing.assert_allclose(bbox2.get_points(), bbox1.get_points() * 10,
(31)                                           rtol=5e-2)
(4)                 assert fig.dpi == dpi
(0)             def test_get_rotation_string():
(4)                 assert Text(rotation='horizontal').get_rotation() == 0.
(4)                 assert Text(rotation='vertical').get_rotation() == 90.
(0)             def test_get_rotation_float():
(4)                 for i in [15., 16.70, 77.4]:
(8)                     assert Text(rotation=i).get_rotation() == i
(0)             def test_get_rotation_int():
(4)                 for i in [67, 16, 41]:
(8)                     assert Text(rotation=i).get_rotation() == float(i)
(0)             def test_get_rotation_raises():
(4)                 with pytest.raises(ValueError):
(8)                     Text(rotation='hozirontal')
(0)             def test_get_rotation_none():
(4)                 assert Text(rotation=None).get_rotation() == 0.0
(0)             def test_get_rotation_mod360():
(4)                 for i, j in zip([360., 377., 720+177.2], [0., 17., 177.2]):
(8)                     assert_almost_equal(Text(rotation=i).get_rotation(), j)
(0)             @pytest.mark.parametrize("ha", ["center", "right", "left"])
(0)             @pytest.mark.parametrize("va", ["center", "top", "bottom",
(32)                                            "baseline", "center_baseline"])
(0)             def test_null_rotation_with_rotation_mode(ha, va):
(4)                 fig, ax = plt.subplots()
(4)                 kw = dict(rotation=0, va=va, ha=ha)
(4)                 t0 = ax.text(.5, .5, 'test', rotation_mode='anchor', **kw)
(4)                 t1 = ax.text(.5, .5, 'test', rotation_mode='default', **kw)
(4)                 fig.canvas.draw()
(4)
assert_almost_equal(t0.get_window_extent(fig.canvas.renderer).get_points(),
(24)
t1.get_window_extent(fig.canvas.renderer).get_points())
(0)             @image_comparison(['text_bboxclip'])
(0)             def test_bbox_clipping():
(4)                 plt.text(0.9, 0.2, 'Is bbox clipped?', backgroundcolor='r', clip_on=True)
(4)                 t = plt.text(0.9, 0.5, 'Is fancy bbox clipped?', clip_on=True)
(4)                 t.set_bbox({"boxstyle": "round, pad=0.1"})
```

```
(0)             @image_comparison(['annotation_negative_ax_coords.png'])
(0)             def test_annotation_negative_ax_coords():
(4)                 fig, ax = plt.subplots()
(4)                 ax.annotate('+ pts',
(16)                            xytext=[30, 20], textcoords='axes points',
(16)                            xy=[30, 20], xycoords='axes points', fontsize=32)
(4)                 ax.annotate('- pts',
(16)                            xytext=[30, -20], textcoords='axes points',
(16)                            xy=[30, -20], xycoords='axes points', fontsize=32,
(16)                            va='top')
(4)                 ax.annotate('+ frac',
(16)                            xytext=[0.75, 0.05], textcoords='axes fraction',
(16)                            xy=[0.75, 0.05], xycoords='axes fraction', fontsize=32)
(4)                 ax.annotate('- frac',
(16)                            xytext=[0.75, -0.05], textcoords='axes fraction',
(16)                            xy=[0.75, -0.05], xycoords='axes fraction', fontsize=32,
(16)                            va='top')
(4)                 ax.annotate('+ pixels',
(16)                            xytext=[160, 25], textcoords='axes pixels',
(16)                            xy=[160, 25], xycoords='axes pixels', fontsize=32)
(4)                 ax.annotate('- pixels',
(16)                            xytext=[160, -25], textcoords='axes pixels',
(16)                            xy=[160, -25], xycoords='axes pixels', fontsize=32,
(16)                            va='top')
(0)             @image_comparison(['annotation_negative_fig_coords.png'])
(0)             def test_annotation_negative_fig_coords():
(4)                 fig, ax = plt.subplots()
(4)                 ax.annotate('+ pts',
(16)                            xytext=[10, 120], textcoords='figure points',
(16)                            xy=[10, 120], xycoords='figure points', fontsize=32)
(4)                 ax.annotate('- pts',
(16)                            xytext=[-10, 180], textcoords='figure points',
(16)                            xy=[-10, 180], xycoords='figure points', fontsize=32,
(16)                            va='top')
(4)                 ax.annotate('+ frac',
(16)                            xytext=[0.05, 0.55], textcoords='figure fraction',
(16)                            xy=[0.05, 0.55], xycoords='figure fraction', fontsize=32)
(4)                 ax.annotate('- frac',
(16)                            xytext=[-0.05, 0.5], textcoords='figure fraction',
(16)                            xy=[-0.05, 0.5], xycoords='figure fraction', fontsize=32,
(16)                            va='top')
(4)                 ax.annotate('+ pixels',
(16)                            xytext=[50, 50], textcoords='figure pixels',
(16)                            xy=[50, 50], xycoords='figure pixels', fontsize=32)
(4)                 ax.annotate('- pixels',
(16)                            xytext=[-50, 100], textcoords='figure pixels',
(16)                            xy=[-50, 100], xycoords='figure pixels', fontsize=32,
(16)                            va='top')
(0)             def test_text_stale():
(4)                 fig, (ax1, ax2) = plt.subplots(1, 2)
(4)                 plt.draw_all()
(4)                 assert not ax1.stale
(4)                 assert not ax2.stale
(4)                 assert not fig.stale
(4)                 txt1 = ax1.text(.5, .5, 'aardvark')
(4)                 assert ax1.stale
(4)                 assert txt1.stale
(4)                 assert fig.stale
(4)                 ann1 = ax2.annotate('aardvark', xy=[.5, .5])
(4)                 assert ax2.stale
(4)                 assert ann1.stale
(4)                 assert fig.stale
(4)                 plt.draw_all()
(4)                 assert not ax1.stale
(4)                 assert not ax2.stale
(4)                 assert not fig.stale
(0)             @image_comparison(['agg_text_clip.png'])
(0)             def test_agg_text_clip():
(4)                 np.random.seed(1)
```

```
(4)         fig, (ax1, ax2) = plt.subplots(2)
(4)         for x, y in np.random.rand(10, 2):
(8)             ax1.text(x, y, "foo", clip_on=True)
(8)             ax2.text(x, y, "foo")
(0)     def test_text_size_binding():
(4)         mpl.rcParams['font.size'] = 10
(4)         fp = mpl.font_manager.FontProperties(size='large')
(4)         sz1 = fp.get_size_in_points()
(4)         mpl.rcParams['font.size'] = 100
(4)         assert sz1 == fp.get_size_in_points()
(0)     @image_comparison(['font_scaling.pdf'])
(0)     def test_font_scaling():
(4)         mpl.rcParams['pdf.fonttype'] = 42
(4)         fig, ax = plt.subplots(figsize=(6.4, 12.4))
(4)         ax.xaxis.set_major_locator(plt.NullLocator())
(4)         ax.yaxis.set_major_locator(plt.NullLocator())
(4)         ax.set_ylim(-10, 600)
(4)         for i, fs in enumerate(range(4, 43, 2)):
(8)             ax.text(0.1, i*30, f"{fs} pt font size", fontsize=fs)
(0)     @pytest.mark.parametrize('spacing1, spacing2', [(0.4, 2), (2, 0.4), (2, 2)])
(0)     def test_two_2line_texts(spacing1, spacing2):
(4)         text_string = 'line1\nline2'
(4)         fig = plt.figure()
(4)         renderer = fig.canvas.get_renderer()
(4)         text1 = fig.text(0.25, 0.5, text_string, linespacing=spacing1)
(4)         text2 = fig.text(0.25, 0.5, text_string, linespacing=spacing2)
(4)         fig.canvas.draw()
(4)         box1 = text1.get_window_extent(renderer=renderer)
(4)         box2 = text2.get_window_extent(renderer=renderer)
(4)         assert box1.width == box2.width
(4)         if spacing1 == spacing2:
(8)             assert box1.height == box2.height
(4)         else:
(8)             assert box1.height != box2.height
(0)     def test_validate_linespacing():
(4)         with pytest.raises(TypeError):
(8)             plt.text(.25, .5, "foo", linespacing="abc")
(0)     def test_nonfinite_pos():
(4)         fig, ax = plt.subplots()
(4)         ax.text(0, np.nan, 'nan')
(4)         ax.text(np.inf, 0, 'inf')
(4)         fig.canvas.draw()
(0)     def test_hinting_factor_backends():
(4)         plt.rcParams['text.hinting_factor'] = 1
(4)         fig = plt.figure()
(4)         t = fig.text(0.5, 0.5, 'some text')
(4)         fig.savefig(io.BytesIO(), format='svg')
(4)         expected = t.get_window_extent().intervalx
(4)         fig.savefig(io.BytesIO(), format='png')
(4)         np.testing.assert_allclose(t.get_window_extent().intervalx, expected,
(31)                                    rtol=0.1)
(0)     @needs_usetex
(0)     def test_usetex_is_copied():
(4)         fig = plt.figure()
(4)         plt.rcParams["text.usetex"] = False
(4)         ax1 = fig.add_subplot(121)
(4)         plt.rcParams["text.usetex"] = True
(4)         ax2 = fig.add_subplot(122)
(4)         fig.canvas.draw()
(4)         for ax, usetex in [(ax1, False), (ax2, True)]:
(8)             for t in ax.xaxis.majorTicks:
(12)                assert t.label1.get_usetex() == usetex
(0)     @needs_usetex
(0)     def test_single_artist_usetex():
(4)         fig = plt.figure()
(4)         fig.text(.5, .5, r"$\frac12$", usetex=True)
(4)         fig.canvas.draw()
(0)     @pytest.mark.parametrize("fmt", ["png", "pdf", "svg"])
(0)     def test_single_artist_usenotex(fmt):
```

```
475: (4)                plt.rcParams["text.usetex"] = True
476: (4)                fig = plt.figure()
477: (4)                fig.text(.5, .5, "2_2_2", usetex=False)
478: (4)                fig.savefig(io.BytesIO(), format=fmt)
479: (0)            @image_comparison(['text_as_path_opacity.svg'])
480: (0)            def test_text_as_path_opacity():
481: (4)                plt.figure()
482: (4)                plt.gca().set_axis_off()
483: (4)                plt.text(0.25, 0.25, 'c', color=(0, 0, 0, 0.5))
484: (4)                plt.text(0.25, 0.5, 'a', alpha=0.5)
485: (4)                plt.text(0.25, 0.75, 'x', alpha=0.5, color=(0, 0, 0, 1))
486: (0)            @image_comparison(['text_as_text_opacity.svg'])
487: (0)            def test_text_as_text_opacity():
488: (4)                mpl.rcParams['svg.fonttype'] = 'none'
489: (4)                plt.figure()
490: (4)                plt.gca().set_axis_off()
491: (4)                plt.text(0.25, 0.25, '50% using `color`', color=(0, 0, 0, 0.5))
492: (4)                plt.text(0.25, 0.5, '50% using `alpha`', alpha=0.5)
493: (4)                plt.text(0.25, 0.75, '50% using `alpha` and 100% `color`', alpha=0.5,
494: (13)                         color=(0, 0, 0, 1))
495: (0)            def test_text_repr():
496: (4)                plt.plot(['A', 'B'], [1, 2])
497: (4)                repr(plt.text(['A'], 0.5, 'Boo'))
498: (0)            def test_annotation_update():
499: (4)                fig, ax = plt.subplots(1, 1)
500: (4)                an = ax.annotate('annotation', xy=(0.5, 0.5))
501: (4)                extent1 = an.get_window_extent(fig.canvas.get_renderer())
502: (4)                fig.tight_layout()
503: (4)                extent2 = an.get_window_extent(fig.canvas.get_renderer())
504: (4)                assert not np.allclose(extent1.get_points(), extent2.get_points(),
505: (27)                                       rtol=1e-6)
506: (0)            @check_figures_equal(extensions=["png"])
507: (0)            def test_annotation_units(fig_test, fig_ref):
508: (4)                ax = fig_test.add_subplot()
509: (4)                ax.plot(datetime.now(), 1, "o")  # Implicitly set axes extents.
510: (4)                ax.annotate("x", (datetime.now(), 0.5), xycoords=("data", "axes
fraction"),
511: (16)                           xytext=(0, 0), textcoords="offset points")
512: (4)                ax = fig_ref.add_subplot()
513: (4)                ax.plot(datetime.now(), 1, "o")
514: (4)                ax.annotate("x", (datetime.now(), 0.5), xycoords=("data", "axes
fraction"))
515: (0)            @image_comparison(['large_subscript_title.png'], style='mpl20')
516: (0)            def test_large_subscript_title():
517: (4)                plt.rcParams['text.kerning_factor'] = 6
518: (4)                plt.rcParams['axes.titley'] = None
519: (4)                fig, axs = plt.subplots(1, 2, figsize=(9, 2.5), constrained_layout=True)
520: (4)                ax = axs[0]
521: (4)                ax.set_title(r'$\sum_{i} x_i$')
522: (4)                ax.set_title('New way', loc='left')
523: (4)                ax.set_xticklabels([])
524: (4)                ax = axs[1]
525: (4)                ax.set_title(r'$\sum_{i} x_i$', y=1.01)
526: (4)                ax.set_title('Old Way', loc='left')
527: (4)                ax.set_xticklabels([])
528: (0)            @pytest.mark.parametrize(
529: (4)                "x, rotation, halign",
530: (4)                [(0.7, 0, 'left'),
531: (5)                 (0.5, 95, 'left'),
532: (5)                 (0.3, 0, 'right'),
533: (5)                 (0.3, 185, 'left')])
534: (0)            def test_wrap(x, rotation, halign):
535: (4)                fig = plt.figure(figsize=(18, 18))
536: (4)                gs = GridSpec(nrows=3, ncols=3, figure=fig)
537: (4)                subfig = fig.add_subfigure(gs[1, 1])
538: (4)                s = 'This is a very long text that should be wrapped multiple times.'
539: (4)                text = subfig.text(x, 0.7, s, wrap=True, rotation=rotation, ha=halign)
540: (4)                fig.canvas.draw()
541: (4)                assert text._get_wrapped_text() == ('This is a very long\n'
```

```
(40)                                                    'text that should be\n'
(40)                                                    'wrapped multiple\n'
(40)                                                    'times.')
(0)             def test_mathwrap():
(4)                 fig = plt.figure(figsize=(6, 4))
(4)                 s = r'This is a very $\overline{\mathrm{long}}$ line of Mathtext.'
(4)                 text = fig.text(0, 0.5, s, size=40, wrap=True)
(4)                 fig.canvas.draw()
(4)                 assert text._get_wrapped_text() == ('This is a very
$\\overline{\\mathrm{long}}$\n'
(40)                                                    'line of Mathtext.')
(0)             def test_get_window_extent_wrapped():
(4)                 fig1 = plt.figure(figsize=(3, 3))
(4)                 fig1.suptitle("suptitle that is clearly too long in this case", wrap=True)
(4)                 window_extent_test = fig1._suptitle.get_window_extent()
(4)                 fig2 = plt.figure(figsize=(3, 3))
(4)                 fig2.suptitle("suptitle that is clearly\ntoo long in this case")
(4)                 window_extent_ref = fig2._suptitle.get_window_extent()
(4)                 assert window_extent_test.y0 == window_extent_ref.y0
(4)                 assert window_extent_test.y1 == window_extent_ref.y1
(0)             def test_long_word_wrap():
(4)                 fig = plt.figure(figsize=(6, 4))
(4)                 text = fig.text(9.5, 8, 'Alonglineoftexttowrap', wrap=True)
(4)                 fig.canvas.draw()
(4)                 assert text._get_wrapped_text() == 'Alonglineoftexttowrap'
(0)             def test_wrap_no_wrap():
(4)                 fig = plt.figure(figsize=(6, 4))
(4)                 text = fig.text(0, 0, 'non wrapped text', wrap=True)
(4)                 fig.canvas.draw()
(4)                 assert text._get_wrapped_text() == 'non wrapped text'
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_buffer_size(fig_test, fig_ref):
(4)                 ax = fig_test.add_subplot()
(4)                 ax.set_yticks([0, 1])
(4)                 ax.set_yticklabels(["€", "a"])
(4)                 ax.yaxis.majorTicks[1].label1.set_color("w")
(4)                 ax = fig_ref.add_subplot()
(4)                 ax.set_yticks([0, 1])
(4)                 ax.set_yticklabels(["€", ""])
(0)             def test_fontproperties_kwarg_precedence():
(4)                 """Test that kwargs take precedence over fontproperties defaults."""
(4)                 plt.figure()
(4)                 text1 = plt.xlabel("value", fontproperties='Times New Roman', size=40.0)
(4)                 text2 = plt.ylabel("counts", size=40.0, fontproperties='Times New Roman')
(4)                 assert text1.get_size() == 40.0
(4)                 assert text2.get_size() == 40.0
(0)             def test_transform_rotates_text():
(4)                 ax = plt.gca()
(4)                 transform = mtransforms.Affine2D().rotate_deg(30)
(4)                 text = ax.text(0, 0, 'test', transform=transform,
(19)                               transform_rotates_text=True)
(4)                 result = text.get_rotation()
(4)                 assert_almost_equal(result, 30)
(0)             def test_update_mutate_input():
(4)                 inp = dict(fontproperties=FontProperties(weight="bold"),
(15)                           bbox=None)
(4)                 cache = dict(inp)
(4)                 t = Text()
(4)                 t.update(inp)
(4)                 assert inp['fontproperties'] == cache['fontproperties']
(4)                 assert inp['bbox'] == cache['bbox']
(0)             @pytest.mark.parametrize('rotation', ['invalid string', [90]])
(0)             def test_invalid_rotation_values(rotation):
(4)                 with pytest.raises(
(12)                        ValueError,
(12)                        match=("rotation must be 'vertical', 'horizontal' or a number")):
(8)                     Text(0, 0, 'foo', rotation=rotation)
(0)             def test_invalid_color():
(4)                 with pytest.raises(ValueError):
```

```
(8)                     plt.figtext(.5, .5, "foo", c="foobar")
(0)             @image_comparison(['text_pdf_kerning.pdf'], style='mpl20')
(0)             def test_pdf_kerning():
(4)                 plt.figure()
(4)                 plt.figtext(0.1, 0.5, "ATATATATATATATATATA", size=30)
(0)             def test_unsupported_script(recwarn):
(4)                 fig = plt.figure()
(4)                 t = fig.text(.5, .5, "\N{BENGALI DIGIT ZERO}")
(4)                 fig.canvas.draw()
(4)                 assert all(isinstance(warn.message, UserWarning) for warn in recwarn)
(4)                 assert (
(8)                     [warn.message.args for warn in recwarn] ==
(8)                     [(r"Glyph 2534 (\N{BENGALI DIGIT ZERO}) missing from font(s) "
(12)                        + f"{t.get_fontname()}.",),
(9)                      (r"Matplotlib currently does not support Bengali natively.",)])
(0)             @pytest.mark.xfail(pyparsing_version.release == (3, 1, 0),
(19)                               reason="Error messages are incorrect with pyparsing 3.1.0")
(0)             def test_parse_math():
(4)                 fig, ax = plt.subplots()
(4)                 ax.text(0, 0, r"$ \wrong{math} $", parse_math=False)
(4)                 fig.canvas.draw()
(4)                 ax.text(0, 0, r"$ \wrong{math} $", parse_math=True)
(4)                 with pytest.raises(ValueError, match='Unknown symbol'):
(8)                     fig.canvas.draw()
(0)             @pytest.mark.xfail(pyparsing_version.release == (3, 1, 0),
(19)                               reason="Error messages are incorrect with pyparsing 3.1.0")
(0)             def test_parse_math_rcparams():
(4)                 fig, ax = plt.subplots()
(4)                 ax.text(0, 0, r"$ \wrong{math} $")
(4)                 with pytest.raises(ValueError, match='Unknown symbol'):
(8)                     fig.canvas.draw()
(4)                 with mpl.rc_context({'text.parse_math': False}):
(8)                     fig, ax = plt.subplots()
(8)                     ax.text(0, 0, r"$ \wrong{math} $")
(8)                     fig.canvas.draw()
(0)             @image_comparison(['text_pdf_font42_kerning.pdf'], style='mpl20')
(0)             def test_pdf_font42_kerning():
(4)                 plt.rcParams['pdf.fonttype'] = 42
(4)                 plt.figure()
(4)                 plt.figtext(0.1, 0.5, "ATAVATAVATAVATAVATA", size=30)
(0)             @image_comparison(['text_pdf_chars_beyond_bmp.pdf'], style='mpl20')
(0)             def test_pdf_chars_beyond_bmp():
(4)                 plt.rcParams['pdf.fonttype'] = 42
(4)                 plt.rcParams['mathtext.fontset'] = 'stixsans'
(4)                 plt.figure()
(4)                 plt.figtext(0.1, 0.5, "Mass $m$ \U00010308", size=30)
(0)             @needs_usetex
(0)             def test_metrics_cache():
(4)                 mpl.text._get_text_metrics_with_cache_impl.cache_clear()
(4)                 fig = plt.figure()
(4)                 fig.text(.3, .5, "foo\nbar")
(4)                 fig.text(.3, .5, "foo\nbar", usetex=True)
(4)                 fig.text(.5, .5, "foo\nbar", usetex=True)
(4)                 fig.canvas.draw()
(4)                 renderer = fig._get_renderer()
(4)                 ys = {}  # mapping of strings to where they were drawn in y with draw_tex.
(4)                 def call(*args, **kwargs):
(8)                     renderer, x, y, s, *_ = args
(8)                     ys.setdefault(s, set()).add(y)
(4)                 renderer.draw_tex = call
(4)                 fig.canvas.draw()
(4)                 assert [*ys] == ["foo", "bar"]
(4)                 assert len(ys["foo"]) == len(ys["bar"]) == 1
(4)                 info = mpl.text._get_text_metrics_with_cache_impl.cache_info()
(4)                 assert info.hits > info.misses
(0)             def test_annotate_offset_fontsize():
(4)                 fig, ax = plt.subplots()
(4)                 text_coords = ['offset points', 'offset fontsize']
(4)                 xy_text = [(10, 10), (1, 1)]
```

```
679: (4)                anns = [ax.annotate('test', xy=(0.5, 0.5),
680: (24)                                   xytext=xy_text[i],
681: (24)                                   fontsize='10',
682: (24)                                   xycoords='data',
683: (24)                                   textcoords=text_coords[i]) for i in range(2)]
684: (4)                points_coords, fontsize_coords = [ann.get_window_extent() for ann in anns]
685: (4)                fig.canvas.draw()
686: (4)                assert str(points_coords) == str(fontsize_coords)
687: (0)            def test_get_set_antialiased():
688: (4)                txt = Text(.5, .5, "foo\nbar")
689: (4)                assert txt._antialiased == mpl.rcParams['text.antialiased']
690: (4)                assert txt.get_antialiased() == mpl.rcParams['text.antialiased']
691: (4)                txt.set_antialiased(True)
692: (4)                assert txt._antialiased is True
693: (4)                assert txt.get_antialiased() == txt._antialiased
694: (4)                txt.set_antialiased(False)
695: (4)                assert txt._antialiased is False
696: (4)                assert txt.get_antialiased() == txt._antialiased
697: (0)            def test_annotation_antialiased():
698: (4)                annot = Annotation("foo\nbar", (.5, .5), antialiased=True)
699: (4)                assert annot._antialiased is True
700: (4)                assert annot.get_antialiased() == annot._antialiased
701: (4)                annot2 = Annotation("foo\nbar", (.5, .5), antialiased=False)
702: (4)                assert annot2._antialiased is False
703: (4)                assert annot2.get_antialiased() == annot2._antialiased
704: (4)                annot3 = Annotation("foo\nbar", (.5, .5), antialiased=False)
705: (4)                annot3.set_antialiased(True)
706: (4)                assert annot3.get_antialiased() is True
707: (4)                assert annot3._antialiased is True
708: (4)                annot4 = Annotation("foo\nbar", (.5, .5))
709: (4)                assert annot4._antialiased == mpl.rcParams['text.antialiased']
710: (0)            @check_figures_equal(extensions=["png"])
711: (0)            def test_annotate_and_offsetfrom_copy_input(fig_test, fig_ref):
712: (4)                ax = fig_test.add_subplot()
713: (4)                l, = ax.plot([0, 2], [0, 2])
714: (4)                of_xy = np.array([.5, .5])
715: (4)                ax.annotate("foo", textcoords=OffsetFrom(l, of_xy), xytext=(10, 0),
716: (16)                           xy=(0, 0))  # xy is unused.
717: (4)                of_xy[:] = 1
718: (4)                ax = fig_ref.add_subplot()
719: (4)                l, = ax.plot([0, 2], [0, 2])
720: (4)                an_xy = np.array([.5, .5])
721: (4)                ax.annotate("foo", xy=an_xy, xycoords=l, xytext=(10, 0),
textcoords="offset points")
722: (4)                an_xy[:] = 2
723: (0)            @check_figures_equal()
724: (0)            def test_text_antialiased_off_default_vs_manual(fig_test, fig_ref):
725: (4)                fig_test.text(0.5, 0.5, '6 inches x 2 inches',
726: (29)                                           antialiased=False)
727: (4)                mpl.rcParams['text.antialiased'] = False
728: (4)                fig_ref.text(0.5, 0.5, '6 inches x 2 inches')
729: (0)            @check_figures_equal()
730: (0)            def test_text_antialiased_on_default_vs_manual(fig_test, fig_ref):
731: (4)                fig_test.text(0.5, 0.5, '6 inches x 2 inches', antialiased=True)
732: (4)                mpl.rcParams['text.antialiased'] = True
733: (4)                fig_ref.text(0.5, 0.5, '6 inches x 2 inches')
734: (0)            def test_text_annotation_get_window_extent():
735: (4)                figure = Figure(dpi=100)
736: (4)                renderer = RendererAgg(200, 200, 100)
737: (4)                annotation = Annotation('test', xy=(0, 0), xycoords='figure pixels')
738: (4)                annotation.set_figure(figure)
739: (4)                text = Text(text='test', x=0, y=0)
740: (4)                text.set_figure(figure)
741: (4)                bbox = annotation.get_window_extent(renderer=renderer)
742: (4)                text_bbox = text.get_window_extent(renderer=renderer)
743: (4)                assert bbox.width == text_bbox.width
744: (4)                assert bbox.height == text_bbox.height
745: (4)                _, _, d = renderer.get_text_width_height_descent(
746: (8)                    'text', annotation._fontproperties, ismath=False)
```

```
(4)                 _, _, lp_d = renderer.get_text_width_height_descent(
(8)                     'lp', annotation._fontproperties, ismath=False)
(4)                 below_line = max(d, lp_d)
(4)                 points = bbox.get_points()
(4)                 assert points[0, 0] == 0.0
(4)                 assert points[1, 0] == text_bbox.width
(4)                 assert points[0, 1] == -below_line
(4)                 assert points[1, 1] == text_bbox.height - below_line
(0)             def test_text_with_arrow_annotation_get_window_extent():
(4)                 headwidth = 21
(4)                 fig, ax = plt.subplots(dpi=100)
(4)                 txt = ax.text(s='test', x=0, y=0)
(4)                 ann = ax.annotate(
(8)                     'test',
(8)                     xy=(0.0, 50.0),
(8)                     xytext=(50.0, 50.0), xycoords='figure pixels',
(8)                     arrowprops={
(12)                        'facecolor': 'black', 'width': 2,
(12)                        'headwidth': headwidth, 'shrink': 0.0})
(4)                 plt.draw()
(4)                 renderer = fig.canvas.renderer
(4)                 text_bbox = txt.get_window_extent(renderer=renderer)
(4)                 bbox = ann.get_window_extent(renderer=renderer)
(4)                 arrow_bbox = ann.arrow_patch.get_window_extent(renderer)
(4)                 ann_txt_bbox = Text.get_window_extent(ann)
(4)                 assert bbox.width == text_bbox.width + 50.0
(4)                 assert ann_txt_bbox.height == text_bbox.height
(4)                 assert ann_txt_bbox.width == text_bbox.width
(4)                 expected_bbox = mtransforms.Bbox.union([ann_txt_bbox, arrow_bbox])
(4)                 assert_almost_equal(bbox.height, expected_bbox.height)
(0)             def test_arrow_annotation_get_window_extent():
(4)                 dpi = 100
(4)                 dots_per_point = dpi / 72
(4)                 figure = Figure(dpi=dpi)
(4)                 figure.set_figwidth(2.0)
(4)                 figure.set_figheight(2.0)
(4)                 renderer = RendererAgg(200, 200, 100)
(4)                 annotation = Annotation(
(8)                     '', xy=(0.0, 50.0), xytext=(50.0, 50.0), xycoords='figure pixels',
(8)                     arrowprops={
(12)                        'facecolor': 'black', 'width': 8, 'headwidth': 10, 'shrink': 0.0})
(4)                 annotation.set_figure(figure)
(4)                 annotation.draw(renderer)
(4)                 bbox = annotation.get_window_extent()
(4)                 points = bbox.get_points()
(4)                 assert bbox.width == 50.0
(4)                 assert_almost_equal(bbox.height, 10.0 * dots_per_point)
(4)                 assert points[0, 0] == 0.0
(4)                 assert points[0, 1] == 50.0 - 5 * dots_per_point
(0)             def test_empty_annotation_get_window_extent():
(4)                 figure = Figure(dpi=100)
(4)                 figure.set_figwidth(2.0)
(4)                 figure.set_figheight(2.0)
(4)                 renderer = RendererAgg(200, 200, 100)
(4)                 annotation = Annotation(
(8)                     '', xy=(0.0, 50.0), xytext=(0.0, 50.0), xycoords='figure pixels')
(4)                 annotation.set_figure(figure)
(4)                 annotation.draw(renderer)
(4)                 bbox = annotation.get_window_extent()
(4)                 points = bbox.get_points()
(4)                 assert points[0, 0] == 0.0
(4)                 assert points[1, 0] == 0.0
(4)                 assert points[1, 1] == 50.0
(4)                 assert points[0, 1] == 50.0
(0)             @image_comparison(baseline_images=['basictext_wrap'],
(18)                              extensions=['png'])
(0)             def test_basic_wrap():
(4)                 fig = plt.figure()
(4)                 plt.axis([0, 10, 0, 10])
```

```
816: (4)                t = "This is a really long string that I'd rather have wrapped so that" \
817: (8)                    " it doesn't go outside of the figure, but if it's long enough it" \
818: (8)                    " will go off the top or bottom!"
819: (4)                plt.text(4, 1, t, ha='left', rotation=15, wrap=True)
820: (4)                plt.text(6, 5, t, ha='left', rotation=15, wrap=True)
821: (4)                plt.text(5, 5, t, ha='right', rotation=-15, wrap=True)
822: (4)                plt.text(5, 10, t, fontsize=18, style='oblique', ha='center',
823: (13)                         va='top', wrap=True)
824: (4)                plt.text(3, 4, t, family='serif', style='italic', ha='right', wrap=True)
825: (4)                plt.text(-1, 0, t, ha='left', rotation=-15, wrap=True)
826: (0)            @image_comparison(baseline_images=['fonttext_wrap'],
827: (18)                              extensions=['png'])
828: (0)            def test_font_wrap():
829: (4)                fig = plt.figure()
830: (4)                plt.axis([0, 10, 0, 10])
831: (4)                t = "This is a really long string that I'd rather have wrapped so that" \
832: (8)                    " it doesn't go outside of the figure, but if it's long enough it" \
833: (8)                    " will go off the top or bottom!"
834: (4)                plt.text(4, -1, t, fontsize=18, family='serif', ha='left', rotation=15,
835: (13)                         wrap=True)
836: (4)                plt.text(6, 5, t, family='sans serif', ha='left', rotation=15, wrap=True)
837: (4)                plt.text(5, 10, t, weight='heavy', ha='center', va='top', wrap=True)
838: (4)                plt.text(3, 4, t, family='monospace', ha='right', wrap=True)
839: (4)                plt.text(-1, 0, t, fontsize=14, style='italic', ha='left', rotation=-15,
840: (13)                         wrap=True)
```

File 222 - test_ticker.py:

```
1: (0)              from contextlib import nullcontext
2: (0)              import itertools
3: (0)              import locale
4: (0)              import logging
5: (0)              import re
6: (0)              from packaging.version import parse as parse_version
7: (0)              import numpy as np
8: (0)              from numpy.testing import assert_almost_equal, assert_array_equal
9: (0)              import pytest
10: (0)             import matplotlib as mpl
11: (0)             import matplotlib.pyplot as plt
12: (0)             import matplotlib.ticker as mticker
13: (0)             class TestMaxNLocator:
14: (4)                 basic_data = [
15: (8)                     (20, 100, np.array([20., 40., 60., 80., 100.])),
16: (8)                     (0.001, 0.0001, np.array([0., 0.0002, 0.0004, 0.0006, 0.0008,
0.001])),
17: (8)                     (-1e15, 1e15, np.array([-1.0e+15, -5.0e+14, 0e+00, 5e+14, 1.0e+15])),
18: (8)                     (0, 0.85e-50, np.arange(6) * 2e-51),
19: (8)                     (-0.85e-50, 0, np.arange(-5, 1) * 2e-51),
20: (4)                 ]
21: (4)                 integer_data = [
22: (8)                     (-0.1, 1.1, None, np.array([-1, 0, 1, 2])),
23: (8)                     (-0.1, 0.95, None, np.array([-0.25, 0, 0.25, 0.5, 0.75, 1.0])),
24: (8)                     (1, 55, [1, 1.5, 5, 6, 10], np.array([0, 15, 30, 45, 60])),
25: (4)                 ]
26: (4)                 @pytest.mark.parametrize('vmin, vmax, expected', basic_data)
27: (4)                 def test_basic(self, vmin, vmax, expected):
28: (8)                     loc = mticker.MaxNLocator(nbins=5)
29: (8)                     assert_almost_equal(loc.tick_values(vmin, vmax), expected)
30: (4)                 @pytest.mark.parametrize('vmin, vmax, steps, expected', integer_data)
31: (4)                 def test_integer(self, vmin, vmax, steps, expected):
32: (8)                     loc = mticker.MaxNLocator(nbins=5, integer=True, steps=steps)
33: (8)                     assert_almost_equal(loc.tick_values(vmin, vmax), expected)
34: (4)                 @pytest.mark.parametrize('kwargs, errortype, match', [
35: (8)                     ({'foo': 0}, TypeError,
36: (9)                      re.escape("set_params() got an unexpected keyword argument 'foo'")),
37: (8)                     ({'steps': [2, 1]}, ValueError, "steps argument must be an
increasing"),
```

```
 38: (8)                     ({'steps': 2}, ValueError, "steps argument must be an increasing"),
 39: (8)                     ({'steps': [2, 11]}, ValueError, "steps argument must be an
increasing"),
 40: (4)                 ])
 41: (4)                 def test_errors(self, kwargs, errortype, match):
 42: (8)                     with pytest.raises(errortype, match=match):
 43: (12)                        mticker.MaxNLocator(**kwargs)
 44: (4)                 @pytest.mark.parametrize('steps, result', [
 45: (8)                     ([1, 2, 10], [1, 2, 10]),
 46: (8)                     ([2, 10], [1, 2, 10]),
 47: (8)                     ([1, 2], [1, 2, 10]),
 48: (8)                     ([2], [1, 2, 10]),
 49: (4)                 ])
 50: (4)                 def test_padding(self, steps, result):
 51: (8)                     loc = mticker.MaxNLocator(steps=steps)
 52: (8)                     assert (loc._steps == result).all()
 53: (0)             class TestLinearLocator:
 54: (4)                 def test_basic(self):
 55: (8)                     loc = mticker.LinearLocator(numticks=3)
 56: (8)                     test_value = np.array([-0.8, -0.3, 0.2])
 57: (8)                     assert_almost_equal(loc.tick_values(-0.8, 0.2), test_value)
 58: (4)                 def test_zero_numticks(self):
 59: (8)                     loc = mticker.LinearLocator(numticks=0)
 60: (8)                     loc.tick_values(-0.8, 0.2) == []
 61: (4)                 def test_set_params(self):
 62: (8)                     """
 63: (8)                     Create linear locator with presets={}, numticks=2 and change it to
 64: (8)                     something else. See if change was successful. Should not exception.
 65: (8)                     """
 66: (8)                     loc = mticker.LinearLocator(numticks=2)
 67: (8)                     loc.set_params(numticks=8, presets={(0, 1): []})
 68: (8)                     assert loc.numticks == 8
 69: (8)                     assert loc.presets == {(0, 1): []}
 70: (4)                 def test_presets(self):
 71: (8)                     loc = mticker.LinearLocator(presets={(1, 2): [1, 1.25, 1.75],
 72: (45)                                                         (0, 2): [0.5, 1.5]})
 73: (8)                     assert loc.tick_values(1, 2) == [1, 1.25, 1.75]
 74: (8)                     assert loc.tick_values(2, 1) == [1, 1.25, 1.75]
 75: (8)                     assert loc.tick_values(0, 2) == [0.5, 1.5]
 76: (8)                     assert loc.tick_values(0.0, 2.0) == [0.5, 1.5]
 77: (8)                     assert (loc.tick_values(0, 1) == np.linspace(0, 1, 11)).all()
 78: (0)             class TestMultipleLocator:
 79: (4)                 def test_basic(self):
 80: (8)                     loc = mticker.MultipleLocator(base=3.147)
 81: (8)                     test_value = np.array([-9.441, -6.294, -3.147, 0., 3.147, 6.294,
 82: (31)                                           9.441, 12.588])
 83: (8)                     assert_almost_equal(loc.tick_values(-7, 10), test_value)
 84: (4)                 def test_basic_with_offset(self):
 85: (8)                     loc = mticker.MultipleLocator(base=3.147, offset=1.2)
 86: (8)                     test_value = np.array([-8.241, -5.094, -1.947, 1.2, 4.347, 7.494,
 87: (31)                                           10.641])
 88: (8)                     assert_almost_equal(loc.tick_values(-7, 10), test_value)
 89: (4)                 def test_view_limits(self):
 90: (8)                     """
 91: (8)                     Test basic behavior of view limits.
 92: (8)                     """
 93: (8)                     with mpl.rc_context({'axes.autolimit_mode': 'data'}):
 94: (12)                        loc = mticker.MultipleLocator(base=3.147)
 95: (12)                        assert_almost_equal(loc.view_limits(-5, 5), (-5, 5))
 96: (4)                 def test_view_limits_round_numbers(self):
 97: (8)                     """
 98: (8)                     Test that everything works properly with 'round_numbers' for auto
 99: (8)                     limit.
 100: (8)                    """
 101: (8)                    with mpl.rc_context({'axes.autolimit_mode': 'round_numbers'}):
 102: (12)                       loc = mticker.MultipleLocator(base=3.147)
 103: (12)                       assert_almost_equal(loc.view_limits(-4, 4), (-6.294, 6.294))
 104: (4)                def test_view_limits_round_numbers_with_offset(self):
 105: (8)                    """
```

```
106: (8)                    Test that everything works properly with 'round_numbers' for auto
107: (8)                    limit.
108: (8)                    """
109: (8)                    with mpl.rc_context({'axes.autolimit_mode': 'round_numbers'}):
110: (12)                       loc = mticker.MultipleLocator(base=3.147, offset=1.3)
111: (12)                       assert_almost_equal(loc.view_limits(-4, 4), (-4.994, 4.447))
112: (4)                def test_view_limits_single_bin(self):
113: (8)                    """
114: (8)                    Test that 'round_numbers' works properly with a single bin.
115: (8)                    """
116: (8)                    with mpl.rc_context({'axes.autolimit_mode': 'round_numbers'}):
117: (12)                       loc = mticker.MaxNLocator(nbins=1)
118: (12)                       assert_almost_equal(loc.view_limits(-2.3, 2.3), (-4, 4))
119: (4)                def test_set_params(self):
120: (8)                    """
121: (8)                    Create multiple locator with 0.7 base, and change it to something
else.
122: (8)                    See if change was successful.
123: (8)                    """
124: (8)                    mult = mticker.MultipleLocator(base=0.7)
125: (8)                    mult.set_params(base=1.7)
126: (8)                    assert mult._edge.step == 1.7
127: (8)                    mult.set_params(offset=3)
128: (8)                    assert mult._offset == 3
129: (0)            class TestAutoMinorLocator:
130: (4)                def test_basic(self):
131: (8)                    fig, ax = plt.subplots()
132: (8)                    ax.set_xlim(0, 1.39)
133: (8)                    ax.minorticks_on()
134: (8)                    test_value = np.array([0.05, 0.1, 0.15, 0.25, 0.3, 0.35, 0.45,
135: (31)                                          0.5, 0.55, 0.65, 0.7, 0.75, 0.85, 0.9,
136: (31)                                          0.95, 1.05, 1.1, 1.15, 1.25, 1.3, 1.35])
137: (8)                    assert_almost_equal(ax.xaxis.get_ticklocs(minor=True), test_value)
138: (4)                params = [
139: (8)                    (0, 0),  # no major tick => no minor tick either
140: (8)                    (1, 0)   # a single major tick => no minor tick
141: (4)                ]
142: (4)                def test_first_and_last_minorticks(self):
143: (8)                    """
144: (8)                    Test that first and last minor tick appear as expected.
145: (8)                    """
146: (8)                    fig, ax = plt.subplots()
147: (8)                    ax.set_xlim(-1.9, 1.9)
148: (8)                    ax.xaxis.set_minor_locator(mticker.AutoMinorLocator())
149: (8)                    test_value = np.array([-1.9, -1.8, -1.7, -1.6, -1.4, -1.3, -1.2, -1.1,
150: (31)                                          -0.9, -0.8, -0.7, -0.6, -0.4, -0.3, -0.2, -0.1,
151: (31)                                          0.1, 0.2, 0.3, 0.4, 0.6, 0.7, 0.8, 0.9, 1.1,
152: (31)                                          1.2, 1.3, 1.4, 1.6, 1.7, 1.8, 1.9])
153: (8)                    assert_almost_equal(ax.xaxis.get_ticklocs(minor=True), test_value)
154: (8)                    ax.set_xlim(-5, 5)
155: (8)                    test_value = np.array([-5.0, -4.5, -3.5, -3.0, -2.5, -1.5, -1.0, -0.5,
156: (31)                                          0.5, 1.0, 1.5, 2.5, 3.0, 3.5, 4.5, 5.0])
157: (8)                    assert_almost_equal(ax.xaxis.get_ticklocs(minor=True), test_value)
158: (4)                @pytest.mark.parametrize('nb_majorticks, expected_nb_minorticks', params)
159: (4)                def test_low_number_of_majorticks(
160: (12)                       self, nb_majorticks, expected_nb_minorticks):
161: (8)                    fig, ax = plt.subplots()
162: (8)                    xlims = (0, 5)  # easier to test the different code paths
163: (8)                    ax.set_xlim(*xlims)
164: (8)                    ax.set_xticks(np.linspace(xlims[0], xlims[1], nb_majorticks))
165: (8)                    ax.minorticks_on()
166: (8)                    ax.xaxis.set_minor_locator(mticker.AutoMinorLocator())
167: (8)                    assert len(ax.xaxis.get_minorticklocs()) == expected_nb_minorticks
168: (4)                majorstep_minordivisions = [(1, 5),
169: (32)                                           (2, 4),
170: (32)                                           (2.5, 5),
171: (32)                                           (5, 5),
172: (32)                                           (10, 5)]
173: (4)                def test_using_all_default_major_steps(self):
```

```
(8)                     with mpl.rc_context({'_internal.classic_mode': False}):
(12)                        majorsteps = [x[0] for x in self.majorstep_minordivisions]
(12)                        np.testing.assert_allclose(majorsteps,
(39)                                                   mticker.AutoLocator()._steps)
(4)                 @pytest.mark.parametrize('major_step, expected_nb_minordivisions',
(29)                                         majorstep_minordivisions)
(4)                 def test_number_of_minor_ticks(
(12)                        self, major_step, expected_nb_minordivisions):
(8)                     fig, ax = plt.subplots()
(8)                     xlims = (0, major_step)
(8)                     ax.set_xlim(*xlims)
(8)                     ax.set_xticks(xlims)
(8)                     ax.minorticks_on()
(8)                     ax.xaxis.set_minor_locator(mticker.AutoMinorLocator())
(8)                     nb_minor_divisions = len(ax.xaxis.get_minorticklocs()) + 1
(8)                     assert nb_minor_divisions == expected_nb_minordivisions
(4)                 limits = [(0, 1.39), (0, 0.139),
(14)                          (0, 0.11e-19), (0, 0.112e-12),
(14)                          (-2.0e-07, -3.3e-08), (1.20e-06, 1.42e-06),
(14)                          (-1.34e-06, -1.44e-06), (-8.76e-07, -1.51e-06)]
(4)                 reference = [
(8)                     [0.05, 0.1, 0.15, 0.25, 0.3, 0.35, 0.45, 0.5, 0.55, 0.65, 0.7,
(9)                      0.75, 0.85, 0.9, 0.95, 1.05, 1.1, 1.15, 1.25, 1.3, 1.35],
(8)                     [0.005, 0.01, 0.015, 0.025, 0.03, 0.035, 0.045, 0.05, 0.055, 0.065,
(9)                      0.07, 0.075, 0.085, 0.09, 0.095, 0.105, 0.11, 0.115, 0.125, 0.13,
(9)                      0.135],
(8)                     [5.00e-22, 1.00e-21, 1.50e-21, 2.50e-21, 3.00e-21, 3.50e-21, 4.50e-21,
(9)                      5.00e-21, 5.50e-21, 6.50e-21, 7.00e-21, 7.50e-21, 8.50e-21, 9.00e-21,
(9)                      9.50e-21, 1.05e-20, 1.10e-20],
(8)                     [5.00e-15, 1.00e-14, 1.50e-14, 2.50e-14, 3.00e-14, 3.50e-14, 4.50e-14,
(9)                      5.00e-14, 5.50e-14, 6.50e-14, 7.00e-14, 7.50e-14, 8.50e-14, 9.00e-14,
(9)                      9.50e-14, 1.05e-13, 1.10e-13],
(8)                     [-1.95e-07, -1.90e-07, -1.85e-07, -1.75e-07, -1.70e-07, -1.65e-07,
(9)                      -1.55e-07, -1.50e-07, -1.45e-07, -1.35e-07, -1.30e-07, -1.25e-07,
(9)                      -1.15e-07, -1.10e-07, -1.05e-07, -9.50e-08, -9.00e-08, -8.50e-08,
(9)                      -7.50e-08, -7.00e-08, -6.50e-08, -5.50e-08, -5.00e-08, -4.50e-08,
(9)                      -3.50e-08],
(8)                     [1.21e-06, 1.22e-06, 1.23e-06, 1.24e-06, 1.26e-06, 1.27e-06, 1.28e-06,
(9)                      1.29e-06, 1.31e-06, 1.32e-06, 1.33e-06, 1.34e-06, 1.36e-06, 1.37e-06,
(9)                      1.38e-06, 1.39e-06, 1.41e-06, 1.42e-06],
(8)                     [-1.435e-06, -1.430e-06, -1.425e-06, -1.415e-06, -1.410e-06,
(9)                      -1.405e-06, -1.395e-06, -1.390e-06, -1.385e-06, -1.375e-06,
(9)                      -1.370e-06, -1.365e-06, -1.355e-06, -1.350e-06, -1.345e-06],
(8)                     [-1.48e-06, -1.46e-06, -1.44e-06, -1.42e-06, -1.38e-06, -1.36e-06,
(9)                      -1.34e-06, -1.32e-06, -1.28e-06, -1.26e-06, -1.24e-06, -1.22e-06,
(9)                      -1.18e-06, -1.16e-06, -1.14e-06, -1.12e-06, -1.08e-06, -1.06e-06,
(9)                      -1.04e-06, -1.02e-06, -9.80e-07, -9.60e-07, -9.40e-07, -9.20e-07,
(9)                      -8.80e-07]]
(4)                 additional_data = list(zip(limits, reference))
(4)                 @pytest.mark.parametrize('lim, ref', additional_data)
(4)                 def test_additional(self, lim, ref):
(8)                     fig, ax = plt.subplots()
(8)                     ax.minorticks_on()
(8)                     ax.grid(True, 'minor', 'y', linewidth=1)
(8)                     ax.grid(True, 'major', color='k', linewidth=1)
(8)                     ax.set_ylim(lim)
(8)                     assert_almost_equal(ax.yaxis.get_ticklocs(minor=True), ref)
(4)                 @pytest.mark.parametrize('use_rcparam', [False, True])
(4)                 @pytest.mark.parametrize(
(8)                     'lim, ref', [
(12)                        ((0, 1.39),
(13)                         [0.05, 0.1, 0.15, 0.25, 0.3, 0.35, 0.45, 0.5, 0.55, 0.65, 0.7,
(14)                          0.75, 0.85, 0.9, 0.95, 1.05, 1.1, 1.15, 1.25, 1.3, 1.35]),
(12)                        ((0, 0.139),
(13)                         [0.005, 0.01, 0.015, 0.025, 0.03, 0.035, 0.045, 0.05, 0.055,
(14)                          0.065, 0.07, 0.075, 0.085, 0.09, 0.095, 0.105, 0.11, 0.115,
(14)                          0.125, 0.13, 0.135]),
(8)                     ])
(4)                 def test_number_of_minor_ticks_auto(self, lim, ref, use_rcparam):
```

```
243: (8)                    if use_rcparam:
244: (12)                       context = {'xtick.minor.ndivs': 'auto', 'ytick.minor.ndivs':
'auto'}
245: (12)                       kwargs = {}
246: (8)                    else:
247: (12)                       context = {}
248: (12)                       kwargs = {'n': 'auto'}
249: (8)                    with mpl.rc_context(context):
250: (12)                       fig, ax = plt.subplots()
251: (12)                       ax.set_xlim(*lim)
252: (12)                       ax.set_ylim(*lim)
253: (12)                       ax.xaxis.set_minor_locator(mticker.AutoMinorLocator(**kwargs))
254: (12)                       ax.yaxis.set_minor_locator(mticker.AutoMinorLocator(**kwargs))
255: (12)                       assert_almost_equal(ax.xaxis.get_ticklocs(minor=True), ref)
256: (12)                       assert_almost_equal(ax.yaxis.get_ticklocs(minor=True), ref)
257: (4)                @pytest.mark.parametrize('use_rcparam', [False, True])
258: (4)                @pytest.mark.parametrize(
259: (8)                    'n, lim, ref', [
260: (12)                       (2, (0, 4), [0.5, 1.5, 2.5, 3.5]),
261: (12)                       (4, (0, 2), [0.25, 0.5, 0.75, 1.25, 1.5, 1.75]),
262: (12)                       (10, (0, 1), [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]),
263: (8)                    ])
264: (4)                def test_number_of_minor_ticks_int(self, n, lim, ref, use_rcparam):
265: (8)                    if use_rcparam:
266: (12)                       context = {'xtick.minor.ndivs': n, 'ytick.minor.ndivs': n}
267: (12)                       kwargs = {}
268: (8)                    else:
269: (12)                       context = {}
270: (12)                       kwargs = {'n': n}
271: (8)                    with mpl.rc_context(context):
272: (12)                       fig, ax = plt.subplots()
273: (12)                       ax.set_xlim(*lim)
274: (12)                       ax.set_ylim(*lim)
275: (12)                       ax.xaxis.set_major_locator(mticker.MultipleLocator(1))
276: (12)                       ax.xaxis.set_minor_locator(mticker.AutoMinorLocator(**kwargs))
277: (12)                       ax.yaxis.set_major_locator(mticker.MultipleLocator(1))
278: (12)                       ax.yaxis.set_minor_locator(mticker.AutoMinorLocator(**kwargs))
279: (12)                       assert_almost_equal(ax.xaxis.get_ticklocs(minor=True), ref)
280: (12)                       assert_almost_equal(ax.yaxis.get_ticklocs(minor=True), ref)
281: (0)            class TestLogLocator:
282: (4)                def test_basic(self):
283: (8)                    loc = mticker.LogLocator(numticks=5)
284: (8)                    with pytest.raises(ValueError):
285: (12)                       loc.tick_values(0, 1000)
286: (8)                    test_value = np.array([1.00000000e-05, 1.00000000e-03, 1.00000000e-01,
287: (31)                                          1.00000000e+01, 1.00000000e+03, 1.00000000e+05,
288: (31)                                          1.00000000e+07, 1.000000000e+09])
289: (8)                    assert_almost_equal(loc.tick_values(0.001, 1.1e5), test_value)
290: (8)                    loc = mticker.LogLocator(base=2)
291: (8)                    test_value = np.array([0.5, 1., 2., 4., 8., 16., 32., 64., 128.,
256.])
292: (8)                    assert_almost_equal(loc.tick_values(1, 100), test_value)
293: (4)                def test_polar_axes(self):
294: (8)                    """
295: (8)                    Polar Axes have a different ticking logic.
296: (8)                    """
297: (8)                    fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
298: (8)                    ax.set_yscale('log')
299: (8)                    ax.set_ylim(1, 100)
300: (8)                    assert_array_equal(ax.get_yticks(), [10, 100, 1000])
301: (4)                def test_switch_to_autolocator(self):
302: (8)                    loc = mticker.LogLocator(subs="all")
303: (8)                    assert_array_equal(loc.tick_values(0.45, 0.55),
304: (27)                                      [0.44, 0.46, 0.48, 0.5, 0.52, 0.54, 0.56])
305: (8)                    loc = mticker.LogLocator(subs=np.arange(2, 10))
306: (8)                    assert 1.0 not in loc.tick_values(0.9, 20.)
307: (8)                    assert 10.0 not in loc.tick_values(0.9, 20.)
308: (4)                def test_set_params(self):
309: (8)                    """
```

```
310: (8)                    Create log locator with default value, base=10.0, subs=[1.0],
311: (8)                    numdecs=4, numticks=15 and change it to something else.
312: (8)                    See if change was successful. Should not raise exception.
313: (8)                    """
314: (8)                    loc = mticker.LogLocator()
315: (8)                    with pytest.warns(mpl.MatplotlibDeprecationWarning, match="numdecs"):
316: (12)                       loc.set_params(numticks=7, numdecs=8, subs=[2.0], base=4)
317: (8)                    assert loc.numticks == 7
318: (8)                    with pytest.warns(mpl.MatplotlibDeprecationWarning, match="numdecs"):
319: (12)                       assert loc.numdecs == 8
320: (8)                    assert loc._base == 4
321: (8)                    assert list(loc._subs) == [2.0]
322: (4)                def test_tick_values_correct(self):
323: (8)                    ll = mticker.LogLocator(subs=(1, 2, 5))
324: (8)                    test_value = np.array([1.e-01, 2.e-01, 5.e-01, 1.e+00, 2.e+00, 5.e+00,
325: (31)                                           1.e+01, 2.e+01, 5.e+01, 1.e+02, 2.e+02, 5.e+02,
326: (31)                                           1.e+03, 2.e+03, 5.e+03, 1.e+04, 2.e+04, 5.e+04,
327: (31)                                           1.e+05, 2.e+05, 5.e+05, 1.e+06, 2.e+06, 5.e+06,
328: (31)                                           1.e+07, 2.e+07, 5.e+07, 1.e+08, 2.e+08,
5.e+08])
329: (8)                    assert_almost_equal(ll.tick_values(1, 1e7), test_value)
330: (4)                def test_tick_values_not_empty(self):
331: (8)                    mpl.rcParams['_internal.classic_mode'] = False
332: (8)                    ll = mticker.LogLocator(subs=(1, 2, 5))
333: (8)                    test_value = np.array([1.e-01, 2.e-01, 5.e-01, 1.e+00, 2.e+00, 5.e+00,
334: (31)                                           1.e+01, 2.e+01, 5.e+01, 1.e+02, 2.e+02, 5.e+02,
335: (31)                                           1.e+03, 2.e+03, 5.e+03, 1.e+04, 2.e+04, 5.e+04,
336: (31)                                           1.e+05, 2.e+05, 5.e+05, 1.e+06, 2.e+06, 5.e+06,
337: (31)                                           1.e+07, 2.e+07, 5.e+07, 1.e+08, 2.e+08, 5.e+08,
338: (31)                                           1.e+09, 2.e+09, 5.e+09])
339: (8)                    assert_almost_equal(ll.tick_values(1, 1e8), test_value)
340: (4)                def test_multiple_shared_axes(self):
341: (8)                    rng = np.random.default_rng(19680801)
342: (8)                    dummy_data = [rng.normal(size=100), [], []]
343: (8)                    fig, axes = plt.subplots(len(dummy_data), sharex=True, sharey=True)
344: (8)                    for ax, data in zip(axes.flatten(), dummy_data):
345: (12)                       ax.hist(data, bins=10)
346: (12)                       ax.set_yscale('log', nonpositive='clip')
347: (8)                    for ax in axes.flatten():
348: (12)                       assert all(ax.get_yticks() == axes[0].get_yticks())
349: (12)                       assert ax.get_ylim() == axes[0].get_ylim()
350: (0)            class TestNullLocator:
351: (4)                def test_set_params(self):
352: (8)                    """
353: (8)                    Create null locator, and attempt to call set_params() on it.
354: (8)                    Should not exception, and should raise a warning.
355: (8)                    """
356: (8)                    loc = mticker.NullLocator()
357: (8)                    with pytest.warns(UserWarning):
358: (12)                       loc.set_params()
359: (0)            class _LogitHelper:
360: (4)                @staticmethod
361: (4)                def isclose(x, y):
362: (8)                    return (np.isclose(-np.log(1/x-1), -np.log(1/y-1))
363: (16)                           if 0 < x < 1 and 0 < y < 1 else False)
364: (4)                @staticmethod
365: (4)                def assert_almost_equal(x, y):
366: (8)                    ax = np.array(x)
367: (8)                    ay = np.array(y)
368: (8)                    assert np.all(ax > 0) and np.all(ax < 1)
369: (8)                    assert np.all(ay > 0) and np.all(ay < 1)
370: (8)                    lx = -np.log(1/ax-1)
371: (8)                    ly = -np.log(1/ay-1)
372: (8)                    assert_almost_equal(lx, ly)
373: (0)            class TestLogitLocator:
374: (4)                ref_basic_limits = [
375: (8)                    (5e-2, 1 - 5e-2),
376: (8)                    (5e-3, 1 - 5e-3),
377: (8)                    (5e-4, 1 - 5e-4),
```

```
(8)                     (5e-5, 1 - 5e-5),
(8)                     (5e-6, 1 - 5e-6),
(8)                     (5e-7, 1 - 5e-7),
(8)                     (5e-8, 1 - 5e-8),
(8)                     (5e-9, 1 - 5e-9),
(4)                 ]
(4)                 ref_basic_major_ticks = [
(8)                     1 / (10 ** np.arange(1, 3)),
(8)                     1 / (10 ** np.arange(1, 4)),
(8)                     1 / (10 ** np.arange(1, 5)),
(8)                     1 / (10 ** np.arange(1, 6)),
(8)                     1 / (10 ** np.arange(1, 7)),
(8)                     1 / (10 ** np.arange(1, 8)),
(8)                     1 / (10 ** np.arange(1, 9)),
(8)                     1 / (10 ** np.arange(1, 10)),
(4)                 ]
(4)                 ref_maxn_limits = [(0.4, 0.6), (5e-2, 2e-1), (1 - 2e-1, 1 - 5e-2)]
(4)                 @pytest.mark.parametrize(
(8)                     "lims, expected_low_ticks",
(8)                     zip(ref_basic_limits, ref_basic_major_ticks),
(4)                 )
(4)                 def test_basic_major(self, lims, expected_low_ticks):
(8)                     """
(8)                     Create logit locator with huge number of major, and tests ticks.
(8)                     """
(8)                     expected_ticks = sorted(
(12)                        [*expected_low_ticks, 0.5, *(1 - expected_low_ticks)]
(8)                     )
(8)                     loc = mticker.LogitLocator(nbins=100)
(8)                     _LogitHelper.assert_almost_equal(
(12)                        loc.tick_values(*lims),
(12)                        expected_ticks
(8)                     )
(4)                 @pytest.mark.parametrize("lims", ref_maxn_limits)
(4)                 def test_maxn_major(self, lims):
(8)                     """
(8)                     When the axis is zoomed, the locator must have the same behavior as
(8)                     MaxNLocator.
(8)                     """
(8)                     loc = mticker.LogitLocator(nbins=100)
(8)                     maxn_loc = mticker.MaxNLocator(nbins=100, steps=[1, 2, 5, 10])
(8)                     for nbins in (4, 8, 16):
(12)                        loc.set_params(nbins=nbins)
(12)                        maxn_loc.set_params(nbins=nbins)
(12)                        ticks = loc.tick_values(*lims)
(12)                        maxn_ticks = maxn_loc.tick_values(*lims)
(12)                        assert ticks.shape == maxn_ticks.shape
(12)                        assert (ticks == maxn_ticks).all()
(4)                 @pytest.mark.parametrize("lims", ref_basic_limits + ref_maxn_limits)
(4)                 def test_nbins_major(self, lims):
(8)                     """
(8)                     Assert logit locator for respecting nbins param.
(8)                     """
(8)                     basic_needed = int(-np.floor(np.log10(lims[0]))) * 2 + 1
(8)                     loc = mticker.LogitLocator(nbins=100)
(8)                     for nbins in range(basic_needed, 2, -1):
(12)                        loc.set_params(nbins=nbins)
(12)                        assert len(loc.tick_values(*lims)) <= nbins + 2
(4)                 @pytest.mark.parametrize(
(8)                     "lims, expected_low_ticks",
(8)                     zip(ref_basic_limits, ref_basic_major_ticks),
(4)                 )
(4)                 def test_minor(self, lims, expected_low_ticks):
(8)                     """
(8)                     In large scale, test the presence of minor,
(8)                     and assert no minor when major are subsampled.
(8)                     """
(8)                     expected_ticks = sorted(
(12)                        [*expected_low_ticks, 0.5, *(1 - expected_low_ticks)]
```

```
447: (8)                    )
448: (8)                    basic_needed = len(expected_ticks)
449: (8)                    loc = mticker.LogitLocator(nbins=100)
450: (8)                    minor_loc = mticker.LogitLocator(nbins=100, minor=True)
451: (8)                    for nbins in range(basic_needed, 2, -1):
452: (12)                       loc.set_params(nbins=nbins)
453: (12)                       minor_loc.set_params(nbins=nbins)
454: (12)                       major_ticks = loc.tick_values(*lims)
455: (12)                       minor_ticks = minor_loc.tick_values(*lims)
456: (12)                       if len(major_ticks) >= len(expected_ticks):
457: (16)                           assert (len(major_ticks) - 1) * 5 < len(minor_ticks)
458: (12)                       else:
459: (16)                           _LogitHelper.assert_almost_equal(
460: (20)                               sorted([*major_ticks, *minor_ticks]), expected_ticks)
461: (4)                def test_minor_attr(self):
462: (8)                    loc = mticker.LogitLocator(nbins=100)
463: (8)                    assert not loc.minor
464: (8)                    loc.minor = True
465: (8)                    assert loc.minor
466: (8)                    loc.set_params(minor=False)
467: (8)                    assert not loc.minor
468: (4)                acceptable_vmin_vmax = [
469: (8)                    *(2.5 ** np.arange(-3, 0)),
470: (8)                    *(1 - 2.5 ** np.arange(-3, 0)),
471: (4)                ]
472: (4)                @pytest.mark.parametrize(
473: (8)                    "lims",
474: (8)                    [
475: (12)                       (a, b)
476: (12)                       for (a, b) in itertools.product(acceptable_vmin_vmax, repeat=2)
477: (12)                       if a != b
478: (8)                    ],
479: (4)                )
480: (4)                def test_nonsingular_ok(self, lims):
481: (8)                    """
482: (8)                    Create logit locator, and test the nonsingular method for acceptable
483: (8)                    value
484: (8)                    """
485: (8)                    loc = mticker.LogitLocator()
486: (8)                    lims2 = loc.nonsingular(*lims)
487: (8)                    assert sorted(lims) == sorted(lims2)
488: (4)                @pytest.mark.parametrize("okval", acceptable_vmin_vmax)
489: (4)                def test_nonsingular_nok(self, okval):
490: (8)                    """
491: (8)                    Create logit locator, and test the nonsingular method for non
492: (8)                    acceptable value
493: (8)                    """
494: (8)                    loc = mticker.LogitLocator()
495: (8)                    vmin, vmax = (-1, okval)
496: (8)                    vmin2, vmax2 = loc.nonsingular(vmin, vmax)
497: (8)                    assert vmax2 == vmax
498: (8)                    assert 0 < vmin2 < vmax2
499: (8)                    vmin, vmax = (okval, 2)
500: (8)                    vmin2, vmax2 = loc.nonsingular(vmin, vmax)
501: (8)                    assert vmin2 == vmin
502: (8)                    assert vmin2 < vmax2 < 1
503: (0)            class TestFixedLocator:
504: (4)                def test_set_params(self):
505: (8)                    """
506: (8)                    Create fixed locator with 5 nbins, and change it to something else.
507: (8)                    See if change was successful.
508: (8)                    Should not exception.
509: (8)                    """
510: (8)                    fixed = mticker.FixedLocator(range(0, 24), nbins=5)
511: (8)                    fixed.set_params(nbins=7)
512: (8)                    assert fixed.nbins == 7
513: (0)            class TestIndexLocator:
514: (4)                def test_set_params(self):
515: (8)                    """
```

```
516: (8)                    Create index locator with 3 base, 4 offset. and change it to something
517: (8)                    else. See if change was successful.
518: (8)                    Should not exception.
519: (8)                    """
520: (8)                    index = mticker.IndexLocator(base=3, offset=4)
521: (8)                    index.set_params(base=7, offset=7)
522: (8)                    assert index._base == 7
523: (8)                    assert index.offset == 7
524: (0)            class TestSymmetricalLogLocator:
525: (4)                def test_set_params(self):
526: (8)                    """
527: (8)                    Create symmetrical log locator with default subs =[1.0] numticks = 15,
528: (8)                    and change it to something else.
529: (8)                    See if change was successful.
530: (8)                    Should not exception.
531: (8)                    """
532: (8)                    sym = mticker.SymmetricalLogLocator(base=10, linthresh=1)
533: (8)                    sym.set_params(subs=[2.0], numticks=8)
534: (8)                    assert sym._subs == [2.0]
535: (8)                    assert sym.numticks == 8
536: (4)                @pytest.mark.parametrize(
537: (12)                       'vmin, vmax, expected',
538: (12)                       [
539: (16)                           (0, 1, [0, 1]),
540: (16)                           (-1, 1, [-1, 0, 1]),
541: (12)                       ],
542: (4)                )
543: (4)                def test_values(self, vmin, vmax, expected):
544: (8)                    sym = mticker.SymmetricalLogLocator(base=10, linthresh=1)
545: (8)                    ticks = sym.tick_values(vmin=vmin, vmax=vmax)
546: (8)                    assert_array_equal(ticks, expected)
547: (4)                def test_subs(self):
548: (8)                    sym = mticker.SymmetricalLogLocator(base=10, linthresh=1, subs=[2.0,
4.0])
549: (8)                    sym.create_dummy_axis()
550: (8)                    sym.axis.set_view_interval(-10, 10)
551: (8)                    assert (sym() == [-20., -40.,  -2.,  -4.,   0.,   2.,   4.,  20.,
40.]).all()
552: (4)                def test_extending(self):
553: (8)                    sym = mticker.SymmetricalLogLocator(base=10, linthresh=1)
554: (8)                    sym.create_dummy_axis()
555: (8)                    sym.axis.set_view_interval(8, 9)
556: (8)                    assert (sym() == [1.0]).all()
557: (8)                    sym.axis.set_view_interval(8, 12)
558: (8)                    assert (sym() == [1.0, 10.0]).all()
559: (8)                    assert sym.view_limits(10, 10) == (1, 100)
560: (8)                    assert sym.view_limits(-10, -10) == (-100, -1)
561: (8)                    assert sym.view_limits(0, 0) == (-0.001, 0.001)
562: (0)            class TestAsinhLocator:
563: (4)                def test_init(self):
564: (8)                    lctr = mticker.AsinhLocator(linear_width=2.718, numticks=19)
565: (8)                    assert lctr.linear_width == 2.718
566: (8)                    assert lctr.numticks == 19
567: (8)                    assert lctr.base == 10
568: (4)                def test_set_params(self):
569: (8)                    lctr = mticker.AsinhLocator(linear_width=5,
570: (36)                                               numticks=17, symthresh=0.125,
571: (36)                                               base=4, subs=(2.5, 3.25))
572: (8)                    assert lctr.numticks == 17
573: (8)                    assert lctr.symthresh == 0.125
574: (8)                    assert lctr.base == 4
575: (8)                    assert lctr.subs == (2.5, 3.25)
576: (8)                    lctr.set_params(numticks=23)
577: (8)                    assert lctr.numticks == 23
578: (8)                    lctr.set_params(None)
579: (8)                    assert lctr.numticks == 23
580: (8)                    lctr.set_params(symthresh=0.5)
581: (8)                    assert lctr.symthresh == 0.5
582: (8)                    lctr.set_params(symthresh=None)
```

```
(8)                     assert lctr.symthresh == 0.5
(8)                     lctr.set_params(base=7)
(8)                     assert lctr.base == 7
(8)                     lctr.set_params(base=None)
(8)                     assert lctr.base == 7
(8)                     lctr.set_params(subs=(2, 4.125))
(8)                     assert lctr.subs == (2, 4.125)
(8)                     lctr.set_params(subs=None)
(8)                     assert lctr.subs == (2, 4.125)
(8)                     lctr.set_params(subs=[])
(8)                     assert lctr.subs is None
(4)                 def test_linear_values(self):
(8)                     lctr = mticker.AsinhLocator(linear_width=100, numticks=11, base=0)
(8)                     assert_almost_equal(lctr.tick_values(-1, 1),
(28)                                         np.arange(-1, 1.01, 0.2))
(8)                     assert_almost_equal(lctr.tick_values(-0.1, 0.1),
(28)                                         np.arange(-0.1, 0.101, 0.02))
(8)                     assert_almost_equal(lctr.tick_values(-0.01, 0.01),
(28)                                         np.arange(-0.01, 0.0101, 0.002))
(4)                 def test_wide_values(self):
(8)                     lctr = mticker.AsinhLocator(linear_width=0.1, numticks=11, base=0)
(8)                     assert_almost_equal(lctr.tick_values(-100, 100),
(28)                                         [-100, -20, -5, -1, -0.2,
(29)                                          0, 0.2, 1, 5, 20, 100])
(8)                     assert_almost_equal(lctr.tick_values(-1000, 1000),
(28)                                         [-1000, -100, -20, -3, -0.4,
(29)                                          0, 0.4, 3, 20, 100, 1000])
(4)                 def test_near_zero(self):
(8)                     """Check that manually injected zero will supersede nearby tick"""
(8)                     lctr = mticker.AsinhLocator(linear_width=100, numticks=3, base=0)
(8)                     assert_almost_equal(lctr.tick_values(-1.1, 0.9), [-1.0, 0.0, 0.9])
(4)                 def test_fallback(self):
(8)                     lctr = mticker.AsinhLocator(1.0, numticks=11)
(8)                     assert_almost_equal(lctr.tick_values(101, 102),
(28)                                         np.arange(101, 102.01, 0.1))
(4)                 def test_symmetrizing(self):
(8)                     lctr = mticker.AsinhLocator(linear_width=1, numticks=3,
(36)                                                 symthresh=0.25, base=0)
(8)                     lctr.create_dummy_axis()
(8)                     lctr.axis.set_view_interval(-1, 2)
(8)                     assert_almost_equal(lctr(), [-1, 0, 2])
(8)                     lctr.axis.set_view_interval(-1, 0.9)
(8)                     assert_almost_equal(lctr(), [-1, 0, 1])
(8)                     lctr.axis.set_view_interval(-0.85, 1.05)
(8)                     assert_almost_equal(lctr(), [-1, 0, 1])
(8)                     lctr.axis.set_view_interval(1, 1.1)
(8)                     assert_almost_equal(lctr(), [1, 1.05, 1.1])
(4)                 def test_base_rounding(self):
(8)                     lctr10 = mticker.AsinhLocator(linear_width=1, numticks=8,
(38)                                                   base=10, subs=(1, 3, 5))
(8)                     assert_almost_equal(lctr10.tick_values(-110, 110),
(28)                                         [-500, -300, -100, -50, -30, -10, -5, -3, -1,
(29)                                          -0.5, -0.3, -0.1, 0, 0.1, 0.3, 0.5,
(29)                                          1, 3, 5, 10, 30, 50, 100, 300, 500])
(8)                     lctr5 = mticker.AsinhLocator(linear_width=1, numticks=20, base=5)
(8)                     assert_almost_equal(lctr5.tick_values(-1050, 1050),
(28)                                         [-625, -125, -25, -5, -1, -0.2, 0,
(29)                                          0.2, 1, 5, 25, 125, 625])
(0)             class TestScalarFormatter:
(4)                 offset_data = [
(8)                     (123, 189, 0),
(8)                     (-189, -123, 0),
(8)                     (12341, 12349, 12340),
(8)                     (-12349, -12341, -12340),
(8)                     (99999.5, 100010.5, 100000),
(8)                     (-100010.5, -99999.5, -100000),
(8)                     (99990.5, 100000.5, 100000),
(8)                     (-100000.5, -99990.5, -100000),
(8)                     (1233999, 1234001, 1234000),
```

```
652: (8)                    (-1234001, -1233999, -1234000),
653: (8)                    (1, 1, 1),
654: (8)                    (123, 123, 0),
655: (8)                    (.4538, .4578, .45),
656: (8)                    (3789.12, 3783.1, 3780),
657: (8)                    (45124.3, 45831.75, 45000),
658: (8)                    (0.000721, 0.0007243, 0.00072),
659: (8)                    (12592.82, 12591.43, 12590),
660: (8)                    (9., 12., 0),
661: (8)                    (900., 1200., 0),
662: (8)                    (1900., 1200., 0),
663: (8)                    (0.99, 1.01, 1),
664: (8)                    (9.99, 10.01, 10),
665: (8)                    (99.99, 100.01, 100),
666: (8)                    (5.99, 6.01, 6),
667: (8)                    (15.99, 16.01, 16),
668: (8)                    (-0.452, 0.492, 0),
669: (8)                    (-0.492, 0.492, 0),
670: (8)                    (12331.4, 12350.5, 12300),
671: (8)                    (-12335.3, 12335.3, 0),
672: (4)                ]
673: (4)                use_offset_data = [True, False]
674: (4)                useMathText_data = [True, False]
675: (4)                scilimits_data = [
676: (8)                    (False, (0, 0), (10.0, 20.0), 0, False),
677: (8)                    (True, (-2, 2), (-10, 20), 0, False),
678: (8)                    (True, (-2, 2), (-20, 10), 0, False),
679: (8)                    (True, (-2, 2), (-110, 120), 2, False),
680: (8)                    (True, (-2, 2), (-120, 110), 2, False),
681: (8)                    (True, (-2, 2), (-.001, 0.002), -3, False),
682: (8)                    (True, (-7, 7), (0.18e10, 0.83e10), 9, True),
683: (8)                    (True, (0, 0), (-1e5, 1e5), 5, False),
684: (8)                    (True, (6, 6), (-1e5, 1e5), 6, False),
685: (4)                ]
686: (4)                cursor_data = [
687: (8)                    [0., "0.000"],
688: (8)                    [0.0123, "0.012"],
689: (8)                    [0.123, "0.123"],
690: (8)                    [1.23,  "1.230"],
691: (8)                    [12.3, "12.300"],
692: (4)                ]
693: (4)                format_data = [
694: (8)                    (.1, "1e-1"),
695: (8)                    (.11, "1.1e-1"),
696: (8)                    (1e8, "1e8"),
697: (8)                    (1.1e8, "1.1e8"),
698: (4)                ]
699: (4)                @pytest.mark.parametrize('unicode_minus, result',
700: (29)                                           [(True, "\N{MINUS SIGN}1"), (False, "-1")])
701: (4)                def test_unicode_minus(self, unicode_minus, result):
702: (8)                    mpl.rcParams['axes.unicode_minus'] = unicode_minus
703: (8)                    assert (
704: (12)
plt.gca().xaxis.get_major_formatter().format_data_short(-1).strip()
705: (12)                       == result)
706: (4)                @pytest.mark.parametrize('left, right, offset', offset_data)
707: (4)                def test_offset_value(self, left, right, offset):
708: (8)                    fig, ax = plt.subplots()
709: (8)                    formatter = ax.xaxis.get_major_formatter()
710: (8)                    with (pytest.warns(UserWarning, match='Attempting to set identical')
711: (14)                         if left == right else nullcontext()):
712: (12)                       ax.set_xlim(left, right)
713: (8)                    ax.xaxis._update_ticks()
714: (8)                    assert formatter.offset == offset
715: (8)                    with (pytest.warns(UserWarning, match='Attempting to set identical')
716: (14)                         if left == right else nullcontext()):
717: (12)                       ax.set_xlim(right, left)
718: (8)                    ax.xaxis._update_ticks()
719: (8)                    assert formatter.offset == offset
```

```
    @pytest.mark.parametrize('use_offset', use_offset_data)
    def test_use_offset(self, use_offset):
        with mpl.rc_context({'axes.formatter.useoffset': use_offset}):
            tmp_form = mticker.ScalarFormatter()
            assert use_offset == tmp_form.get_useOffset()
            assert tmp_form.offset == 0
    @pytest.mark.parametrize('use_math_text', useMathText_data)
    def test_useMathText(self, use_math_text):
        with mpl.rc_context({'axes.formatter.use_mathtext': use_math_text}):
            tmp_form = mticker.ScalarFormatter()
            assert use_math_text == tmp_form.get_useMathText()
    def test_set_use_offset_float(self):
        tmp_form = mticker.ScalarFormatter()
        tmp_form.set_useOffset(0.5)
        assert not tmp_form.get_useOffset()
        assert tmp_form.offset == 0.5
    def test_use_locale(self):
        conv = locale.localeconv()
        sep = conv['thousands_sep']
        if not sep or conv['grouping'][-1:] in ([], [locale.CHAR_MAX]):
            pytest.skip('Locale does not apply grouping')  # pragma: no cover
        with mpl.rc_context({'axes.formatter.use_locale': True}):
            tmp_form = mticker.ScalarFormatter()
            assert tmp_form.get_useLocale()
            tmp_form.create_dummy_axis()
            tmp_form.axis.set_data_interval(0, 10)
            tmp_form.set_locs([1, 2, 3])
            assert sep in tmp_form(1e9)
    @pytest.mark.parametrize(
        'sci_type, scilimits, lim, orderOfMag, fewticks', scilimits_data)
    def test_scilimits(self, sci_type, scilimits, lim, orderOfMag, fewticks):
        tmp_form = mticker.ScalarFormatter()
        tmp_form.set_scientific(sci_type)
        tmp_form.set_powerlimits(scilimits)
        fig, ax = plt.subplots()
        ax.yaxis.set_major_formatter(tmp_form)
        ax.set_ylim(*lim)
        if fewticks:
            ax.yaxis.set_major_locator(mticker.MaxNLocator(4))
        tmp_form.set_locs(ax.yaxis.get_majorticklocs())
        assert orderOfMag == tmp_form.orderOfMagnitude
    @pytest.mark.parametrize('value, expected', format_data)
    def test_format_data(self, value, expected):
        mpl.rcParams['axes.unicode_minus'] = False
        sf = mticker.ScalarFormatter()
        assert sf.format_data(value) == expected
    @pytest.mark.parametrize('data, expected', cursor_data)
    def test_cursor_precision(self, data, expected):
        fig, ax = plt.subplots()
        ax.set_xlim(-1, 1)  # Pointing precision of 0.001.
        fmt = ax.xaxis.get_major_formatter().format_data_short
        assert fmt(data) == expected
    @pytest.mark.parametrize('data, expected', cursor_data)
    def test_cursor_dummy_axis(self, data, expected):
        sf = mticker.ScalarFormatter()
        sf.create_dummy_axis()
        sf.axis.set_view_interval(0, 10)
        fmt = sf.format_data_short
        assert fmt(data) == expected
        assert sf.axis.get_tick_space() == 9
        assert sf.axis.get_minpos() == 0
    def test_mathtext_ticks(self):
        mpl.rcParams.update({
            'font.family': 'serif',
            'font.serif': 'cmr10',
            'axes.formatter.use_mathtext': False
        })
        if parse_version(pytest.__version__).major < 8:
            with pytest.warns(UserWarning, match='cmr10 font should ideally'):
```

```
789: (16)                            fig, ax = plt.subplots()
790: (16)                            ax.set_xticks([-1, 0, 1])
791: (16)                            fig.canvas.draw()
792: (8)                    else:
793: (12)                       with (pytest.warns(UserWarning, match="Glyph 8722"),
794: (18)                             pytest.warns(UserWarning, match='cmr10 font should
ideally')):
795: (16)                            fig, ax = plt.subplots()
796: (16)                            ax.set_xticks([-1, 0, 1])
797: (16)                            fig.canvas.draw()
798: (4)                def test_cmr10_substitutions(self, caplog):
799: (8)                    mpl.rcParams.update({
800: (12)                       'font.family': 'cmr10',
801: (12)                       'mathtext.fontset': 'cm',
802: (12)                       'axes.formatter.use_mathtext': True,
803: (8)                    })
804: (8)                    with caplog.at_level(logging.WARNING, logger='matplotlib.mathtext'):
805: (12)                       fig, ax = plt.subplots()
806: (12)                       ax.plot([-0.03, 0.05], [40, 0.05])
807: (12)                       ax.set_yscale('log')
808: (12)                       yticks = [0.02, 0.3, 4, 50]
809: (12)                       formatter = mticker.LogFormatterSciNotation()
810: (12)                       ax.set_yticks(yticks, map(formatter, yticks))
811: (12)                       fig.canvas.draw()
812: (12)                       assert not caplog.text
813: (4)                def test_empty_locs(self):
814: (8)                    sf = mticker.ScalarFormatter()
815: (8)                    sf.set_locs([])
816: (8)                    assert sf(0.5) == ''
817: (0)            class TestLogFormatterExponent:
818: (4)                param_data = [
819: (8)                    (True, 4, np.arange(-3, 4.0), np.arange(-3, 4.0),
820: (9)                     ['-3', '-2', '-1', '0', '1', '2', '3']),
821: (8)                    (False, 10, np.array([0.1, 0.00001, np.pi, 0.2, -0.2, -0.00001]),
822: (9)                     range(6), ['0.1', '1e-05', '3.14', '0.2', '-0.2', '-1e-05']),
823: (8)                    (False, 50, np.array([3, 5, 12, 42], dtype=float), range(6),
824: (9)                     ['3', '5', '12', '42']),
825: (4)                ]
826: (4)                base_data = [2.0, 5.0, 10.0, np.pi, np.e]
827: (4)                @pytest.mark.parametrize(
828: (12)                       'labelOnlyBase, exponent, locs, positions, expected', param_data)
829: (4)                @pytest.mark.parametrize('base', base_data)
830: (4)                def test_basic(self, labelOnlyBase, base, exponent, locs, positions,
831: (19)                              expected):
832: (8)                    formatter = mticker.LogFormatterExponent(base=base,
833: (49)                                                             labelOnlyBase=labelOnlyBase)
834: (8)                    formatter.create_dummy_axis()
835: (8)                    formatter.axis.set_view_interval(1, base**exponent)
836: (8)                    vals = base**locs
837: (8)                    labels = [formatter(x, pos) for (x, pos) in zip(vals, positions)]
838: (8)                    expected = [label.replace('-', '\N{Minus Sign}') for label in
expected]
839: (8)                    assert labels == expected
840: (4)                def test_blank(self):
841: (8)                    formatter = mticker.LogFormatterExponent(base=10, labelOnlyBase=True)
842: (8)                    formatter.create_dummy_axis()
843: (8)                    formatter.axis.set_view_interval(1, 10)
844: (8)                    assert formatter(10**0.1) == ''
845: (0)            class TestLogFormatterMathtext:
846: (4)                fmt = mticker.LogFormatterMathtext()
847: (4)                test_data = [
848: (8)                    (0, 1, '$\\mathdefault{10^{0}}$'),
849: (8)                    (0, 1e-2, '$\\mathdefault{10^{-2}}$'),
850: (8)                    (0, 1e2, '$\\mathdefault{10^{2}}$'),
851: (8)                    (3, 1, '$\\mathdefault{1}$'),
852: (8)                    (3, 1e-2, '$\\mathdefault{0.01}$'),
853: (8)                    (3, 1e2, '$\\mathdefault{100}$'),
854: (8)                    (3, 1e-3, '$\\mathdefault{10^{-3}}$'),
855: (8)                    (3, 1e3, '$\\mathdefault{10^{3}}$'),
```

```
(4)                 ]
(4)                 @pytest.mark.parametrize('min_exponent, value, expected', test_data)
(4)                 def test_min_exponent(self, min_exponent, value, expected):
(8)                     with mpl.rc_context({'axes.formatter.min_exponent': min_exponent}):
(12)                        assert self.fmt(value) == expected
(0)             class TestLogFormatterSciNotation:
(4)                 test_data = [
(8)                     (2, 0.03125, '$\\mathdefault{2^{-5}}$'),
(8)                     (2, 1, '$\\mathdefault{2^{0}}$'),
(8)                     (2, 32, '$\\mathdefault{2^{5}}$'),
(8)                     (2, 0.0375, '$\\mathdefault{1.2\\times2^{-5}}$'),
(8)                     (2, 1.2, '$\\mathdefault{1.2\\times2^{0}}$'),
(8)                     (2, 38.4, '$\\mathdefault{1.2\\times2^{5}}$'),
(8)                     (10, -1, '$\\mathdefault{-10^{0}}$'),
(8)                     (10, 1e-05, '$\\mathdefault{10^{-5}}$'),
(8)                     (10, 1, '$\\mathdefault{10^{0}}$'),
(8)                     (10, 100000, '$\\mathdefault{10^{5}}$'),
(8)                     (10, 2e-05, '$\\mathdefault{2\\times10^{-5}}$'),
(8)                     (10, 2, '$\\mathdefault{2\\times10^{0}}$'),
(8)                     (10, 200000, '$\\mathdefault{2\\times10^{5}}$'),
(8)                     (10, 5e-05, '$\\mathdefault{5\\times10^{-5}}$'),
(8)                     (10, 5, '$\\mathdefault{5\\times10^{0}}$'),
(8)                     (10, 500000, '$\\mathdefault{5\\times10^{5}}$'),
(4)                 ]
(4)                 @mpl.style.context('default')
(4)                 @pytest.mark.parametrize('base, value, expected', test_data)
(4)                 def test_basic(self, base, value, expected):
(8)                     formatter = mticker.LogFormatterSciNotation(base=base)
(8)                     with mpl.rc_context({'text.usetex': False}):
(12)                        assert formatter(value) == expected
(0)             class TestLogFormatter:
(4)                 pprint_data = [
(8)                     (3.141592654e-05, 0.001, '3.142e-5'),
(8)                     (0.0003141592654, 0.001, '3.142e-4'),
(8)                     (0.003141592654, 0.001, '3.142e-3'),
(8)                     (0.03141592654, 0.001, '3.142e-2'),
(8)                     (0.3141592654, 0.001, '3.142e-1'),
(8)                     (3.141592654, 0.001, '3.142'),
(8)                     (31.41592654, 0.001, '3.142e1'),
(8)                     (314.1592654, 0.001, '3.142e2'),
(8)                     (3141.592654, 0.001, '3.142e3'),
(8)                     (31415.92654, 0.001, '3.142e4'),
(8)                     (314159.2654, 0.001, '3.142e5'),
(8)                     (1e-05, 0.001, '1e-5'),
(8)                     (0.0001, 0.001, '1e-4'),
(8)                     (0.001, 0.001, '1e-3'),
(8)                     (0.01, 0.001, '1e-2'),
(8)                     (0.1, 0.001, '1e-1'),
(8)                     (1, 0.001, '1'),
(8)                     (10, 0.001, '10'),
(8)                     (100, 0.001, '100'),
(8)                     (1000, 0.001, '1000'),
(8)                     (10000, 0.001, '1e4'),
(8)                     (100000, 0.001, '1e5'),
(8)                     (3.141592654e-05, 0.015, '0'),
(8)                     (0.0003141592654, 0.015, '0'),
(8)                     (0.003141592654, 0.015, '0.003'),
(8)                     (0.03141592654, 0.015, '0.031'),
(8)                     (0.3141592654, 0.015, '0.314'),
(8)                     (3.141592654, 0.015, '3.142'),
(8)                     (31.41592654, 0.015, '31.416'),
(8)                     (314.1592654, 0.015, '314.159'),
(8)                     (3141.592654, 0.015, '3141.593'),
(8)                     (31415.92654, 0.015, '31415.927'),
(8)                     (314159.2654, 0.015, '314159.265'),
(8)                     (1e-05, 0.015, '0'),
(8)                     (0.0001, 0.015, '0'),
(8)                     (0.001, 0.015, '0.001'),
(8)                     (0.01, 0.015, '0.01'),
```

```
(8)                     (0.1, 0.015, '0.1'),
(8)                     (1, 0.015, '1'),
(8)                     (10, 0.015, '10'),
(8)                     (100, 0.015, '100'),
(8)                     (1000, 0.015, '1000'),
(8)                     (10000, 0.015, '10000'),
(8)                     (100000, 0.015, '100000'),
(8)                     (3.141592654e-05, 0.5, '0'),
(8)                     (0.0003141592654, 0.5, '0'),
(8)                     (0.003141592654, 0.5, '0.003'),
(8)                     (0.03141592654, 0.5, '0.031'),
(8)                     (0.3141592654, 0.5, '0.314'),
(8)                     (3.141592654, 0.5, '3.142'),
(8)                     (31.41592654, 0.5, '31.416'),
(8)                     (314.1592654, 0.5, '314.159'),
(8)                     (3141.592654, 0.5, '3141.593'),
(8)                     (31415.92654, 0.5, '31415.927'),
(8)                     (314159.2654, 0.5, '314159.265'),
(8)                     (1e-05, 0.5, '0'),
(8)                     (0.0001, 0.5, '0'),
(8)                     (0.001, 0.5, '0.001'),
(8)                     (0.01, 0.5, '0.01'),
(8)                     (0.1, 0.5, '0.1'),
(8)                     (1, 0.5, '1'),
(8)                     (10, 0.5, '10'),
(8)                     (100, 0.5, '100'),
(8)                     (1000, 0.5, '1000'),
(8)                     (10000, 0.5, '10000'),
(8)                     (100000, 0.5, '100000'),
(8)                     (3.141592654e-05, 5, '0'),
(8)                     (0.0003141592654, 5, '0'),
(8)                     (0.003141592654, 5, '0'),
(8)                     (0.03141592654, 5, '0.03'),
(8)                     (0.3141592654, 5, '0.31'),
(8)                     (3.141592654, 5, '3.14'),
(8)                     (31.41592654, 5, '31.42'),
(8)                     (314.1592654, 5, '314.16'),
(8)                     (3141.592654, 5, '3141.59'),
(8)                     (31415.92654, 5, '31415.93'),
(8)                     (314159.2654, 5, '314159.27'),
(8)                     (1e-05, 5, '0'),
(8)                     (0.0001, 5, '0'),
(8)                     (0.001, 5, '0'),
(8)                     (0.01, 5, '0.01'),
(8)                     (0.1, 5, '0.1'),
(8)                     (1, 5, '1'),
(8)                     (10, 5, '10'),
(8)                     (100, 5, '100'),
(8)                     (1000, 5, '1000'),
(8)                     (10000, 5, '10000'),
(8)                     (100000, 5, '100000'),
(8)                     (3.141592654e-05, 100, '0'),
(8)                     (0.0003141592654, 100, '0'),
(8)                     (0.003141592654, 100, '0'),
(8)                     (0.03141592654, 100, '0'),
(8)                     (0.3141592654, 100, '0.3'),
(8)                     (3.141592654, 100, '3.1'),
(8)                     (31.41592654, 100, '31.4'),
(8)                     (314.1592654, 100, '314.2'),
(8)                     (3141.592654, 100, '3141.6'),
(8)                     (31415.92654, 100, '31415.9'),
(8)                     (314159.2654, 100, '314159.3'),
(8)                     (1e-05, 100, '0'),
(8)                     (0.0001, 100, '0'),
(8)                     (0.001, 100, '0'),
(8)                     (0.01, 100, '0'),
(8)                     (0.1, 100, '0.1'),
(8)                     (1, 100, '1'),
(8)                     (10, 100, '10'),
```

```
994: (8)                    (100, 100, '100'),
995: (8)                    (1000, 100, '1000'),
996: (8)                    (10000, 100, '10000'),
997: (8)                    (100000, 100, '100000'),
998: (8)                    (3.141592654e-05, 1000000.0, '3.1e-5'),
999: (8)                    (0.0003141592654, 1000000.0, '3.1e-4'),
1000: (8)                   (0.003141592654, 1000000.0, '3.1e-3'),
1001: (8)                   (0.03141592654, 1000000.0, '3.1e-2'),
1002: (8)                   (0.3141592654, 1000000.0, '3.1e-1'),
1003: (8)                   (3.141592654, 1000000.0, '3.1'),
1004: (8)                   (31.41592654, 1000000.0, '3.1e1'),
1005: (8)                   (314.1592654, 1000000.0, '3.1e2'),
1006: (8)                   (3141.592654, 1000000.0, '3.1e3'),
1007: (8)                   (31415.92654, 1000000.0, '3.1e4'),
1008: (8)                   (314159.2654, 1000000.0, '3.1e5'),
1009: (8)                   (1e-05, 1000000.0, '1e-5'),
1010: (8)                   (0.0001, 1000000.0, '1e-4'),
1011: (8)                   (0.001, 1000000.0, '1e-3'),
1012: (8)                   (0.01, 1000000.0, '1e-2'),
1013: (8)                   (0.1, 1000000.0, '1e-1'),
1014: (8)                   (1, 1000000.0, '1'),
1015: (8)                   (10, 1000000.0, '10'),
1016: (8)                   (100, 1000000.0, '100'),
1017: (8)                   (1000, 1000000.0, '1000'),
1018: (8)                   (10000, 1000000.0, '1e4'),
1019: (8)                   (100000, 1000000.0, '1e5'),
1020: (4)               ]
1021: (4)               @pytest.mark.parametrize('value, domain, expected', pprint_data)
1022: (4)               def test_pprint(self, value, domain, expected):
1023: (8)                   fmt = mticker.LogFormatter()
1024: (8)                   label = fmt._pprint_val(value, domain)
1025: (8)                   assert label == expected
1026: (4)               @pytest.mark.parametrize('value, long, short', [
1027: (8)                   (0.0, "0", "0"),
1028: (8)                   (0, "0", "0"),
1029: (8)                   (-1.0, "-10^0", "-1"),
1030: (8)                   (2e-10, "2x10^-10", "2e-10"),
1031: (8)                   (1e10, "10^10", "1e+10"),
1032: (4)               ])
1033: (4)               def test_format_data(self, value, long, short):
1034: (8)                   fig, ax = plt.subplots()
1035: (8)                   ax.set_xscale('log')
1036: (8)                   fmt = ax.xaxis.get_major_formatter()
1037: (8)                   assert fmt.format_data(value) == long
1038: (8)                   assert fmt.format_data_short(value) == short
1039: (4)               def _sub_labels(self, axis, subs=()):
1040: (8)                   """Test whether locator marks subs to be labeled."""
1041: (8)                   fmt = axis.get_minor_formatter()
1042: (8)                   minor_tlocs = axis.get_minorticklocs()
1043: (8)                   fmt.set_locs(minor_tlocs)
1044: (8)                   coefs = minor_tlocs / 10**(np.floor(np.log10(minor_tlocs)))
1045: (8)                   label_expected = [round(c) in subs for c in coefs]
1046: (8)                   label_test = [fmt(x) != '' for x in minor_tlocs]
1047: (8)                   assert label_test == label_expected
1048: (4)               @mpl.style.context('default')
1049: (4)               def test_sublabel(self):
1050: (8)                   fig, ax = plt.subplots()
1051: (8)                   ax.set_xscale('log')
1052: (8)                   ax.xaxis.set_major_locator(mticker.LogLocator(base=10, subs=[]))
1053: (8)                   ax.xaxis.set_minor_locator(mticker.LogLocator(base=10,
1054: (54)                                                          subs=np.arange(2, 10)))
1055: (8)                   ax.xaxis.set_major_formatter(mticker.LogFormatter(labelOnlyBase=True))
1056: (8)
ax.xaxis.set_minor_formatter(mticker.LogFormatter(labelOnlyBase=False))
1057: (8)                   ax.set_xlim(1, 1e4)
1058: (8)                   fmt = ax.xaxis.get_major_formatter()
1059: (8)                   fmt.set_locs(ax.xaxis.get_majorticklocs())
1060: (8)                   show_major_labels = [fmt(x) != ''
1061: (29)                                       for x in ax.xaxis.get_majorticklocs()]
```

```
(8)                 assert np.all(show_major_labels)
(8)                 self._sub_labels(ax.xaxis, subs=[])
(8)                 ax.set_xlim(1, 800)
(8)                 self._sub_labels(ax.xaxis, subs=[])
(8)                 ax.set_xlim(1, 80)
(8)                 self._sub_labels(ax.xaxis, subs=[])
(8)                 ax.set_xlim(1, 8)
(8)                 self._sub_labels(ax.xaxis, subs=[2, 3, 4, 6])
(8)                 ax.set_xlim(0.5, 0.9)
(8)                 self._sub_labels(ax.xaxis, subs=np.arange(2, 10, dtype=int))
(4)             @pytest.mark.parametrize('val', [1, 10, 100, 1000])
(4)             def test_LogFormatter_call(self, val):
(8)                 temp_lf = mticker.LogFormatter()
(8)                 temp_lf.create_dummy_axis()
(8)                 temp_lf.axis.set_view_interval(1, 10)
(8)                 assert temp_lf(val) == str(val)
(4)             @pytest.mark.parametrize('val', [1e-323, 2e-323, 10e-323, 11e-323])
(4)             def test_LogFormatter_call_tiny(self, val):
(8)                 temp_lf = mticker.LogFormatter()
(8)                 temp_lf.create_dummy_axis()
(8)                 temp_lf.axis.set_view_interval(1, 10)
(8)                 temp_lf(val)
(0)         class TestLogitFormatter:
(4)             @staticmethod
(4)             def logit_deformatter(string):
(8)                 r"""
(8)                 Parser to convert string as r'$\mathdefault{1.41\cdot10^{-4}}$' in
(8)                 float 1.41e-4, as '0.5' or as r'$\mathdefault{\frac{1}{2}}$' in float
(8)                 0.5,
(8)                 """
(8)                 match = re.match(
(12)                    r"[^\d]*"
(12)                    r"(?P<comp>1-)?"
(12)                    r"(?P<mant>\d*\.?\d*)?"
(12)                    r"(?:\\cdot)?"
(12)                    r"(?:10\^\{(?P<expo>-?\d*)})?"
(12)                    r"[^\d]*$",
(12)                    string,
(8)                 )
(8)                 if match:
(12)                    comp = match["comp"] is not None
(12)                    mantissa = float(match["mant"]) if match["mant"] else 1
(12)                    expo = int(match["expo"]) if match["expo"] is not None else 0
(12)                    value = mantissa * 10 ** expo
(12)                    if match["mant"] or match["expo"] is not None:
(16)                        if comp:
(20)                            return 1 - value
(16)                        return value
(8)                 match = re.match(
(12)                    r"[^\d]*\\frac\{(?P<num>\d+)\}\{(?P<deno>\d+)\}[^\d]*$", string
(8)                 )
(8)                 if match:
(12)                    num, deno = float(match["num"]), float(match["deno"])
(12)                    return num / deno
(8)                 raise ValueError("Not formatted by LogitFormatter")
(4)             @pytest.mark.parametrize(
(8)                 "fx, x",
(8)                 [
(12)                    (r"STUFF0.41OTHERSTUFF", 0.41),
(12)                    (r"STUFF1.41\cdot10^{-2}OTHERSTUFF", 1.41e-2),
(12)                    (r"STUFF1-0.41OTHERSTUFF", 1 - 0.41),
(12)                    (r"STUFF1-1.41\cdot10^{-2}OTHERSTUFF", 1 - 1.41e-2),
(12)                    (r"STUFF", None),
(12)                    (r"STUFF12.4e-3OTHERSTUFF", None),
(8)                 ],
(4)             )
(4)             def test_logit_deformater(self, fx, x):
(8)                 if x is None:
(12)                    with pytest.raises(ValueError):
```

```
(16)                        TestLogitFormatter.logit_deformatter(fx)
(8)                 else:
(12)                    y = TestLogitFormatter.logit_deformatter(fx)
(12)                    assert _LogitHelper.isclose(x, y)
(4)             decade_test = sorted(
(8)                 [10 ** (-i) for i in range(1, 10)]
(8)                 + [1 - 10 ** (-i) for i in range(1, 10)]
(8)                 + [1 / 2]
(4)             )
(4)             @pytest.mark.parametrize("x", decade_test)
(4)             def test_basic(self, x):
(8)                 """
(8)                 Test the formatted value correspond to the value for ideal ticks in
(8)                 logit space.
(8)                 """
(8)                 formatter = mticker.LogitFormatter(use_overline=False)
(8)                 formatter.set_locs(self.decade_test)
(8)                 s = formatter(x)
(8)                 x2 = TestLogitFormatter.logit_deformatter(s)
(8)                 assert _LogitHelper.isclose(x, x2)
(4)             @pytest.mark.parametrize("x", (-1, -0.5, -0.1, 1.1, 1.5, 2))
(4)             def test_invalid(self, x):
(8)                 """
(8)                 Test that invalid value are formatted with empty string without
(8)                 raising exception.
(8)                 """
(8)                 formatter = mticker.LogitFormatter(use_overline=False)
(8)                 formatter.set_locs(self.decade_test)
(8)                 s = formatter(x)
(8)                 assert s == ""
(4)             @pytest.mark.parametrize("x", 1 / (1 + np.exp(-np.linspace(-7, 7, 10))))
(4)             def test_variablelength(self, x):
(8)                 """
(8)                 The format length should change depending on the neighbor labels.
(8)                 """
(8)                 formatter = mticker.LogitFormatter(use_overline=False)
(8)                 for N in (10, 20, 50, 100, 200, 1000, 2000, 5000, 10000):
(12)                    if x + 1 / N < 1:
(16)                        formatter.set_locs([x - 1 / N, x, x + 1 / N])
(16)                        sx = formatter(x)
(16)                        sx1 = formatter(x + 1 / N)
(16)                        d = (
(20)                            TestLogitFormatter.logit_deformatter(sx1)
(20)                            - TestLogitFormatter.logit_deformatter(sx)
(16)                        )
(16)                        assert 0 < d < 2 / N
(4)             lims_minor_major = [
(8)                 (True, (5e-8, 1 - 5e-8), ((25, False), (75, False))),
(8)                 (True, (5e-5, 1 - 5e-5), ((25, False), (75, True))),
(8)                 (True, (5e-2, 1 - 5e-2), ((25, True), (75, True))),
(8)                 (False, (0.75, 0.76, 0.77), ((7, True), (25, True), (75, True))),
(4)             ]
(4)             @pytest.mark.parametrize("method, lims, cases", lims_minor_major)
(4)             def test_minor_vs_major(self, method, lims, cases):
(8)                 """
(8)                 Test minor/major displays.
(8)                 """
(8)                 if method:
(12)                    min_loc = mticker.LogitLocator(minor=True)
(12)                    ticks = min_loc.tick_values(*lims)
(8)                 else:
(12)                    ticks = np.array(lims)
(8)                 min_form = mticker.LogitFormatter(minor=True)
(8)                 for threshold, has_minor in cases:
(12)                    min_form.set_minor_threshold(threshold)
(12)                    formatted = min_form.format_ticks(ticks)
(12)                    labelled = [f for f in formatted if len(f) > 0]
(12)                    if has_minor:
(16)                        assert len(labelled) > 0, (threshold, has_minor)
```

```
(12)                    else:
(16)                        assert len(labelled) == 0, (threshold, has_minor)
(4)             def test_minor_number(self):
(8)                 """
(8)                 Test the parameter minor_number
(8)                 """
(8)                 min_loc = mticker.LogitLocator(minor=True)
(8)                 min_form = mticker.LogitFormatter(minor=True)
(8)                 ticks = min_loc.tick_values(5e-2, 1 - 5e-2)
(8)                 for minor_number in (2, 4, 8, 16):
(12)                    min_form.set_minor_number(minor_number)
(12)                    formatted = min_form.format_ticks(ticks)
(12)                    labelled = [f for f in formatted if len(f) > 0]
(12)                    assert len(labelled) == minor_number
(4)             def test_use_overline(self):
(8)                 """
(8)                 Test the parameter use_overline
(8)                 """
(8)                 x = 1 - 1e-2
(8)                 fx1 = r"$\mathdefault{1-10^{-2}}$"
(8)                 fx2 = r"$\mathdefault{\overline{10^{-2}}}$"
(8)                 form = mticker.LogitFormatter(use_overline=False)
(8)                 assert form(x) == fx1
(8)                 form.use_overline(True)
(8)                 assert form(x) == fx2
(8)                 form.use_overline(False)
(8)                 assert form(x) == fx1
(4)             def test_one_half(self):
(8)                 """
(8)                 Test the parameter one_half
(8)                 """
(8)                 form = mticker.LogitFormatter()
(8)                 assert r"\frac{1}{2}" in form(1/2)
(8)                 form.set_one_half("1/2")
(8)                 assert "1/2" in form(1/2)
(8)                 form.set_one_half("one half")
(8)                 assert "one half" in form(1/2)
(4)             @pytest.mark.parametrize("N", (100, 253, 754))
(4)             def test_format_data_short(self, N):
(8)                 locs = np.linspace(0, 1, N)[1:-1]
(8)                 form = mticker.LogitFormatter()
(8)                 for x in locs:
(12)                    fx = form.format_data_short(x)
(12)                    if fx.startswith("1-"):
(16)                        x2 = 1 - float(fx[2:])
(12)                    else:
(16)                        x2 = float(fx)
(12)                    assert abs(x - x2) < 1 / N
(0)         class TestFormatStrFormatter:
(4)             def test_basic(self):
(8)                 tmp_form = mticker.FormatStrFormatter('%05d')
(8)                 assert '00002' == tmp_form(2)
(0)         class TestStrMethodFormatter:
(4)             test_data = [
(8)                 ('{x:05d}', (2,), False, '00002'),
(8)                 ('{x:05d}', (2,), True, '00002'),
(8)                 ('{x:05d}', (-2,), False, '-0002'),
(8)                 ('{x:05d}', (-2,), True, '\N{MINUS SIGN}0002'),
(8)                 ('{x:03d}-{pos:02d}', (2, 1), False, '002-01'),
(8)                 ('{x:03d}-{pos:02d}', (2, 1), True, '002-01'),
(8)                 ('{x:03d}-{pos:02d}', (-2, 1), False, '-02-01'),
(8)                 ('{x:03d}-{pos:02d}', (-2, 1), True, '\N{MINUS SIGN}02-01'),
(4)             ]
(4)             @pytest.mark.parametrize('format, input, unicode_minus, expected',
test_data)
(4)             def test_basic(self, format, input, unicode_minus, expected):
(8)                 with mpl.rc_context({"axes.unicode_minus": unicode_minus}):
(12)                    fmt = mticker.StrMethodFormatter(format)
(12)                    assert fmt(*input) == expected
```

```
(0)           class TestEngFormatter:
(4)               raw_format_data = [
(8)                   (False, -1234.56789, ('-1.23457 k', '-1 k', '-1.23 k')),
(8)                   (True, -1234.56789, ('\N{MINUS SIGN}1.23457 k', '\N{MINUS SIGN}1 k',
(29)                                       '\N{MINUS SIGN}1.23 k')),
(8)                   (False, -1.23456789, ('-1.23457', '-1', '-1.23')),
(8)                   (True, -1.23456789, ('\N{MINUS SIGN}1.23457', '\N{MINUS SIGN}1',
(29)                                       '\N{MINUS SIGN}1.23')),
(8)                   (False, -0.123456789, ('-123.457 m', '-123 m', '-123.46 m')),
(8)                   (True, -0.123456789, ('\N{MINUS SIGN}123.457 m', '\N{MINUS SIGN}123
m',
(30)                                        '\N{MINUS SIGN}123.46 m')),
(8)                   (False, -0.00123456789, ('-1.23457 m', '-1 m', '-1.23 m')),
(8)                   (True, -0.00123456789, ('\N{MINUS SIGN}1.23457 m', '\N{MINUS SIGN}1
m',
(32)                                          '\N{MINUS SIGN}1.23 m')),
(8)                   (True, -0.0, ('0', '0', '0.00')),
(8)                   (True, -0, ('0', '0', '0.00')),
(8)                   (True, 0, ('0', '0', '0.00')),
(8)                   (True, 1.23456789e-6, ('1.23457 µ', '1 µ', '1.23 µ')),
(8)                   (True, 0.123456789, ('123.457 m', '123 m', '123.46 m')),
(8)                   (True, 0.1, ('100 m', '100 m', '100.00 m')),
(8)                   (True, 1, ('1', '1', '1.00')),
(8)                   (True, 1.23456789, ('1.23457', '1', '1.23')),
(8)                   (True, 999.9, ('999.9', '1 k', '999.90')),
(8)                   (True, 999.9999, ('1 k', '1 k', '1.00 k')),
(8)                   (False, -999.9999, ('-1 k', '-1 k', '-1.00 k')),
(8)                   (True, -999.9999, ('\N{MINUS SIGN}1 k', '\N{MINUS SIGN}1 k',
(27)                                     '\N{MINUS SIGN}1.00 k')),
(8)                   (True, 1000, ('1 k', '1 k', '1.00 k')),
(8)                   (True, 1001, ('1.001 k', '1 k', '1.00 k')),
(8)                   (True, 100001, ('100.001 k', '100 k', '100.00 k')),
(8)                   (True, 987654.321, ('987.654 k', '988 k', '987.65 k')),
(8)                   (True, 1.23e33, ('1230 Q', '1230 Q', '1230.00 Q'))
(4)               ]
(4)               @pytest.mark.parametrize('unicode_minus, input, expected',
raw_format_data)
(4)               def test_params(self, unicode_minus, input, expected):
(8)                   """
(8)                   Test the formatting of EngFormatter for various values of the 'places'
(8)                   argument, in several cases:
(8)                   0. without a unit symbol but with a (default) space separator;
(8)                   1. with both a unit symbol and a (default) space separator;
(8)                   2. with both a unit symbol and some non default separators;
(8)                   3. without a unit symbol but with some non default separators.
(8)                   Note that cases 2. and 3. are looped over several separator strings.
(8)                   """
(8)                   plt.rcParams['axes.unicode_minus'] = unicode_minus
(8)                   UNIT = 's'  # seconds
(8)                   DIGITS = '0123456789'  # %timeit showed 10-20% faster search than set
(8)                   exp_outputs = expected
(8)                   formatters = (
(12)                      mticker.EngFormatter(),  # places=None (default)
(12)                      mticker.EngFormatter(places=0),
(12)                      mticker.EngFormatter(places=2)
(8)                   )
(8)                   for _formatter, _exp_output in zip(formatters, exp_outputs):
(12)                      assert _formatter(input) == _exp_output
(8)                   exp_outputs = (_s + " " + UNIT if _s[-1] in DIGITS  # case w/o prefix
(23)                                 else _s + UNIT for _s in expected)
(8)                   formatters = (
(12)                      mticker.EngFormatter(unit=UNIT),  # places=None (default)
(12)                      mticker.EngFormatter(unit=UNIT, places=0),
(12)                      mticker.EngFormatter(unit=UNIT, places=2)
(8)                   )
(8)                   for _formatter, _exp_output in zip(formatters, exp_outputs):
(12)                      assert _formatter(input) == _exp_output
(8)                   for _sep in ("", "\N{NARROW NO-BREAK SPACE}", "@_@"):
(12)                      exp_outputs = (_s + _sep + UNIT if _s[-1] in DIGITS  # no prefix
```

```
(27)                                    else _s.replace(" ", _sep) + UNIT
(27)                                    for _s in expected)
(12)                        formatters = (
(16)                            mticker.EngFormatter(unit=UNIT, sep=_sep),  # places=None
(16)                            mticker.EngFormatter(unit=UNIT, places=0, sep=_sep),
(16)                            mticker.EngFormatter(unit=UNIT, places=2, sep=_sep)
(12)                        )
(12)                        for _formatter, _exp_output in zip(formatters, exp_outputs):
(16)                            assert _formatter(input) == _exp_output
(12)                        exp_outputs = (_s.replace(" ", _sep) for _s in expected)
(12)                        formatters = (
(16)                            mticker.EngFormatter(sep=_sep),  # places=None (default)
(16)                            mticker.EngFormatter(places=0, sep=_sep),
(16)                            mticker.EngFormatter(places=2, sep=_sep)
(12)                        )
(12)                        for _formatter, _exp_output in zip(formatters, exp_outputs):
(16)                            assert _formatter(input) == _exp_output
(0)             def test_engformatter_usetex_useMathText():
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot([0, 500, 1000], [0, 500, 1000])
(4)                 ax.set_xticks([0, 500, 1000])
(4)                 for formatter in (mticker.EngFormatter(usetex=True),
(22)                                  mticker.EngFormatter(useMathText=True)):
(8)                     ax.xaxis.set_major_formatter(formatter)
(8)                     fig.canvas.draw()
(8)                     x_tick_label_text = [labl.get_text() for labl in ax.get_xticklabels()]
(8)                     assert x_tick_label_text == ['$0$', '$500$', '$1$ k']
(0)             class TestPercentFormatter:
(4)                 percent_data = [
(8)                     (100, 0, '%', 120, 100, '120%'),
(8)                     (100, 0, '%', 100, 90, '100%'),
(8)                     (100, 0, '%', 90, 50, '90%'),
(8)                     (100, 0, '%', -1.7, 40, '-2%'),
(8)                     (100, 1, '%', 90.0, 100, '90.0%'),
(8)                     (100, 1, '%', 80.1, 90, '80.1%'),
(8)                     (100, 1, '%', 70.23, 50, '70.2%'),
(8)                     (100, 1, '%', -60.554, 40, '-60.6%'),
(8)                     (100, None, '%', 95, 1, '95.00%'),
(8)                     (1.0, None, '%', 3, 6, '300%'),
(8)                     (17.0, None, '%', 1, 8.5, '6%'),
(8)                     (17.0, None, '%', 1, 8.4, '5.9%'),
(8)                     (5, None, '%', -100, 0.000001, '-2000.00000%'),
(8)                     (1.0, 2, None, 1.2, 100, '120.00'),
(8)                     (75, 3, '', 50, 100, '66.667'),
(8)                     (42, None, '^^Foobar$$', 21, 12, '50.0^^Foobar$$'),
(4)                 ]
(4)                 percent_ids = [
(8)                     'decimals=0, x>100%',
(8)                     'decimals=0, x=100%',
(8)                     'decimals=0, x<100%',
(8)                     'decimals=0, x<0%',
(8)                     'decimals=1, x>100%',
(8)                     'decimals=1, x=100%',
(8)                     'decimals=1, x<100%',
(8)                     'decimals=1, x<0%',
(8)                     'autodecimal, x<100%, display_range=1',
(8)                     'autodecimal, x>100%, display_range=6 (custom xmax test)',
(8)                     'autodecimal, x<100%, display_range=8.5 (autodecimal test 1)',
(8)                     'autodecimal, x<100%, display_range=8.4 (autodecimal test 2)',
(8)                     'autodecimal, x<-100%, display_range=1e-6 (tiny display range)',
(8)                     'None as percent symbol',
(8)                     'Empty percent symbol',
(8)                     'Custom percent symbol',
(4)                 ]
(4)                 latex_data = [
(8)                     (False, False, r'50\{t}%'),
(8)                     (False, True, r'50\\\{t\}\%'),
(8)                     (True, False, r'50\{t}%'),
(8)                     (True, True, r'50\{t}%'),
```

```
(4)                 ]
(4)                 @pytest.mark.parametrize(
(12)                        'xmax, decimals, symbol, x, display_range, expected',
(12)                        percent_data, ids=percent_ids)
(4)                 def test_basic(self, xmax, decimals, symbol,
(19)                               x, display_range, expected):
(8)                     formatter = mticker.PercentFormatter(xmax, decimals, symbol)
(8)                     with mpl.rc_context(rc={'text.usetex': False}):
(12)                        assert formatter.format_pct(x, display_range) == expected
(4)                 @pytest.mark.parametrize('is_latex, usetex, expected', latex_data)
(4)                 def test_latex(self, is_latex, usetex, expected):
(8)                     fmt = mticker.PercentFormatter(symbol='\\{t}%', is_latex=is_latex)
(8)                     with mpl.rc_context(rc={'text.usetex': usetex}):
(12)                        assert fmt.format_pct(50, 100) == expected
(0)             def _impl_locale_comma():
(4)                 try:
(8)                     locale.setlocale(locale.LC_ALL, 'de_DE.UTF-8')
(4)                 except locale.Error:
(8)                     print('SKIP: Locale de_DE.UTF-8 is not supported on this machine')
(8)                     return
(4)                 ticks = mticker.ScalarFormatter(useMathText=True, useLocale=True)
(4)                 fmt = '$\\mathdefault{%1.1f}$'
(4)                 x = ticks._format_maybe_minus_and_locale(fmt, 0.5)
(4)                 assert x == '$\\mathdefault{0{,}5}$'
(4)                 fmt = ',$\\mathdefault{,%1.1f},$'
(4)                 x = ticks._format_maybe_minus_and_locale(fmt, 0.5)
(4)                 assert x == ',$\\mathdefault{,0{,}5},$'
(4)                 ticks = mticker.ScalarFormatter(useMathText=False, useLocale=True)
(4)                 fmt = '%1.1f'
(4)                 x = ticks._format_maybe_minus_and_locale(fmt, 0.5)
(4)                 assert x == '0,5'
(0)             def test_locale_comma():
(4)                 proc = mpl.testing.subprocess_run_helper(_impl_locale_comma, timeout=60,
(45)                                                         extra_env={'MPLBACKEND': 'Agg'})
(4)                 skip_msg = next((line[len('SKIP:'):].strip()
(21)                                 for line in proc.stdout.splitlines()
(21)                                 if line.startswith('SKIP:')),
(20)                                '')
(4)                 if skip_msg:
(8)                     pytest.skip(skip_msg)
(0)             def test_majformatter_type():
(4)                 fig, ax = plt.subplots()
(4)                 with pytest.raises(TypeError):
(8)                     ax.xaxis.set_major_formatter(mticker.LogLocator())
(0)             def test_minformatter_type():
(4)                 fig, ax = plt.subplots()
(4)                 with pytest.raises(TypeError):
(8)                     ax.xaxis.set_minor_formatter(mticker.LogLocator())
(0)             def test_majlocator_type():
(4)                 fig, ax = plt.subplots()
(4)                 with pytest.raises(TypeError):
(8)                     ax.xaxis.set_major_locator(mticker.LogFormatter())
(0)             def test_minlocator_type():
(4)                 fig, ax = plt.subplots()
(4)                 with pytest.raises(TypeError):
(8)                     ax.xaxis.set_minor_locator(mticker.LogFormatter())
(0)             def test_minorticks_rc():
(4)                 fig = plt.figure()
(4)                 def minorticksubplot(xminor, yminor, i):
(8)                     rc = {'xtick.minor.visible': xminor,
(14)                          'ytick.minor.visible': yminor}
(8)                     with plt.rc_context(rc=rc):
(12)                        ax = fig.add_subplot(2, 2, i)
(8)                     assert (len(ax.xaxis.get_minor_ticks()) > 0) == xminor
(8)                     assert (len(ax.yaxis.get_minor_ticks()) > 0) == yminor
(4)                 minorticksubplot(False, False, 1)
(4)                 minorticksubplot(True, False, 2)
(4)                 minorticksubplot(False, True, 3)
(4)                 minorticksubplot(True, True, 4)
```

```
(0)  def test_minorticks_toggle():
(4)      """
(4)      Test toggling minor ticks
(4)      Test `.Axis.minorticks_on()` and `.Axis.minorticks_off()`. Testing is
(4)      limited to a subset of built-in scales - `'linear'`, `'log'`, `'asinh'`
(4)      and `'logit'`. `symlog` scale does not seem to have a working minor
(4)      locator and is omitted. In future, this test should cover all scales in
(4)      `matplotlib.scale.get_scale_names()`.
(4)      """
(4)      fig = plt.figure()
(4)      def minortickstoggle(xminor, yminor, scale, i):
(8)          ax = fig.add_subplot(2, 2, i)
(8)          ax.set_xscale(scale)
(8)          ax.set_yscale(scale)
(8)          if not xminor and not yminor:
(12)             ax.minorticks_off()
(8)          if xminor and not yminor:
(12)             ax.xaxis.minorticks_on()
(12)             ax.yaxis.minorticks_off()
(8)          if not xminor and yminor:
(12)             ax.xaxis.minorticks_off()
(12)             ax.yaxis.minorticks_on()
(8)          if xminor and yminor:
(12)             ax.minorticks_on()
(8)          assert (len(ax.xaxis.get_minor_ticks()) > 0) == xminor
(8)          assert (len(ax.yaxis.get_minor_ticks()) > 0) == yminor
(4)      scales = ['linear', 'log', 'asinh', 'logit']
(4)      for scale in scales:
(8)          minortickstoggle(False, False, scale, 1)
(8)          minortickstoggle(True, False, scale, 2)
(8)          minortickstoggle(False, True, scale, 3)
(8)          minortickstoggle(True, True, scale, 4)
(8)          fig.clear()
(4)      plt.close(fig)
(0)  @pytest.mark.parametrize('remove_overlapping_locs, expected_num',
(25)                            ((True, 6),
(26)                             (None, 6),  # this tests the default
(26)                             (False, 9)))
(0)  def test_remove_overlap(remove_overlapping_locs, expected_num):
(4)      t = np.arange("2018-11-03", "2018-11-06", dtype="datetime64")
(4)      x = np.ones(len(t))
(4)      fig, ax = plt.subplots()
(4)      ax.plot(t, x)
(4)      ax.xaxis.set_major_locator(mpl.dates.DayLocator())
(4)      ax.xaxis.set_major_formatter(mpl.dates.DateFormatter('\n%a'))
(4)      ax.xaxis.set_minor_locator(mpl.dates.HourLocator((0, 6, 12, 18)))
(4)      ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%H:%M'))
(4)      ax.xaxis.get_minor_ticks(15)
(4)      if remove_overlapping_locs is not None:
(8)          ax.xaxis.remove_overlapping_locs = remove_overlapping_locs
(4)      current = ax.xaxis.remove_overlapping_locs
(4)      assert (current == ax.xaxis.get_remove_overlapping_locs())
(4)      plt.setp(ax.xaxis, remove_overlapping_locs=current)
(4)      new = ax.xaxis.remove_overlapping_locs
(4)      assert (new == ax.xaxis.remove_overlapping_locs)
(4)      assert len(ax.xaxis.get_minorticklocs()) == expected_num
(4)      assert len(ax.xaxis.get_minor_ticks()) == expected_num
(4)      assert len(ax.xaxis.get_minorticklabels()) == expected_num
(4)      assert len(ax.xaxis.get_minorticklines()) == expected_num*2
(0)  @pytest.mark.parametrize('sub', [
(4)      ['hi', 'aardvark'],
(4)      np.zeros((2, 2))])
(0)  def test_bad_locator_subs(sub):
(4)      ll = mticker.LogLocator()
(4)      with pytest.raises(ValueError):
(8)          ll.set_params(subs=sub)
(0)  @pytest.mark.parametrize('numticks', [1, 2, 3, 9])
(0)  @mpl.style.context('default')
(0)  def test_small_range_loglocator(numticks):
```

```
1541: (4)               ll = mticker.LogLocator()
1542: (4)               ll.set_params(numticks=numticks)
1543: (4)               for top in [5, 7, 9, 11, 15, 50, 100, 1000]:
1544: (8)                   ticks = ll.tick_values(.5, top)
1545: (8)                   assert (np.diff(np.log10(ll.tick_values(6, 150))) == 1).all()
1546: (0)           def test_NullFormatter():
1547: (4)               formatter = mticker.NullFormatter()
1548: (4)               assert formatter(1.0) == ''
1549: (4)               assert formatter.format_data(1.0) == ''
1550: (4)               assert formatter.format_data_short(1.0) == ''
1551: (0)           @pytest.mark.parametrize('formatter', (
1552: (4)               mticker.FuncFormatter(lambda a: f'val: {a}'),
1553: (4)               mticker.FixedFormatter(('foo', 'bar'))))
1554: (0)           def test_set_offset_string(formatter):
1555: (4)               assert formatter.get_offset() == ''
1556: (4)               formatter.set_offset_string('mpl')
1557: (4)               assert formatter.get_offset() == 'mpl'
1558: (0)           def test_minorticks_on_multi_fig():
1559: (4)               """
1560: (4)               Turning on minor gridlines in a multi-Axes Figure
1561: (4)               that contains more than one boxplot and shares the x-axis
1562: (4)               should not raise an exception.
1563: (4)               """
1564: (4)               fig, ax = plt.subplots()
1565: (4)               ax.boxplot(np.arange(10), positions=[0])
1566: (4)               ax.boxplot(np.arange(10), positions=[0])
1567: (4)               ax.boxplot(np.arange(10), positions=[1])
1568: (4)               ax.grid(which="major")
1569: (4)               ax.grid(which="minor")
1570: (4)               ax.minorticks_on()
1571: (4)               fig.draw_without_rendering()
1572: (4)               assert ax.get_xgridlines()
1573: (4)               assert isinstance(ax.xaxis.get_minor_locator(),
mpl.ticker.AutoMinorLocator)

----------------------------------------

File 223 - test_tightlayout.py:

1: (0)              import warnings
2: (0)              import numpy as np
3: (0)              from numpy.testing import assert_array_equal
4: (0)              import pytest
5: (0)              import matplotlib as mpl
6: (0)              from matplotlib.testing.decorators import image_comparison
7: (0)              import matplotlib.pyplot as plt
8: (0)              from matplotlib.offsetbox import AnchoredOffsetbox, DrawingArea
9: (0)              from matplotlib.patches import Rectangle
10: (0)             def example_plot(ax, fontsize=12):
11: (4)                 ax.plot([1, 2])
12: (4)                 ax.locator_params(nbins=3)
13: (4)                 ax.set_xlabel('x-label', fontsize=fontsize)
14: (4)                 ax.set_ylabel('y-label', fontsize=fontsize)
15: (4)                 ax.set_title('Title', fontsize=fontsize)
16: (0)             @image_comparison(['tight_layout1'], tol=1.9)
17: (0)             def test_tight_layout1():
18: (4)                 """Test tight_layout for a single subplot."""
19: (4)                 fig, ax = plt.subplots()
20: (4)                 example_plot(ax, fontsize=24)
21: (4)                 plt.tight_layout()
22: (0)             @image_comparison(['tight_layout2'])
23: (0)             def test_tight_layout2():
24: (4)                 """Test tight_layout for multiple subplots."""
25: (4)                 fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
26: (4)                 example_plot(ax1)
27: (4)                 example_plot(ax2)
28: (4)                 example_plot(ax3)
29: (4)                 example_plot(ax4)
30: (4)                 plt.tight_layout()
```

```
(0)             @image_comparison(['tight_layout3'])
(0)             def test_tight_layout3():
(4)                 """Test tight_layout for multiple subplots."""
(4)                 ax1 = plt.subplot(221)
(4)                 ax2 = plt.subplot(223)
(4)                 ax3 = plt.subplot(122)
(4)                 example_plot(ax1)
(4)                 example_plot(ax2)
(4)                 example_plot(ax3)
(4)                 plt.tight_layout()
(0)             @image_comparison(['tight_layout4'], freetype_version=('2.5.5', '2.6.1'),
(18)                              tol=0.015)
(0)             def test_tight_layout4():
(4)                 """Test tight_layout for subplot2grid."""
(4)                 ax1 = plt.subplot2grid((3, 3), (0, 0))
(4)                 ax2 = plt.subplot2grid((3, 3), (0, 1), colspan=2)
(4)                 ax3 = plt.subplot2grid((3, 3), (1, 0), colspan=2, rowspan=2)
(4)                 ax4 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)
(4)                 example_plot(ax1)
(4)                 example_plot(ax2)
(4)                 example_plot(ax3)
(4)                 example_plot(ax4)
(4)                 plt.tight_layout()
(0)             @image_comparison(['tight_layout5'])
(0)             def test_tight_layout5():
(4)                 """Test tight_layout for image."""
(4)                 ax = plt.subplot()
(4)                 arr = np.arange(100).reshape((10, 10))
(4)                 ax.imshow(arr, interpolation="none")
(4)                 plt.tight_layout()
(0)             @image_comparison(['tight_layout6'])
(0)             def test_tight_layout6():
(4)                 """Test tight_layout for gridspec."""
(4)                 with warnings.catch_warnings():
(8)                     warnings.simplefilter("ignore", UserWarning)
(8)                     fig = plt.figure()
(8)                     gs1 = mpl.gridspec.GridSpec(2, 1)
(8)                     ax1 = fig.add_subplot(gs1[0])
(8)                     ax2 = fig.add_subplot(gs1[1])
(8)                     example_plot(ax1)
(8)                     example_plot(ax2)
(8)                     gs1.tight_layout(fig, rect=[0, 0, 0.5, 1])
(8)                     gs2 = mpl.gridspec.GridSpec(3, 1)
(8)                     for ss in gs2:
(12)                        ax = fig.add_subplot(ss)
(12)                        example_plot(ax)
(12)                        ax.set_title("")
(12)                        ax.set_xlabel("")
(8)                     ax.set_xlabel("x-label", fontsize=12)
(8)                     gs2.tight_layout(fig, rect=[0.5, 0, 1, 1], h_pad=0.45)
(8)                     top = min(gs1.top, gs2.top)
(8)                     bottom = max(gs1.bottom, gs2.bottom)
(8)                     gs1.tight_layout(fig, rect=[None, 0 + (bottom-gs1.bottom),
(36)                                                 0.5, 1 - (gs1.top-top)])
(8)                     gs2.tight_layout(fig, rect=[0.5, 0 + (bottom-gs2.bottom),
(36)                                                 None, 1 - (gs2.top-top)],
(25)                                    h_pad=0.45)
(0)             @image_comparison(['tight_layout7'], tol=1.9)
(0)             def test_tight_layout7():
(4)                 fontsize = 24
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot([1, 2])
(4)                 ax.locator_params(nbins=3)
(4)                 ax.set_xlabel('x-label', fontsize=fontsize)
(4)                 ax.set_ylabel('y-label', fontsize=fontsize)
(4)                 ax.set_title('Left Title', loc='left', fontsize=fontsize)
(4)                 ax.set_title('Right Title', loc='right', fontsize=fontsize)
(4)                 plt.tight_layout()
(0)             @image_comparison(['tight_layout8'], tol=0.005)
```

```
(0)             def test_tight_layout8():
(4)                 """Test automatic use of tight_layout."""
(4)                 fig = plt.figure()
(4)                 fig.set_layout_engine(layout='tight', pad=0.1)
(4)                 ax = fig.add_subplot()
(4)                 example_plot(ax, fontsize=24)
(4)                 fig.draw_without_rendering()
(0)             @image_comparison(['tight_layout9'])
(0)             def test_tight_layout9():
(4)                 f, axarr = plt.subplots(2, 2)
(4)                 axarr[1][1].set_visible(False)
(4)                 plt.tight_layout()
(0)             def test_outward_ticks():
(4)                 """Test automatic use of tight_layout."""
(4)                 fig = plt.figure()
(4)                 ax = fig.add_subplot(221)
(4)                 ax.xaxis.set_tick_params(tickdir='out', length=16, width=3)
(4)                 ax.yaxis.set_tick_params(tickdir='out', length=16, width=3)
(4)                 ax.xaxis.set_tick_params(
(8)                     tickdir='out', length=32, width=3, tick1On=True, which='minor')
(4)                 ax.yaxis.set_tick_params(
(8)                     tickdir='out', length=32, width=3, tick1On=True, which='minor')
(4)                 ax.xaxis.set_ticks([0], minor=True)
(4)                 ax.yaxis.set_ticks([0], minor=True)
(4)                 ax = fig.add_subplot(222)
(4)                 ax.xaxis.set_tick_params(tickdir='in', length=32, width=3)
(4)                 ax.yaxis.set_tick_params(tickdir='in', length=32, width=3)
(4)                 ax = fig.add_subplot(223)
(4)                 ax.xaxis.set_tick_params(tickdir='inout', length=32, width=3)
(4)                 ax.yaxis.set_tick_params(tickdir='inout', length=32, width=3)
(4)                 ax = fig.add_subplot(224)
(4)                 ax.xaxis.set_tick_params(tickdir='out', length=32, width=3)
(4)                 ax.yaxis.set_tick_params(tickdir='out', length=32, width=3)
(4)                 plt.tight_layout()
(4)                 expected = [
(8)                     [[0.091, 0.607], [0.433, 0.933]],
(8)                     [[0.579, 0.607], [0.922, 0.933]],
(8)                     [[0.091, 0.140], [0.433, 0.466]],
(8)                     [[0.579, 0.140], [0.922, 0.466]],
(4)                 ]
(4)                 for nn, ax in enumerate(fig.axes):
(8)                     assert_array_equal(np.round(ax.get_position().get_points(), 3),
(27)                                        expected[nn])
(0)             def add_offsetboxes(ax, size=10, margin=.1, color='black'):
(4)                 """
(4)                 Surround ax with OffsetBoxes
(4)                 """
(4)                 m, mp = margin, 1+margin
(4)                 anchor_points = [(-m, -m), (-m, .5), (-m, mp),
(21)                                 (mp, .5), (.5, mp), (mp, mp),
(21)                                 (.5, -m), (mp, -m), (.5, -m)]
(4)                 for point in anchor_points:
(8)                     da = DrawingArea(size, size)
(8)                     background = Rectangle((0, 0), width=size,
(31)                                           height=size,
(31)                                           facecolor=color,
(31)                                           edgecolor='None',
(31)                                           linewidth=0,
(31)                                           antialiased=False)
(8)                     da.add_artist(background)
(8)                     anchored_box = AnchoredOffsetbox(
(12)                        loc='center',
(12)                        child=da,
(12)                        pad=0.,
(12)                        frameon=False,
(12)                        bbox_to_anchor=point,
(12)                        bbox_transform=ax.transAxes,
(12)                        borderpad=0.)
(8)                     ax.add_artist(anchored_box)
```

```
169: (4)                return anchored_box
170: (0)            @image_comparison(['tight_layout_offsetboxes1', 'tight_layout_offsetboxes2'])
171: (0)            def test_tight_layout_offsetboxes():
172: (4)                rows = cols = 2
173: (4)                colors = ['red', 'blue', 'green', 'yellow']
174: (4)                x = y = [0, 1]
175: (4)                def _subplots():
176: (8)                    _, axs = plt.subplots(rows, cols)
177: (8)                    axs = axs.flat
178: (8)                    for ax, color in zip(axs, colors):
179: (12)                       ax.plot(x, y, color=color)
180: (12)                       add_offsetboxes(ax, 20, color=color)
181: (8)                    return axs
182: (4)                axs = _subplots()
183: (4)                plt.tight_layout()
184: (4)                axs = _subplots()
185: (4)                for ax in (axs[cols-1::rows]):
186: (8)                    for child in ax.get_children():
187: (12)                       if isinstance(child, AnchoredOffsetbox):
188: (16)                           child.set_visible(False)
189: (4)                plt.tight_layout()
190: (0)            def test_empty_layout():
191: (4)                """Test that tight layout doesn't cause an error when there are no
Axes."""
192: (4)                fig = plt.gcf()
193: (4)                fig.tight_layout()
194: (0)            @pytest.mark.parametrize("label", ["xlabel", "ylabel"])
195: (0)            def test_verybig_decorators(label):
196: (4)                """Test that no warning emitted when xlabel/ylabel too big."""
197: (4)                fig, ax = plt.subplots(figsize=(3, 2))
198: (4)                ax.set(**{label: 'a' * 100})
199: (0)            def test_big_decorators_horizontal():
200: (4)                """Test that doesn't warn when xlabel too big."""
201: (4)                fig, axs = plt.subplots(1, 2, figsize=(3, 2))
202: (4)                axs[0].set_xlabel('a' * 30)
203: (4)                axs[1].set_xlabel('b' * 30)
204: (0)            def test_big_decorators_vertical():
205: (4)                """Test that doesn't warn when ylabel too big."""
206: (4)                fig, axs = plt.subplots(2, 1, figsize=(3, 2))
207: (4)                axs[0].set_ylabel('a' * 20)
208: (4)                axs[1].set_ylabel('b' * 20)
209: (0)            def test_badsubplotgrid():
210: (4)                plt.subplot2grid((4, 5), (0, 0))
211: (4)                plt.subplot2grid((5, 5), (0, 3), colspan=3, rowspan=5)
212: (4)                with pytest.warns(UserWarning):
213: (8)                    plt.tight_layout()
214: (0)            def test_collapsed():
215: (4)                fig, ax = plt.subplots(tight_layout=True)
216: (4)                ax.set_xlim([0, 1])
217: (4)                ax.set_ylim([0, 1])
218: (4)                ax.annotate('BIG LONG STRING', xy=(1.25, 2), xytext=(10.5, 1.75),
219: (16)                           annotation_clip=False)
220: (4)                p1 = ax.get_position()
221: (4)                with pytest.warns(UserWarning):
222: (8)                    plt.tight_layout()
223: (8)                    p2 = ax.get_position()
224: (8)                    assert p1.width == p2.width
225: (4)                with pytest.warns(UserWarning):
226: (8)                    plt.tight_layout(rect=[0, 0, 0.8, 0.8])
227: (0)            def test_suptitle():
228: (4)                fig, ax = plt.subplots(tight_layout=True)
229: (4)                st = fig.suptitle("foo")
230: (4)                t = ax.set_title("bar")
231: (4)                fig.canvas.draw()
232: (4)                assert st.get_window_extent().y0 > t.get_window_extent().y1
233: (0)            @pytest.mark.backend("pdf")
234: (0)            def test_non_agg_renderer(monkeypatch, recwarn):
235: (4)                unpatched_init = mpl.backend_bases.RendererBase.__init__
236: (4)                def __init__(self, *args, **kwargs):
```

```
237: (8)                    assert isinstance(self, mpl.backends.backend_pdf.RendererPdf)
238: (8)                    unpatched_init(self, *args, **kwargs)
239: (4)                monkeypatch.setattr(mpl.backend_bases.RendererBase, "__init__", __init__)
240: (4)                fig, ax = plt.subplots()
241: (4)                fig.tight_layout()
242: (0)            def test_manual_colorbar():
243: (4)                fig, axes = plt.subplots(1, 2)
244: (4)                pts = axes[1].scatter([0, 1], [0, 1], c=[1, 5])
245: (4)                ax_rect = axes[1].get_position()
246: (4)                cax = fig.add_axes(
247: (8)                    [ax_rect.x1 + 0.005, ax_rect.y0, 0.015, ax_rect.height]
248: (4)                )
249: (4)                fig.colorbar(pts, cax=cax)
250: (4)                with pytest.warns(UserWarning, match="This figure includes Axes"):
251: (8)                    fig.tight_layout()
252: (0)            def test_clipped_to_axes():
253: (4)                arr = np.arange(100).reshape((10, 10))
254: (4)                fig = plt.figure(figsize=(6, 2))
255: (4)                ax1 = fig.add_subplot(131, projection='rectilinear')
256: (4)                ax2 = fig.add_subplot(132, projection='mollweide')
257: (4)                ax3 = fig.add_subplot(133, projection='polar')
258: (4)                for ax in (ax1, ax2, ax3):
259: (8)                    ax.grid(False)
260: (8)                    h, = ax.plot(arr[:, 0])
261: (8)                    m = ax.pcolor(arr)
262: (8)                    assert h._fully_clipped_to_axes()
263: (8)                    assert m._fully_clipped_to_axes()
264: (8)                    rect = Rectangle((0, 0), 0.5, 0.5, transform=ax.transAxes)
265: (8)                    h.set_clip_path(rect)
266: (8)                    m.set_clip_path(rect.get_path(), rect.get_transform())
267: (8)                    assert not h._fully_clipped_to_axes()
268: (8)                    assert not m._fully_clipped_to_axes()
269: (0)            def test_tight_pads():
270: (4)                fig, ax = plt.subplots()
271: (4)                with pytest.warns(PendingDeprecationWarning,
272: (22)                                  match='will be deprecated'):
273: (8)                    fig.set_tight_layout({'pad': 0.15})
274: (4)                fig.draw_without_rendering()
275: (0)            def test_tight_kwargs():
276: (4)                fig, ax = plt.subplots(tight_layout={'pad': 0.15})
277: (4)                fig.draw_without_rendering()
278: (0)            def test_tight_toggle():
279: (4)                fig, ax = plt.subplots()
280: (4)                with pytest.warns(PendingDeprecationWarning):
281: (8)                    fig.set_tight_layout(True)
282: (8)                    assert fig.get_tight_layout()
283: (8)                    fig.set_tight_layout(False)
284: (8)                    assert not fig.get_tight_layout()
285: (8)                    fig.set_tight_layout(True)
286: (8)                    assert fig.get_tight_layout()
```

File 224 - test_transforms.py:

```
1: (0)              import copy
2: (0)              import numpy as np
3: (0)              from numpy.testing import (assert_allclose, assert_almost_equal,
4: (27)                                        assert_array_equal, assert_array_almost_equal)
5: (0)              import pytest
6: (0)              from matplotlib import scale
7: (0)              import matplotlib.pyplot as plt
8: (0)              import matplotlib.patches as mpatches
9: (0)              import matplotlib.transforms as mtransforms
10: (0)             from matplotlib.transforms import Affine2D, Bbox, TransformedBbox
11: (0)             from matplotlib.path import Path
12: (0)             from matplotlib.testing.decorators import image_comparison,
check_figures_equal
13: (0)             class TestAffine2D:
```

```
(4)                 single_point = [1.0, 1.0]
(4)                 multiple_points = [[0.0, 2.0], [3.0, 3.0], [4.0, 0.0]]
(4)                 pivot = single_point
(4)                 def test_init(self):
(8)                     Affine2D([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
(8)                     Affine2D(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], int))
(8)                     Affine2D(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], float))
(4)                 def test_values(self):
(8)                     np.random.seed(19680801)
(8)                     values = np.random.random(6)
(8)                     assert_array_equal(Affine2D.from_values(*values).to_values(), values)
(4)                 def test_modify_inplace(self):
(8)                     trans = Affine2D()
(8)                     mtx = trans.get_matrix()
(8)                     mtx[0, 0] = 42
(8)                     assert_array_equal(trans.get_matrix(), [[42, 0, 0], [0, 1, 0], [0, 0,
1]])
(4)                 def test_clear(self):
(8)                     a = Affine2D(np.random.rand(3, 3) + 5)  # Anything non-identity.
(8)                     a.clear()
(8)                     assert_array_equal(a.get_matrix(), [[1, 0, 0], [0, 1, 0], [0, 0, 1]])
(4)                 def test_rotate(self):
(8)                     r_pi_2 = Affine2D().rotate(np.pi / 2)
(8)                     r90 = Affine2D().rotate_deg(90)
(8)                     assert_array_equal(r_pi_2.get_matrix(), r90.get_matrix())
(8)                     assert_array_almost_equal(r90.transform(self.single_point), [-1, 1])
(8)                     assert_array_almost_equal(r90.transform(self.multiple_points),
(34)                                                [[-2, 0], [-3, 3], [0, 4]])
(8)                     r_pi = Affine2D().rotate(np.pi)
(8)                     r180 = Affine2D().rotate_deg(180)
(8)                     assert_array_equal(r_pi.get_matrix(), r180.get_matrix())
(8)                     assert_array_almost_equal(r180.transform(self.single_point), [-1, -1])
(8)                     assert_array_almost_equal(r180.transform(self.multiple_points),
(34)                                                [[0, -2], [-3, -3], [-4, 0]])
(8)                     r_pi_3_2 = Affine2D().rotate(3 * np.pi / 2)
(8)                     r270 = Affine2D().rotate_deg(270)
(8)                     assert_array_equal(r_pi_3_2.get_matrix(), r270.get_matrix())
(8)                     assert_array_almost_equal(r270.transform(self.single_point), [1, -1])
(8)                     assert_array_almost_equal(r270.transform(self.multiple_points),
(34)                                                [[2, 0], [3, -3], [0, -4]])
(8)                     assert_array_equal((r90 + r90).get_matrix(), r180.get_matrix())
(8)                     assert_array_equal((r90 + r180).get_matrix(), r270.get_matrix())
(4)                 def test_rotate_around(self):
(8)                     r_pi_2 = Affine2D().rotate_around(*self.pivot, np.pi / 2)
(8)                     r90 = Affine2D().rotate_deg_around(*self.pivot, 90)
(8)                     assert_array_equal(r_pi_2.get_matrix(), r90.get_matrix())
(8)                     assert_array_almost_equal(r90.transform(self.single_point), [1, 1])
(8)                     assert_array_almost_equal(r90.transform(self.multiple_points),
(34)                                                [[0, 0], [-1, 3], [2, 4]])
(8)                     r_pi = Affine2D().rotate_around(*self.pivot, np.pi)
(8)                     r180 = Affine2D().rotate_deg_around(*self.pivot, 180)
(8)                     assert_array_equal(r_pi.get_matrix(), r180.get_matrix())
(8)                     assert_array_almost_equal(r180.transform(self.single_point), [1, 1])
(8)                     assert_array_almost_equal(r180.transform(self.multiple_points),
(34)                                                [[2, 0], [-1, -1], [-2, 2]])
(8)                     r_pi_3_2 = Affine2D().rotate_around(*self.pivot, 3 * np.pi / 2)
(8)                     r270 = Affine2D().rotate_deg_around(*self.pivot, 270)
(8)                     assert_array_equal(r_pi_3_2.get_matrix(), r270.get_matrix())
(8)                     assert_array_almost_equal(r270.transform(self.single_point), [1, 1])
(8)                     assert_array_almost_equal(r270.transform(self.multiple_points),
(34)                                                [[2, 2], [3, -1], [0, -2]])
(8)                     assert_array_almost_equal((r90 + r90).get_matrix(), r180.get_matrix())
(8)                     assert_array_almost_equal((r90 + r180).get_matrix(),
r270.get_matrix())
(4)                 def test_scale(self):
(8)                     sx = Affine2D().scale(3, 1)
(8)                     sy = Affine2D().scale(1, -2)
(8)                     trans = Affine2D().scale(3, -2)
(8)                     assert_array_equal((sx + sy).get_matrix(), trans.get_matrix())
```

```
 81: (8)                     assert_array_equal(trans.transform(self.single_point), [3, -2])
 82: (8)                     assert_array_equal(trans.transform(self.multiple_points),
 83: (27)                                       [[0, -4], [9, -6], [12, 0]])
 84: (4)                 def test_skew(self):
 85: (8)                     trans_rad = Affine2D().skew(np.pi / 8, np.pi / 12)
 86: (8)                     trans_deg = Affine2D().skew_deg(22.5, 15)
 87: (8)                     assert_array_equal(trans_rad.get_matrix(), trans_deg.get_matrix())
 88: (8)                     trans = Affine2D().skew_deg(26.5650512, 14.0362435)
 89: (8)                     assert_array_almost_equal(trans.transform(self.single_point), [1.5,
1.25])
 90: (8)                     assert_array_almost_equal(trans.transform(self.multiple_points),
 91: (34)                                              [[1, 2], [4.5, 3.75], [4, 1]])
 92: (4)                 def test_translate(self):
 93: (8)                     tx = Affine2D().translate(23, 0)
 94: (8)                     ty = Affine2D().translate(0, 42)
 95: (8)                     trans = Affine2D().translate(23, 42)
 96: (8)                     assert_array_equal((tx + ty).get_matrix(), trans.get_matrix())
 97: (8)                     assert_array_equal(trans.transform(self.single_point), [24, 43])
 98: (8)                     assert_array_equal(trans.transform(self.multiple_points),
 99: (27)                                       [[23, 44], [26, 45], [27, 42]])
 100: (4)                def test_rotate_plus_other(self):
 101: (8)                    trans = Affine2D().rotate_deg(90).rotate_deg_around(*self.pivot, 180)
 102: (8)                    trans_added = (Affine2D().rotate_deg(90) +
 103: (23)                                  Affine2D().rotate_deg_around(*self.pivot, 180))
 104: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
 105: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [3, 1])
 106: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
 107: (34)                                             [[4, 2], [5, -1], [2, -2]])
 108: (8)                    trans = Affine2D().rotate_deg(90).scale(3, -2)
 109: (8)                    trans_added = Affine2D().rotate_deg(90) + Affine2D().scale(3, -2)
 110: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
 111: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [-3,
-2])
 112: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
 113: (34)                                             [[-6, -0], [-9, -6], [0, -8]])
 114: (8)                    trans = (Affine2D().rotate_deg(90)
 115: (17)                            .skew_deg(26.5650512, 14.0362435))  # ~atan(0.5), ~atan(0.25)
 116: (8)                    trans_added = (Affine2D().rotate_deg(90) +
 117: (23)                                  Affine2D().skew_deg(26.5650512, 14.0362435))
 118: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
 119: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [-0.5,
0.75])
 120: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
 121: (34)                                             [[-2, -0.5], [-1.5, 2.25], [2, 4]])
 122: (8)                    trans = Affine2D().rotate_deg(90).translate(23, 42)
 123: (8)                    trans_added = Affine2D().rotate_deg(90) + Affine2D().translate(23, 42)
 124: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
 125: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [22,
43])
 126: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
 127: (34)                                             [[21, 42], [20, 45], [23, 46]])
 128: (4)                def test_rotate_around_plus_other(self):
 129: (8)                    trans = Affine2D().rotate_deg_around(*self.pivot, 90).rotate_deg(180)
 130: (8)                    trans_added = (Affine2D().rotate_deg_around(*self.pivot, 90) +
 131: (23)                                  Affine2D().rotate_deg(180))
 132: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
 133: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [-1,
-1])
 134: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
 135: (34)                                             [[0, 0], [1, -3], [-2, -4]])
 136: (8)                    trans = Affine2D().rotate_deg_around(*self.pivot, 90).scale(3, -2)
 137: (8)                    trans_added = (Affine2D().rotate_deg_around(*self.pivot, 90) +
 138: (23)                                  Affine2D().scale(3, -2))
 139: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
 140: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [3, -2])
 141: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
 142: (34)                                             [[0, 0], [-3, -6], [6, -8]])
 143: (8)                    trans = (Affine2D().rotate_deg_around(*self.pivot, 90)
 144: (17)                            .skew_deg(26.5650512, 14.0362435))  # ~atan(0.5), ~atan(0.25)
```

```
145: (8)                    trans_added = (Affine2D().rotate_deg_around(*self.pivot, 90) +
146: (23)                                  Affine2D().skew_deg(26.5650512, 14.0362435))
147: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
148: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [1.5,
1.25])
149: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
150: (34)                                             [[0, 0], [0.5, 2.75], [4, 4.5]])
151: (8)                    trans = Affine2D().rotate_deg_around(*self.pivot, 90).translate(23,
42)
152: (8)                    trans_added = (Affine2D().rotate_deg_around(*self.pivot, 90) +
153: (23)                                  Affine2D().translate(23, 42))
154: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
155: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [24,
43])
156: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
157: (34)                                             [[23, 42], [22, 45], [25, 46]])
158: (4)                def test_scale_plus_other(self):
159: (8)                    trans = Affine2D().scale(3, -2).rotate_deg(90)
160: (8)                    trans_added = Affine2D().scale(3, -2) + Affine2D().rotate_deg(90)
161: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
162: (8)                    assert_array_equal(trans.transform(self.single_point), [2, 3])
163: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
164: (34)                                             [[4, 0], [6, 9], [0, 12]])
165: (8)                    trans = Affine2D().scale(3, -2).rotate_deg_around(*self.pivot, 90)
166: (8)                    trans_added = (Affine2D().scale(3, -2) +
167: (23)                                  Affine2D().rotate_deg_around(*self.pivot, 90))
168: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
169: (8)                    assert_array_equal(trans.transform(self.single_point), [4, 3])
170: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
171: (34)                                             [[6, 0], [8, 9], [2, 12]])
172: (8)                    trans = (Affine2D().scale(3, -2)
173: (17)                            .skew_deg(26.5650512, 14.0362435))  # ~atan(0.5), ~atan(0.25)
174: (8)                    trans_added = (Affine2D().scale(3, -2) +
175: (23)                                  Affine2D().skew_deg(26.5650512, 14.0362435))
176: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
177: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [2,
-1.25])
178: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
179: (34)                                             [[-2, -4], [6, -3.75], [12, 3]])
180: (8)                    trans = Affine2D().scale(3, -2).translate(23, 42)
181: (8)                    trans_added = Affine2D().scale(3, -2) + Affine2D().translate(23, 42)
182: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
183: (8)                    assert_array_equal(trans.transform(self.single_point), [26, 40])
184: (8)                    assert_array_equal(trans.transform(self.multiple_points),
185: (27)                                       [[23, 38], [32, 36], [35, 42]])
186: (4)                def test_skew_plus_other(self):
187: (8)                    trans = Affine2D().skew_deg(26.5650512, 14.0362435).rotate_deg(90)
188: (8)                    trans_added = (Affine2D().skew_deg(26.5650512, 14.0362435) +
189: (23)                                  Affine2D().rotate_deg(90))
190: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
191: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [-1.25,
1.5])
192: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
193: (34)                                             [[-2, 1], [-3.75, 4.5], [-1, 4]])
194: (8)                    trans = (Affine2D().skew_deg(26.5650512, 14.0362435)
195: (17)                            .rotate_deg_around(*self.pivot, 90))
196: (8)                    trans_added = (Affine2D().skew_deg(26.5650512, 14.0362435) +
197: (23)                                  Affine2D().rotate_deg_around(*self.pivot, 90))
198: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
199: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [0.75,
1.5])
200: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
201: (34)                                             [[0, 1], [-1.75, 4.5], [1, 4]])
202: (8)                    trans = Affine2D().skew_deg(26.5650512, 14.0362435).scale(3, -2)
203: (8)                    trans_added = (Affine2D().skew_deg(26.5650512, 14.0362435) +
204: (23)                                  Affine2D().scale(3, -2))
205: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
206: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [4.5,
-2.5])
```

```
207: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
208: (34)                                             [[3, -4], [13.5, -7.5], [12, -2]])
209: (8)                    trans = Affine2D().skew_deg(26.5650512, 14.0362435).translate(23, 42)
210: (8)                    trans_added = (Affine2D().skew_deg(26.5650512, 14.0362435) +
211: (23)                                  Affine2D().translate(23, 42))
212: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
213: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [24.5,
43.25])
214: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
215: (34)                                             [[24, 44], [27.5, 45.75], [27, 43]])
216: (4)                def test_translate_plus_other(self):
217: (8)                    trans = Affine2D().translate(23, 42).rotate_deg(90)
218: (8)                    trans_added = Affine2D().translate(23, 42) + Affine2D().rotate_deg(90)
219: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
220: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [-43,
24])
221: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
222: (34)                                             [[-44, 23], [-45, 26], [-42, 27]])
223: (8)                    trans = Affine2D().translate(23, 42).rotate_deg_around(*self.pivot,
90)
224: (8)                    trans_added = (Affine2D().translate(23, 42) +
225: (23)                                  Affine2D().rotate_deg_around(*self.pivot, 90))
226: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
227: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [-41,
24])
228: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
229: (34)                                             [[-42, 23], [-43, 26], [-40, 27]])
230: (8)                    trans = Affine2D().translate(23, 42).scale(3, -2)
231: (8)                    trans_added = Affine2D().translate(23, 42) + Affine2D().scale(3, -2)
232: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
233: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [72,
-86])
234: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
235: (34)                                             [[69, -88], [78, -90], [81, -84]])
236: (8)                    trans = (Affine2D().translate(23, 42)
237: (17)                            .skew_deg(26.5650512, 14.0362435))  # ~atan(0.5), ~atan(0.25)
238: (8)                    trans_added = (Affine2D().translate(23, 42) +
239: (23)                                  Affine2D().skew_deg(26.5650512, 14.0362435))
240: (8)                    assert_array_equal(trans.get_matrix(), trans_added.get_matrix())
241: (8)                    assert_array_almost_equal(trans.transform(self.single_point), [45.5,
49])
242: (8)                    assert_array_almost_equal(trans.transform(self.multiple_points),
243: (34)                                             [[45, 49.75], [48.5, 51.5], [48, 48.75]])
244: (4)                def test_invalid_transform(self):
245: (8)                    t = mtransforms.Affine2D()
246: (8)                    with pytest.raises(ValueError):
247: (12)                       t.transform(1)
248: (8)                    with pytest.raises(ValueError):
249: (12)                       t.transform([[[1]]])
250: (8)                    with pytest.raises(RuntimeError):
251: (12)                       t.transform([])
252: (8)                    with pytest.raises(RuntimeError):
253: (12)                       t.transform([1])
254: (8)                    with pytest.raises(ValueError):
255: (12)                       t.transform([[1]])
256: (8)                    with pytest.raises(ValueError):
257: (12)                       t.transform([[1, 2, 3]])
258: (4)                def test_copy(self):
259: (8)                    a = mtransforms.Affine2D()
260: (8)                    b = mtransforms.Affine2D()
261: (8)                    s = a + b
262: (8)                    s.get_matrix()  # resolve it.
263: (8)                    s1 = copy.copy(s)
264: (8)                    assert not s._invalid and not s1._invalid
265: (8)                    a.translate(1, 2)
266: (8)                    assert s._invalid and s1._invalid
267: (8)                    assert (s1.get_matrix() == a.get_matrix()).all()
268: (8)                    s.get_matrix()  # resolve it.
269: (8)                    b1 = copy.copy(b)
```

```
270: (8)                    b1.translate(3, 4)
271: (8)                    assert not s._invalid
272: (8)                    assert_array_equal(s.get_matrix(), a.get_matrix())
273: (4)                def test_deepcopy(self):
274: (8)                    a = mtransforms.Affine2D()
275: (8)                    b = mtransforms.Affine2D()
276: (8)                    s = a + b
277: (8)                    s.get_matrix()  # resolve it.
278: (8)                    s1 = copy.deepcopy(s)
279: (8)                    assert not s._invalid and not s1._invalid
280: (8)                    a.translate(1, 2)
281: (8)                    assert s._invalid and not s1._invalid
282: (8)                    assert_array_equal(s1.get_matrix(),
mtransforms.Affine2D().get_matrix())
283: (8)                    s.get_matrix()  # resolve it.
284: (8)                    b1 = copy.deepcopy(b)
285: (8)                    b1.translate(3, 4)
286: (8)                    assert not s._invalid
287: (8)                    assert_array_equal(s.get_matrix(), a.get_matrix())
288: (0)            def test_non_affine_caching():
289: (4)                class AssertingNonAffineTransform(mtransforms.Transform):
290: (8)                    """
291: (8)                    This transform raises an assertion error when called when it
292: (8)                    shouldn't be and ``self.raise_on_transform`` is True.
293: (8)                    """
294: (8)                    input_dims = output_dims = 2
295: (8)                    is_affine = False
296: (8)                    def __init__(self, *args, **kwargs):
297: (12)                       super().__init__(*args, **kwargs)
298: (12)                       self.raise_on_transform = False
299: (12)                       self.underlying_transform = mtransforms.Affine2D().scale(10, 10)
300: (8)                    def transform_path_non_affine(self, path):
301: (12)                       assert not self.raise_on_transform, \
302: (16)                           'Invalidated affine part of transform unnecessarily.'
303: (12)                       return self.underlying_transform.transform_path(path)
304: (8)                    transform_path = transform_path_non_affine
305: (8)                    def transform_non_affine(self, path):
306: (12)                       assert not self.raise_on_transform, \
307: (16)                           'Invalidated affine part of transform unnecessarily.'
308: (12)                       return self.underlying_transform.transform(path)
309: (8)                    transform = transform_non_affine
310: (4)                my_trans = AssertingNonAffineTransform()
311: (4)                ax = plt.axes()
312: (4)                plt.plot(np.arange(10), transform=my_trans + ax.transData)
313: (4)                plt.draw()
314: (4)                my_trans.raise_on_transform = True
315: (4)                ax.transAxes.invalidate()
316: (4)                plt.draw()
317: (0)            def test_external_transform_api():
318: (4)                class ScaledBy:
319: (8)                    def __init__(self, scale_factor):
320: (12)                       self._scale_factor = scale_factor
321: (8)                    def _as_mpl_transform(self, axes):
322: (12)                       return (mtransforms.Affine2D().scale(self._scale_factor)
323: (20)                               + axes.transData)
324: (4)                ax = plt.axes()
325: (4)                line, = plt.plot(np.arange(10), transform=ScaledBy(10))
326: (4)                ax.set_xlim(0, 100)
327: (4)                ax.set_ylim(0, 100)
328: (4)                assert_allclose(line.get_transform()._a.get_matrix(),
329: (20)                               mtransforms.Affine2D().scale(10).get_matrix())
330: (0)            @image_comparison(['pre_transform_data'], remove_text=True, style='mpl20',
331: (18)                             tol=0.05)
332: (0)            def test_pre_transform_plotting():
333: (4)                ax = plt.axes()
334: (4)                times10 = mtransforms.Affine2D().scale(10)
335: (4)                ax.contourf(np.arange(48).reshape(6, 8), transform=times10 + ax.transData)
336: (4)                ax.pcolormesh(np.linspace(0, 4, 7),
337: (18)                             np.linspace(5.5, 8, 9),
```

```
(18)                            np.arange(48).reshape(8, 6),
(18)                            transform=times10 + ax.transData)
(4)                 ax.scatter(np.linspace(0, 10), np.linspace(10, 0),
(15)                           transform=times10 + ax.transData)
(4)                 x = np.linspace(8, 10, 20)
(4)                 y = np.linspace(1, 5, 20)
(4)                 u = 2*np.sin(x) + np.cos(y[:, np.newaxis])
(4)                 v = np.sin(x) - np.cos(y[:, np.newaxis])
(4)                 ax.streamplot(x, y, u, v, transform=times10 + ax.transData,
(18)                            linewidth=np.hypot(u, v))
(4)                 x, y = x[::3], y[::3]
(4)                 u, v = u[::3, ::3], v[::3, ::3]
(4)                 ax.quiver(x, y + 5, u, v, transform=times10 + ax.transData)
(4)                 ax.barbs(x - 3, y + 5, u**2, v**2, transform=times10 + ax.transData)
(0)             def test_contour_pre_transform_limits():
(4)                 ax = plt.axes()
(4)                 xs, ys = np.meshgrid(np.linspace(15, 20, 15), np.linspace(12.4, 12.5, 20))
(4)                 ax.contourf(xs, ys, np.log(xs * ys),
(16)                            transform=mtransforms.Affine2D().scale(0.1) + ax.transData)
(4)                 expected = np.array([[1.5, 1.24],
(25)                                     [2., 1.25]])
(4)                 assert_almost_equal(expected, ax.dataLim.get_points())
(0)             def test_pcolor_pre_transform_limits():
(4)                 ax = plt.axes()
(4)                 xs, ys = np.meshgrid(np.linspace(15, 20, 15), np.linspace(12.4, 12.5, 20))
(4)                 ax.pcolor(xs, ys, np.log(xs * ys)[:-1, :-1],
(14)                          transform=mtransforms.Affine2D().scale(0.1) + ax.transData)
(4)                 expected = np.array([[1.5, 1.24],
(25)                                     [2., 1.25]])
(4)                 assert_almost_equal(expected, ax.dataLim.get_points())
(0)             def test_pcolormesh_pre_transform_limits():
(4)                 ax = plt.axes()
(4)                 xs, ys = np.meshgrid(np.linspace(15, 20, 15), np.linspace(12.4, 12.5, 20))
(4)                 ax.pcolormesh(xs, ys, np.log(xs * ys)[:-1, :-1],
(18)                            transform=mtransforms.Affine2D().scale(0.1) + ax.transData)
(4)                 expected = np.array([[1.5, 1.24],
(25)                                     [2., 1.25]])
(4)                 assert_almost_equal(expected, ax.dataLim.get_points())
(0)             def test_pcolormesh_gouraud_nans():
(4)                 np.random.seed(19680801)
(4)                 values = np.linspace(0, 180, 3)
(4)                 radii = np.linspace(100, 1000, 10)
(4)                 z, y = np.meshgrid(values, radii)
(4)                 x = np.radians(np.random.rand(*z.shape) * 100)
(4)                 fig = plt.figure()
(4)                 ax = fig.add_subplot(111, projection="polar")
(4)                 ax.set_rlim(101, 1000)
(4)                 ax.pcolormesh(x, y, z, shading="gouraud")
(4)                 fig.canvas.draw()
(0)             def test_Affine2D_from_values():
(4)                 points = np.array([[0, 0],
(23)                                   [10, 20],
(23)                                   [-1, 0],
(23)                                   ])
(4)                 t = mtransforms.Affine2D.from_values(1, 0, 0, 0, 0, 0)
(4)                 actual = t.transform(points)
(4)                 expected = np.array([[0, 0], [10, 0], [-1, 0]])
(4)                 assert_almost_equal(actual, expected)
(4)                 t = mtransforms.Affine2D.from_values(0, 2, 0, 0, 0, 0)
(4)                 actual = t.transform(points)
(4)                 expected = np.array([[0, 0], [0, 20], [0, -2]])
(4)                 assert_almost_equal(actual, expected)
(4)                 t = mtransforms.Affine2D.from_values(0, 0, 3, 0, 0, 0)
(4)                 actual = t.transform(points)
(4)                 expected = np.array([[0, 0], [60, 0], [0, 0]])
(4)                 assert_almost_equal(actual, expected)
(4)                 t = mtransforms.Affine2D.from_values(0, 0, 0, 4, 0, 0)
(4)                 actual = t.transform(points)
(4)                 expected = np.array([[0, 0], [0, 80], [0, 0]])
```

```
407: (4)                assert_almost_equal(actual, expected)
408: (4)                t = mtransforms.Affine2D.from_values(0, 0, 0, 0, 5, 0)
409: (4)                actual = t.transform(points)
410: (4)                expected = np.array([[5, 0], [5, 0], [5, 0]])
411: (4)                assert_almost_equal(actual, expected)
412: (4)                t = mtransforms.Affine2D.from_values(0, 0, 0, 0, 0, 6)
413: (4)                actual = t.transform(points)
414: (4)                expected = np.array([[0, 6], [0, 6], [0, 6]])
415: (4)                assert_almost_equal(actual, expected)
416: (0)            def test_affine_inverted_invalidated():
417: (4)                point = [1.0, 1.0]
418: (4)                t = mtransforms.Affine2D()
419: (4)                assert_almost_equal(point, t.transform(t.inverted().transform(point)))
420: (4)                t.translate(1.0, 1.0).get_matrix()
421: (4)                assert_almost_equal(point, t.transform(t.inverted().transform(point)))
422: (0)            def test_clipping_of_log():
423: (4)                path = Path._create_closed([(0.2, -99), (0.4, -99), (0.4, 20), (0.2, 20)])
424: (4)                trans = mtransforms.BlendedGenericTransform(
425: (8)                    mtransforms.Affine2D(), scale.LogTransform(10, 'clip'))
426: (4)                tpath = trans.transform_path_non_affine(path)
427: (4)                result = tpath.iter_segments(trans.get_affine(),
428: (33)                                            clip=(0, 0, 100, 100),
429: (33)                                            simplify=False)
430: (4)                tpoints, tcodes = zip(*result)
431: (4)                assert_allclose(tcodes, path.codes[:-1])  # No longer closed.
432: (0)            class NonAffineForTest(mtransforms.Transform):
433: (4)                """
434: (4)                A class which looks like a non affine transform, but does whatever
435: (4)                the given transform does (even if it is affine). This is very useful
436: (4)                for testing NonAffine behaviour with a simple Affine transform.
437: (4)                """
438: (4)                is_affine = False
439: (4)                output_dims = 2
440: (4)                input_dims = 2
441: (4)                def __init__(self, real_trans, *args, **kwargs):
442: (8)                    self.real_trans = real_trans
443: (8)                    super().__init__(*args, **kwargs)
444: (4)                def transform_non_affine(self, values):
445: (8)                    return self.real_trans.transform(values)
446: (4)                def transform_path_non_affine(self, path):
447: (8)                    return self.real_trans.transform_path(path)
448: (0)            class TestBasicTransform:
449: (4)                def setup_method(self):
450: (8)                    self.ta1 = mtransforms.Affine2D(shorthand_name='ta1').rotate(np.pi /
2)
451: (8)                    self.ta2 = mtransforms.Affine2D(shorthand_name='ta2').translate(10, 0)
452: (8)                    self.ta3 = mtransforms.Affine2D(shorthand_name='ta3').scale(1, 2)
453: (8)                    self.tn1 = NonAffineForTest(mtransforms.Affine2D().translate(1, 2),
454: (36)                                               shorthand_name='tn1')
455: (8)                    self.tn2 = NonAffineForTest(mtransforms.Affine2D().translate(1, 2),
456: (36)                                               shorthand_name='tn2')
457: (8)                    self.tn3 = NonAffineForTest(mtransforms.Affine2D().translate(1, 2),
458: (36)                                               shorthand_name='tn3')
459: (8)                    self.stack1 = (self.ta1 + (self.tn1 + self.ta2)) + self.ta3
460: (8)                    self.stack2 = self.ta1 + self.tn1 + self.ta2 + self.ta3
461: (8)                    self.stack2_subset = self.tn1 + self.ta2 + self.ta3
462: (4)                def test_transform_depth(self):
463: (8)                    assert self.stack1.depth == 4
464: (8)                    assert self.stack2.depth == 4
465: (8)                    assert self.stack2_subset.depth == 3
466: (4)                def test_left_to_right_iteration(self):
467: (8)                    stack3 = (self.ta1 + (self.tn1 + (self.ta2 + self.tn2))) + self.ta3
468: (8)                    target_transforms = [stack3,
469: (29)                                        (self.tn1 + (self.ta2 + self.tn2)) + self.ta3,
470: (29)                                        (self.ta2 + self.tn2) + self.ta3,
471: (29)                                        self.tn2 + self.ta3,
472: (29)                                        self.ta3,
473: (29)                                        ]
474: (8)                    r = [rh for _, rh in stack3._iter_break_from_left_to_right()]
```

```
(8)                     assert len(r) == len(target_transforms)
(8)                     for target_stack, stack in zip(target_transforms, r):
(12)                        assert target_stack == stack
(4)                 def test_transform_shortcuts(self):
(8)                     assert self.stack1 - self.stack2_subset == self.ta1
(8)                     assert self.stack2 - self.stack2_subset == self.ta1
(8)                     assert self.stack2_subset - self.stack2 == self.ta1.inverted()
(8)                     assert (self.stack2_subset - self.stack2).depth == 1
(8)                     with pytest.raises(ValueError):
(12)                        self.stack1 - self.stack2
(8)                     aff1 = self.ta1 + (self.ta2 + self.ta3)
(8)                     aff2 = self.ta2 + self.ta3
(8)                     assert aff1 - aff2 == self.ta1
(8)                     assert aff1 - self.ta2 == aff1 + self.ta2.inverted()
(8)                     assert self.stack1 - self.ta3 == self.ta1 + (self.tn1 + self.ta2)
(8)                     assert self.stack2 - self.ta3 == self.ta1 + self.tn1 + self.ta2
(8)                     assert ((self.ta2 + self.ta3) - self.ta3 + self.ta3 ==
(16)                            self.ta2 + self.ta3)
(4)                 def test_contains_branch(self):
(8)                     r1 = (self.ta2 + self.ta1)
(8)                     r2 = (self.ta2 + self.ta1)
(8)                     assert r1 == r2
(8)                     assert r1 != self.ta1
(8)                     assert r1.contains_branch(r2)
(8)                     assert r1.contains_branch(self.ta1)
(8)                     assert not r1.contains_branch(self.ta2)
(8)                     assert not r1.contains_branch(self.ta2 + self.ta2)
(8)                     assert r1 == r2
(8)                     assert self.stack1.contains_branch(self.ta3)
(8)                     assert self.stack2.contains_branch(self.ta3)
(8)                     assert self.stack1.contains_branch(self.stack2_subset)
(8)                     assert self.stack2.contains_branch(self.stack2_subset)
(8)                     assert not self.stack2_subset.contains_branch(self.stack1)
(8)                     assert not self.stack2_subset.contains_branch(self.stack2)
(8)                     assert self.stack1.contains_branch(self.ta2 + self.ta3)
(8)                     assert self.stack2.contains_branch(self.ta2 + self.ta3)
(8)                     assert not self.stack1.contains_branch(self.tn1 + self.ta2)
(8)                     blend = mtransforms.BlendedGenericTransform(self.tn2, self.stack2)
(8)                     x, y = blend.contains_branch_seperately(self.stack2_subset)
(8)                     stack_blend = self.tn3 + blend
(8)                     sx, sy = stack_blend.contains_branch_seperately(self.stack2_subset)
(8)                     assert x is sx is False
(8)                     assert y is sy is True
(4)                 def test_affine_simplification(self):
(8)                     points = np.array([[0, 0], [10, 20], [np.nan, 1], [-1, 0]],
(26)                                      dtype=np.float64)
(8)                     na_pts = self.stack1.transform_non_affine(points)
(8)                     all_pts = self.stack1.transform(points)
(8)                     na_expected = np.array([[1., 2.], [-19., 12.],
(32)                                            [np.nan, np.nan], [1., 1.]], dtype=np.float64)
(8)                     all_expected = np.array([[11., 4.], [-9., 24.],
(33)                                             [np.nan, np.nan], [11., 2.]],
(32)                                            dtype=np.float64)
(8)                     assert_array_almost_equal(na_pts, na_expected)
(8)                     assert_array_almost_equal(all_pts, all_expected)
(8)                     assert_array_almost_equal(self.stack1.transform_affine(na_pts),
(34)                                              all_expected)
(8)                     assert_array_almost_equal(self.stack1.get_affine().transform(na_pts),
(34)                                              all_expected)
(8)                     expected_result = (self.ta2 + self.ta3).get_matrix()
(8)                     result = self.stack1.get_affine().get_matrix()
(8)                     assert_array_equal(expected_result, result)
(8)                     result = self.stack2.get_affine().get_matrix()
(8)                     assert_array_equal(expected_result, result)
(0)             class TestTransformPlotInterface:
(4)                 def test_line_extent_axes_coords(self):
(8)                     ax = plt.axes()
(8)                     ax.plot([0.1, 1.2, 0.8], [0.9, 0.5, 0.8], transform=ax.transAxes)
(8)                     assert_array_equal(ax.dataLim.get_points(),
```

```
544: (27)                                    np.array([[np.inf, np.inf],
545: (37)                                              [-np.inf, -np.inf]]))
546: (4)                def test_line_extent_data_coords(self):
547: (8)                    ax = plt.axes()
548: (8)                    ax.plot([0.1, 1.2, 0.8], [0.9, 0.5, 0.8], transform=ax.transData)
549: (8)                    assert_array_equal(ax.dataLim.get_points(),
550: (27)                                    np.array([[0.1,  0.5], [1.2,  0.9]]))
551: (4)                def test_line_extent_compound_coords1(self):
552: (8)                    ax = plt.axes()
553: (8)                    trans = mtransforms.blended_transform_factory(ax.transAxes,
554: (54)                                                                 ax.transData)
555: (8)                    ax.plot([0.1, 1.2, 0.8], [35, -5, 18], transform=trans)
556: (8)                    assert_array_equal(ax.dataLim.get_points(),
557: (27)                                    np.array([[np.inf, -5.],
558: (37)                                              [-np.inf, 35.]]))
559: (4)                def test_line_extent_predata_transform_coords(self):
560: (8)                    ax = plt.axes()
561: (8)                    trans = mtransforms.Affine2D().scale(10) + ax.transData
562: (8)                    ax.plot([0.1, 1.2, 0.8], [35, -5, 18], transform=trans)
563: (8)                    assert_array_equal(ax.dataLim.get_points(),
564: (27)                                    np.array([[1., -50.], [12., 350.]]))
565: (4)                def test_line_extent_compound_coords2(self):
566: (8)                    ax = plt.axes()
567: (8)                    trans = mtransforms.blended_transform_factory(
568: (12)                       ax.transAxes, mtransforms.Affine2D().scale(10) + ax.transData)
569: (8)                    ax.plot([0.1, 1.2, 0.8], [35, -5, 18], transform=trans)
570: (8)                    assert_array_equal(ax.dataLim.get_points(),
571: (27)                                    np.array([[np.inf, -50.], [-np.inf, 350.]]))
572: (4)                def test_line_extents_affine(self):
573: (8)                    ax = plt.axes()
574: (8)                    offset = mtransforms.Affine2D().translate(10, 10)
575: (8)                    plt.plot(np.arange(10), transform=offset + ax.transData)
576: (8)                    expected_data_lim = np.array([[0., 0.], [9.,  9.]]) + 10
577: (8)                    assert_array_almost_equal(ax.dataLim.get_points(), expected_data_lim)
578: (4)                def test_line_extents_non_affine(self):
579: (8)                    ax = plt.axes()
580: (8)                    offset = mtransforms.Affine2D().translate(10, 10)
581: (8)                    na_offset = NonAffineForTest(mtransforms.Affine2D().translate(10, 10))
582: (8)                    plt.plot(np.arange(10), transform=offset + na_offset + ax.transData)
583: (8)                    expected_data_lim = np.array([[0., 0.], [9.,  9.]]) + 20
584: (8)                    assert_array_almost_equal(ax.dataLim.get_points(), expected_data_lim)
585: (4)                def test_pathc_extents_non_affine(self):
586: (8)                    ax = plt.axes()
587: (8)                    offset = mtransforms.Affine2D().translate(10, 10)
588: (8)                    na_offset = NonAffineForTest(mtransforms.Affine2D().translate(10, 10))
589: (8)                    pth = Path([[0, 0], [0, 10], [10, 10], [10, 0]])
590: (8)                    patch = mpatches.PathPatch(pth,
591: (35)                                              transform=offset + na_offset +
ax.transData)
592: (8)                    ax.add_patch(patch)
593: (8)                    expected_data_lim = np.array([[0., 0.], [10.,  10.]]) + 20
594: (8)                    assert_array_almost_equal(ax.dataLim.get_points(), expected_data_lim)
595: (4)                def test_pathc_extents_affine(self):
596: (8)                    ax = plt.axes()
597: (8)                    offset = mtransforms.Affine2D().translate(10, 10)
598: (8)                    pth = Path([[0, 0], [0, 10], [10, 10], [10, 0]])
599: (8)                    patch = mpatches.PathPatch(pth, transform=offset + ax.transData)
600: (8)                    ax.add_patch(patch)
601: (8)                    expected_data_lim = np.array([[0., 0.], [10.,  10.]]) + 10
602: (8)                    assert_array_almost_equal(ax.dataLim.get_points(), expected_data_lim)
603: (4)                def test_line_extents_for_non_affine_transData(self):
604: (8)                    ax = plt.axes(projection='polar')
605: (8)                    offset = mtransforms.Affine2D().translate(0, 10)
606: (8)                    plt.plot(np.arange(10), transform=offset + ax.transData)
607: (8)                    expected_data_lim = np.array([[0., 0.], [9.,  9.]]) + [0, 10]
608: (8)                    assert_array_almost_equal(ax.dataLim.get_points(), expected_data_lim)
609: (0)            def assert_bbox_eq(bbox1, bbox2):
610: (4)                assert_array_equal(bbox1.bounds, bbox2.bounds)
611: (0)            def test_bbox_frozen_copies_minpos():
```

```
(4)                 bbox = mtransforms.Bbox.from_extents(0.0, 0.0, 1.0, 1.0, minpos=1.0)
(4)                 frozen = bbox.frozen()
(4)                 assert_array_equal(frozen.minpos, bbox.minpos)
(0)             def test_bbox_intersection():
(4)                 bbox_from_ext = mtransforms.Bbox.from_extents
(4)                 inter = mtransforms.Bbox.intersection
(4)                 r1 = bbox_from_ext(0, 0, 1, 1)
(4)                 r2 = bbox_from_ext(0.5, 0.5, 1.5, 1.5)
(4)                 r3 = bbox_from_ext(0.5, 0, 0.75, 0.75)
(4)                 r4 = bbox_from_ext(0.5, 1.5, 1, 2.5)
(4)                 r5 = bbox_from_ext(1, 1, 2, 2)
(4)                 assert_bbox_eq(inter(r1, r1), r1)
(4)                 assert_bbox_eq(inter(r1, r2), bbox_from_ext(0.5, 0.5, 1, 1))
(4)                 assert_bbox_eq(inter(r1, r3), r3)
(4)                 assert inter(r1, r4) is None
(4)                 assert_bbox_eq(inter(r1, r5), bbox_from_ext(1, 1, 1, 1))
(0)             def test_bbox_as_strings():
(4)                 b = mtransforms.Bbox([[.5, 0], [.75, .75]])
(4)                 assert_bbox_eq(b, eval(repr(b), {'Bbox': mtransforms.Bbox}))
(4)                 asdict = eval(str(b), {'Bbox': dict})
(4)                 for k, v in asdict.items():
(8)                     assert getattr(b, k) == v
(4)                 fmt = '.1f'
(4)                 asdict = eval(format(b, fmt), {'Bbox': dict})
(4)                 for k, v in asdict.items():
(8)                     assert eval(format(getattr(b, k), fmt)) == v
(0)             def test_str_transform():
(4)                 assert str(plt.subplot(projection="polar").transData) == """\
(0)             CompositeGenericTransform(
(4)                 CompositeGenericTransform(
(8)                     CompositeGenericTransform(
(12)                        TransformWrapper(
(16)                            BlendedAffine2D(
(20)                                IdentityTransform(),
(20)                                IdentityTransform())),
(12)                        CompositeAffine2D(
(16)                            Affine2D().scale(1.0),
(16)                            Affine2D().scale(1.0))),
(8)                     PolarTransform(
(12)                        PolarAxes(0.125,0.1;0.775x0.8),
(12)                        use_rmin=True,
(12)                        apply_theta_transforms=False)),
(4)                 CompositeGenericTransform(
(8)                     CompositeGenericTransform(
(12)                        PolarAffine(
(16)                            TransformWrapper(
(20)                                BlendedAffine2D(
(24)                                    IdentityTransform(),
(24)                                    IdentityTransform())),
(16)                            LockableBbox(
(20)                                Bbox(x0=0.0, y0=0.0, x1=6.283185307179586, y1=1.0),
(20)                                [[-- --]
(21)                                 [-- --]])),
(12)                        BboxTransformFrom(
(16)                            _WedgeBbox(
(20)                                (0.5, 0.5),
(20)                                TransformedBbox(
(24)                                    Bbox(x0=0.0, y0=0.0, x1=6.283185307179586, y1=1.0),
(24)                                    CompositeAffine2D(
(28)                                        Affine2D().scale(1.0),
(28)                                        Affine2D().scale(1.0))),
(20)                                LockableBbox(
(24)                                    Bbox(x0=0.0, y0=0.0, x1=6.283185307179586, y1=1.0),
(24)                                    [[-- --]
(25)                                     [-- --]]))))),
(8)                     BboxTransformTo(
(12)                        TransformedBbox(
(16)                            Bbox(x0=0.125, y0=0.09999999999999998, x1=0.9, y1=0.9),
(16)                            BboxTransformTo(
```

```
(20)                                TransformedBbox(
(24)                                    Bbox(x0=0.0, y0=0.0, x1=8.0, y1=6.0),
(24)                                    Affine2D().scale(80.0)))))))"""
(0)             def test_transform_single_point():
(4)                 t = mtransforms.Affine2D()
(4)                 r = t.transform_affine((1, 1))
(4)                 assert r.shape == (2,)
(0)             def test_log_transform():
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_yscale('log')
(4)                 ax.transData.transform((1, 1))
(0)             def test_nan_overlap():
(4)                 a = mtransforms.Bbox([[0, 0], [1, 1]])
(4)                 b = mtransforms.Bbox([[0, 0], [1, np.nan]])
(4)                 assert not a.overlaps(b)
(0)             def test_transform_angles():
(4)                 t = mtransforms.Affine2D()  # Identity transform
(4)                 angles = np.array([20, 45, 60])
(4)                 points = np.array([[0, 0], [1, 1], [2, 2]])
(4)                 new_angles = t.transform_angles(angles, points)
(4)                 assert_array_almost_equal(angles, new_angles)
(4)                 with pytest.raises(ValueError):
(8)                     t.transform_angles(angles, points[0:2, 0:1])
(4)                 with pytest.raises(ValueError):
(8)                     t.transform_angles(angles, points[0:2, :])
(0)             def test_nonsingular():
(4)                 zero_expansion = np.array([-0.001, 0.001])
(4)                 cases = [(0, np.nan), (0, 0), (0, 7.9e-317)]
(4)                 for args in cases:
(8)                     out = np.array(mtransforms.nonsingular(*args))
(8)                     assert_array_equal(out, zero_expansion)
(0)             def test_transformed_path():
(4)                 points = [(0, 0), (1, 0), (1, 1), (0, 1)]
(4)                 path = Path(points, closed=True)
(4)                 trans = mtransforms.Affine2D()
(4)                 trans_path = mtransforms.TransformedPath(path, trans)
(4)                 assert_allclose(trans_path.get_fully_transformed_path().vertices, points)
(4)                 r2 = 1 / np.sqrt(2)
(4)                 trans.rotate(np.pi / 4)
(4)                 assert_allclose(trans_path.get_fully_transformed_path().vertices,
(20)                                [(0, 0), (r2, r2), (0, 2 * r2), (-r2, r2)],
(20)                                atol=1e-15)
(4)                 path.points = [(0, 0)] * 4
(4)                 assert_allclose(trans_path.get_fully_transformed_path().vertices,
(20)                                [(0, 0), (r2, r2), (0, 2 * r2), (-r2, r2)],
(20)                                atol=1e-15)
(0)             def test_transformed_patch_path():
(4)                 trans = mtransforms.Affine2D()
(4)                 patch = mpatches.Wedge((0, 0), 1, 45, 135, transform=trans)
(4)                 tpatch = mtransforms.TransformedPatchPath(patch)
(4)                 points = tpatch.get_fully_transformed_path().vertices
(4)                 trans.scale(2)
(4)                 assert_allclose(tpatch.get_fully_transformed_path().vertices, points * 2)
(4)                 patch.set_radius(0.5)
(4)                 assert_allclose(tpatch.get_fully_transformed_path().vertices, points)
(0)             @pytest.mark.parametrize('locked_element', ['x0', 'y0', 'x1', 'y1'])
(0)             def test_lockable_bbox(locked_element):
(4)                 other_elements = ['x0', 'y0', 'x1', 'y1']
(4)                 other_elements.remove(locked_element)
(4)                 orig = mtransforms.Bbox.unit()
(4)                 locked = mtransforms.LockableBbox(orig, **{locked_element: 2})
(4)                 assert getattr(locked, locked_element) == 2
(4)                 assert getattr(locked, 'locked_' + locked_element) == 2
(4)                 for elem in other_elements:
(8)                     assert getattr(locked, elem) == getattr(orig, elem)
(4)                 orig.set_points(orig.get_points() + 10)
(4)                 assert getattr(locked, locked_element) == 2
(4)                 assert getattr(locked, 'locked_' + locked_element) == 2
(4)                 for elem in other_elements:
```

```
750: (8)                    assert getattr(locked, elem) == getattr(orig, elem)
751: (4)                setattr(locked, 'locked_' + locked_element, None)
752: (4)                assert getattr(locked, 'locked_' + locked_element) is None
753: (4)                assert np.all(orig.get_points() == locked.get_points())
754: (4)                setattr(locked, 'locked_' + locked_element, 3)
755: (4)                assert getattr(locked, locked_element) == 3
756: (4)                assert getattr(locked, 'locked_' + locked_element) == 3
757: (4)                for elem in other_elements:
758: (8)                    assert getattr(locked, elem) == getattr(orig, elem)
759: (0)            def test_transformwrapper():
760: (4)                t = mtransforms.TransformWrapper(mtransforms.Affine2D())
761: (4)                with pytest.raises(ValueError, match=(
762: (12)                       r"The input and output dims of the new child \(1, 1\) "
763: (12)                       r"do not match those of current child \(2, 2\)")):
764: (8)                    t.set(scale.LogTransform(10))
765: (0)            @check_figures_equal(extensions=["png"])
766: (0)            def test_scale_swapping(fig_test, fig_ref):
767: (4)                np.random.seed(19680801)
768: (4)                samples = np.random.normal(size=10)
769: (4)                x = np.linspace(-5, 5, 10)
770: (4)                for fig, log_state in zip([fig_test, fig_ref], [True, False]):
771: (8)                    ax = fig.subplots()
772: (8)                    ax.hist(samples, log=log_state, density=True)
773: (8)                    ax.plot(x, np.exp(-(x**2) / 2) / np.sqrt(2 * np.pi))
774: (8)                    fig.canvas.draw()
775: (8)                    ax.set_yscale('linear')
776: (0)            def test_offset_copy_errors():
777: (4)                with pytest.raises(ValueError,
778: (23)                                   match="'fontsize' is not a valid value for units;"
779: (29)                                         " supported values are 'dots', 'points',
'inches'"):
780: (8)                    mtransforms.offset_copy(None, units='fontsize')
781: (4)                with pytest.raises(ValueError,
782: (23)                                   match='For units of inches or points a fig kwarg is
needed'):
783: (8)                    mtransforms.offset_copy(None, units='inches')
784: (0)            def test_transformedbbox_contains():
785: (4)                bb = TransformedBbox(Bbox.unit(), Affine2D().rotate_deg(30))
786: (4)                assert bb.contains(.8, .5)
787: (4)                assert bb.contains(-.4, .85)
788: (4)                assert not bb.contains(.9, .5)
789: (4)                bb = TransformedBbox(Bbox.unit(), Affine2D().translate(.25, .5))
790: (4)                assert bb.contains(1.25, 1.5)
791: (4)                assert not bb.fully_contains(1.25, 1.5)
792: (4)                assert not bb.fully_contains(.1, .1)
793: (0)            def test_interval_contains():
794: (4)                assert mtransforms.interval_contains((0, 1), 0.5)
795: (4)                assert mtransforms.interval_contains((0, 1), 0)
796: (4)                assert mtransforms.interval_contains((0, 1), 1)
797: (4)                assert not mtransforms.interval_contains((0, 1), -1)
798: (4)                assert not mtransforms.interval_contains((0, 1), 2)
799: (4)                assert mtransforms.interval_contains((1, 0), 0.5)
800: (0)            def test_interval_contains_open():
801: (4)                assert mtransforms.interval_contains_open((0, 1), 0.5)
802: (4)                assert not mtransforms.interval_contains_open((0, 1), 0)
803: (4)                assert not mtransforms.interval_contains_open((0, 1), 1)
804: (4)                assert not mtransforms.interval_contains_open((0, 1), -1)
805: (4)                assert not mtransforms.interval_contains_open((0, 1), 2)
806: (4)                assert mtransforms.interval_contains_open((1, 0), 0.5)

----------------------------------------

File 225 - test_triangulation.py:

1: (0)              import numpy as np
2: (0)              from numpy.testing import (
3: (4)                  assert_array_equal, assert_array_almost_equal, assert_array_less)
4: (0)              import numpy.ma.testutils as matest
5: (0)              import pytest
```

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.tri as mtri
from matplotlib.path import Path
from matplotlib.testing.decorators import image_comparison, check_figures_equal
class TestTriangulationParams:
    x = [-1, 0, 1, 0]
    y = [0, -1, 0, 1]
    triangles = [[0, 1, 2], [0, 2, 3]]
    mask = [False, True]
    @pytest.mark.parametrize('args, kwargs, expected', [
        ([x, y], {}, [x, y, None, None]),
        ([x, y, triangles], {}, [x, y, triangles, None]),
        ([x, y], dict(triangles=triangles), [x, y, triangles, None]),
        ([x, y], dict(mask=mask), [x, y, None, mask]),
        ([x, y, triangles], dict(mask=mask), [x, y, triangles, mask]),
        ([x, y], dict(triangles=triangles, mask=mask), [x, y, triangles, mask])
    ])
    def test_extract_triangulation_params(self, args, kwargs, expected):
        other_args = [1, 2]
        other_kwargs = {'a': 3, 'b': '4'}
        x_, y_, triangles_, mask_, args_, kwargs_ = \
            mtri.Triangulation._extract_triangulation_params(
                args + other_args, {**kwargs, **other_kwargs})
        x, y, triangles, mask = expected
        assert x_ is x
        assert y_ is y
        assert_array_equal(triangles_, triangles)
        assert mask_ is mask
        assert args_ == other_args
        assert kwargs_ == other_kwargs
def test_extract_triangulation_positional_mask():
    mask = [True]
    args = [[0, 2, 1], [0, 0, 1], [[0, 1, 2]], mask]
    x_, y_, triangles_, mask_, args_, kwargs_ = \
        mtri.Triangulation._extract_triangulation_params(args, {})
    assert mask_ is None
    assert args_ == [mask]
def test_triangulation_init():
    x = [-1, 0, 1, 0]
    y = [0, -1, 0, 1]
    with pytest.raises(ValueError, match="x and y must be equal-length"):
        mtri.Triangulation(x, [1, 2])
    with pytest.raises(
            ValueError,
            match=r"triangles must be a \(N, 3\) int array, but found shape "
                  r"\(3,\)"):
        mtri.Triangulation(x, y, [0, 1, 2])
    with pytest.raises(
            ValueError,
            match=r"triangles must be a \(N, 3\) int array, not 'other'"):
        mtri.Triangulation(x, y, 'other')
    with pytest.raises(ValueError, match="found value 99"):
        mtri.Triangulation(x, y, [[0, 1, 99]])
    with pytest.raises(ValueError, match="found value -1"):
        mtri.Triangulation(x, y, [[0, 1, -1]])
def test_triangulation_set_mask():
    x = [-1, 0, 1, 0]
    y = [0, -1, 0, 1]
    triangles = [[0, 1, 2], [2, 3, 0]]
    triang = mtri.Triangulation(x, y, triangles)
    assert_array_equal(triang.neighbors, [[-1, -1, 1], [-1, -1, 0]])
    triang.set_mask([False, True])
    assert_array_equal(triang.mask, [False, True])
    triang.set_mask(None)
    assert triang.mask is None
    msg = r"mask array must have same length as triangles array"
```

```
 73: (4)                for mask in ([False, True, False], [False], [True], False, True):
 74: (8)                    with pytest.raises(ValueError, match=msg):
 75: (12)                       triang.set_mask(mask)
 76: (0)            def test_delaunay():
 77: (4)                nx = 5
 78: (4)                ny = 4
 79: (4)                x, y = np.meshgrid(np.linspace(0.0, 1.0, nx), np.linspace(0.0, 1.0, ny))
 80: (4)                x = x.ravel()
 81: (4)                y = y.ravel()
 82: (4)                npoints = nx*ny
 83: (4)                ntriangles = 2 * (nx-1) * (ny-1)
 84: (4)                nedges = 3*nx*ny - 2*nx - 2*ny + 1
 85: (4)                triang = mtri.Triangulation(x, y)
 86: (4)                assert_array_almost_equal(triang.x, x)
 87: (4)                assert_array_almost_equal(triang.y, y)
 88: (4)                assert len(triang.triangles) == ntriangles
 89: (4)                assert np.min(triang.triangles) == 0
 90: (4)                assert np.max(triang.triangles) == npoints-1
 91: (4)                assert len(triang.edges) == nedges
 92: (4)                assert np.min(triang.edges) == 0
 93: (4)                assert np.max(triang.edges) == npoints-1
 94: (4)                neighbors = triang.neighbors
 95: (4)                triang._neighbors = None
 96: (4)                assert_array_equal(triang.neighbors, neighbors)
 97: (4)                assert_array_equal(np.unique(triang.triangles), np.arange(npoints))
 98: (0)            def test_delaunay_duplicate_points():
 99: (4)                npoints = 10
 100: (4)               duplicate = 7
 101: (4)               duplicate_of = 3
 102: (4)               np.random.seed(23)
 103: (4)               x = np.random.random(npoints)
 104: (4)               y = np.random.random(npoints)
 105: (4)               x[duplicate] = x[duplicate_of]
 106: (4)               y[duplicate] = y[duplicate_of]
 107: (4)               triang = mtri.Triangulation(x, y)
 108: (4)               assert_array_equal(np.unique(triang.triangles),
 109: (23)                                 np.delete(np.arange(npoints), duplicate))
 110: (0)           def test_delaunay_points_in_line():
 111: (4)               x = np.linspace(0.0, 10.0, 11)
 112: (4)               y = np.linspace(0.0, 10.0, 11)
 113: (4)               with pytest.raises(RuntimeError):
 114: (8)                   mtri.Triangulation(x, y)
 115: (4)               x = np.append(x, 2.0)
 116: (4)               y = np.append(y, 8.0)
 117: (4)               mtri.Triangulation(x, y)
 118: (0)           @pytest.mark.parametrize('x, y', [
 119: (4)               ([], []),
 120: (4)               ([1], [5]),
 121: (4)               ([1, 2], [5, 6]),
 122: (4)               ([1, 2, 1], [5, 6, 5]),
 123: (4)               ([1, 2, 2], [5, 6, 6]),
 124: (4)               ([1, 1, 1, 2, 1, 2], [5, 5, 5, 6, 5, 6]),
 125: (0)           ])
 126: (0)           def test_delaunay_insufficient_points(x, y):
 127: (4)               with pytest.raises(ValueError):
 128: (8)                   mtri.Triangulation(x, y)
 129: (0)           def test_delaunay_robust():
 130: (4)               tri_points = np.array([
 131: (8)                   [0.8660254037844384, -0.5000000000000004],
 132: (8)                   [0.7577722283113836, -0.5000000000000004],
 133: (8)                   [0.6495190528383288, -0.5000000000000003],
 134: (8)                   [0.5412658773652739, -0.5000000000000003],
 135: (8)                   [0.811898816047911, -0.40625000000000044],
 136: (8)                   [0.7036456405748561, -0.4062500000000004],
 137: (8)                   [0.5953924651018013, -0.40625000000000033]])
 138: (4)               test_points = np.asarray([
 139: (8)                   [0.58, -0.46],
 140: (8)                   [0.65, -0.46],
 141: (8)                   [0.65, -0.42],
```

```
(8)                     [0.7, -0.48],
(8)                     [0.7, -0.44],
(8)                     [0.75, -0.44],
(8)                     [0.8, -0.48]])
(4)                 def tri_contains_point(xtri, ytri, xy):
(8)                     tri_points = np.vstack((xtri, ytri)).T
(8)                     return Path(tri_points).contains_point(xy)
(4)                 def tris_contain_point(triang, xy):
(8)                     return sum(tri_contains_point(triang.x[tri], triang.y[tri], xy)
(19)                                for tri in triang.triangles)
(4)                 triang = mtri.Triangulation(tri_points[:, 0], tri_points[:, 1])
(4)                 for test_point in test_points:
(8)                     assert tris_contain_point(triang, test_point) == 1
(4)                 triang = mtri.Triangulation(tri_points[1:, 0], tri_points[1:, 1])
(0)             @image_comparison(['tripcolor1.png'])
(0)             def test_tripcolor():
(4)                 x = np.asarray([0, 0.5, 1, 0,   0.5, 1,   0, 0.5, 1, 0.75])
(4)                 y = np.asarray([0, 0,   0, 0.5, 0.5, 0.5, 1, 1,   1, 0.75])
(4)                 triangles = np.asarray([
(8)                     [0, 1, 3], [1, 4, 3],
(8)                     [1, 2, 4], [2, 5, 4],
(8)                     [3, 4, 6], [4, 7, 6],
(8)                     [4, 5, 9], [7, 4, 9], [8, 7, 9], [5, 8, 9]])
(4)                 triang = mtri.Triangulation(x, y, triangles)
(4)                 Cpoints = x + 0.5*y
(4)                 xmid = x[triang.triangles].mean(axis=1)
(4)                 ymid = y[triang.triangles].mean(axis=1)
(4)                 Cfaces = 0.5*xmid + ymid
(4)                 plt.subplot(121)
(4)                 plt.tripcolor(triang, Cpoints, edgecolors='k')
(4)                 plt.title('point colors')
(4)                 plt.subplot(122)
(4)                 plt.tripcolor(triang, facecolors=Cfaces, edgecolors='k')
(4)                 plt.title('facecolors')
(0)             def test_tripcolor_color():
(4)                 x = [-1, 0, 1, 0]
(4)                 y = [0, -1, 0, 1]
(4)                 fig, ax = plt.subplots()
(4)                 with pytest.raises(TypeError, match=r"tripcolor\(\) missing 1 required "):
(8)                     ax.tripcolor(x, y)
(4)                 with pytest.raises(ValueError, match="The length of c must match either"):
(8)                     ax.tripcolor(x, y, [1, 2, 3])
(4)                 with pytest.raises(ValueError,
(23)                                   match="length of facecolors must match .* triangles"):
(8)                     ax.tripcolor(x, y, facecolors=[1, 2, 3, 4])
(4)                 with pytest.raises(ValueError,
(23)                                   match="'gouraud' .* at the points.* not at the faces"):
(8)                     ax.tripcolor(x, y, facecolors=[1, 2], shading='gouraud')
(4)                 with pytest.raises(ValueError,
(23)                                   match="'gouraud' .* at the points.* not at the faces"):
(8)                     ax.tripcolor(x, y, [1, 2], shading='gouraud')  # faces
(4)                 with pytest.raises(TypeError,
(23)                                   match="positional.*'c'.*keyword-only.*'facecolors'"):
(8)                     ax.tripcolor(x, y, C=[1, 2, 3, 4])
(4)                 with pytest.raises(TypeError, match="Unexpected positional parameter"):
(8)                     ax.tripcolor(x, y, [1, 2], 'unused_positional')
(4)                 ax.tripcolor(x, y, [1, 2, 3, 4])  # edges
(4)                 ax.tripcolor(x, y, [1, 2, 3, 4], shading='gouraud')  # edges
(4)                 ax.tripcolor(x, y, [1, 2])  # faces
(4)                 ax.tripcolor(x, y, facecolors=[1, 2])  # faces
(0)             def test_tripcolor_clim():
(4)                 np.random.seed(19680801)
(4)                 a, b, c = np.random.rand(10), np.random.rand(10), np.random.rand(10)
(4)                 ax = plt.figure().add_subplot()
(4)                 clim = (0.25, 0.75)
(4)                 norm = ax.tripcolor(a, b, c, clim=clim).norm
(4)                 assert (norm.vmin, norm.vmax) == clim
(0)             def test_tripcolor_warnings():
(4)                 x = [-1, 0, 1, 0]
```

```
211: (4)                y = [0, -1, 0, 1]
212: (4)                c = [0.4, 0.5]
213: (4)                fig, ax = plt.subplots()
214: (4)                with pytest.warns(UserWarning, match="Positional parameter c .*no
effect"):
215: (8)                    ax.tripcolor(x, y, c, facecolors=c)
216: (4)                with pytest.warns(UserWarning, match="Positional parameter c .*no
effect"):
217: (8)                    ax.tripcolor(x, y, 'interpreted as c', facecolors=c)
218: (0)            def test_no_modify():
219: (4)                triangles = np.array([[3, 2, 0], [3, 1, 0]], dtype=np.int32)
220: (4)                points = np.array([(0, 0), (0, 1.1), (1, 0), (1, 1)])
221: (4)                old_triangles = triangles.copy()
222: (4)                mtri.Triangulation(points[:, 0], points[:, 1], triangles).edges
223: (4)                assert_array_equal(old_triangles, triangles)
224: (0)            def test_trifinder():
225: (4)                x, y = np.meshgrid(np.arange(4), np.arange(4))
226: (4)                x = x.ravel()
227: (4)                y = y.ravel()
228: (4)                triangles = [[0, 1, 4], [1, 5, 4], [1, 2, 5], [2, 6, 5], [2, 3, 6],
229: (17)                            [3, 7, 6], [4, 5, 8], [5, 9, 8], [5, 6, 9], [6, 10, 9],
230: (17)                            [6, 7, 10], [7, 11, 10], [8, 9, 12], [9, 13, 12], [9, 10,
13],
231: (17)                            [10, 14, 13], [10, 11, 14], [11, 15, 14]]
232: (4)                mask = np.zeros(len(triangles))
233: (4)                mask[8:10] = 1
234: (4)                triang = mtri.Triangulation(x, y, triangles, mask)
235: (4)                trifinder = triang.get_trifinder()
236: (4)                xs = [0.25, 1.25, 2.25, 3.25]
237: (4)                ys = [0.25, 1.25, 2.25, 3.25]
238: (4)                xs, ys = np.meshgrid(xs, ys)
239: (4)                xs = xs.ravel()
240: (4)                ys = ys.ravel()
241: (4)                tris = trifinder(xs, ys)
242: (4)                assert_array_equal(tris, [0, 2, 4, -1, 6, -1, 10, -1,
243: (30)                                         12, 14, 16, -1, -1, -1, -1, -1])
244: (4)                tris = trifinder(xs-0.5, ys-0.5)
245: (4)                assert_array_equal(tris, [-1, -1, -1, -1, -1, 1, 3, 5,
246: (30)                                         -1, 7, -1, 11, -1, 13, 15, 17])
247: (4)                xs = [0.5, 1.5, 2.5, 0.5, 1.5, 2.5, 1.5, 1.5, 0.0, 1.0, 2.0, 3.0]
248: (4)                ys = [0.0, 0.0, 0.0, 3.0, 3.0, 3.0, 1.0, 2.0, 1.5, 1.5, 1.5, 1.5]
249: (4)                tris = trifinder(xs, ys)
250: (4)                assert_array_equal(tris, [0, 2, 4, 13, 15, 17, 3, 14, 6, 7, 10, 11])
251: (4)                xs = [0.0, 3.0]
252: (4)                ys = [0.0, 3.0]
253: (4)                tris = trifinder(xs, ys)
254: (4)                assert_array_equal(tris, [0, 17])
255: (4)                delta = 0.0
256: (4)                x = [1.5, 0,  1,  2, 3, 1.5,   1.5]
257: (4)                y = [-1,  0,  0,  0, 0, delta, 1]
258: (4)                triangles = [[0, 2, 1], [0, 3, 2], [0, 4, 3], [1, 2, 5], [2, 3, 5],
259: (17)                            [3, 4, 5], [1, 5, 6], [4, 6, 5]]
260: (4)                triang = mtri.Triangulation(x, y, triangles)
261: (4)                trifinder = triang.get_trifinder()
262: (4)                xs = [-0.1, 0.4, 0.9, 1.4, 1.9, 2.4, 2.9]
263: (4)                ys = [-0.1, 0.1]
264: (4)                xs, ys = np.meshgrid(xs, ys)
265: (4)                tris = trifinder(xs, ys)
266: (4)                assert_array_equal(tris, [[-1, 0, 0, 1, 1, 2, -1],
267: (30)                                         [-1, 6, 6, 6, 7, 7, -1]])
268: (4)                delta = 0.0
269: (4)                x = [-1, -delta, 0,  0,  0, 0, 1]
270: (4)                y = [1.5, 1.5,   0,  1,  2, 3, 1.5]
271: (4)                triangles = [[0, 1, 2], [0, 1, 5], [1, 2, 3], [1, 3, 4], [1, 4, 5],
272: (17)                            [2, 6, 3], [3, 6, 4], [4, 6, 5]]
273: (4)                triang = mtri.Triangulation(x, y, triangles)
274: (4)                trifinder = triang.get_trifinder()
275: (4)                xs = [-0.1, 0.1]
276: (4)                ys = [-0.1, 0.4, 0.9, 1.4, 1.9, 2.4, 2.9]
```

```
(4)                 xs, ys = np.meshgrid(xs, ys)
(4)                 tris = trifinder(xs, ys)
(4)                 assert_array_equal(tris, [[-1, -1], [0, 5], [0, 5], [0, 6], [1, 6], [1,
7],
(30)                                          [-1, -1]])
(4)                 x = [0, 1, 0, 1]
(4)                 y = [0, 0, 1, 1]
(4)                 triangles = [[0, 1, 2], [1, 3, 2]]
(4)                 triang = mtri.Triangulation(x, y, triangles)
(4)                 trifinder = triang.get_trifinder()
(4)                 xs = [-0.2, 0.2, 0.8, 1.2]
(4)                 ys = [0.5, 0.5, 0.5, 0.5]
(4)                 tris = trifinder(xs, ys)
(4)                 assert_array_equal(tris, [-1, 0, 1, -1])
(4)                 triang.set_mask([1, 0])
(4)                 assert trifinder == triang.get_trifinder()
(4)                 tris = trifinder(xs, ys)
(4)                 assert_array_equal(tris, [-1, -1, 1, -1])
(0)             def test_triinterp():
(4)                 x, y = np.meshgrid(np.arange(4), np.arange(4))
(4)                 x = x.ravel()
(4)                 y = y.ravel()
(4)                 z = 1.23*x - 4.79*y
(4)                 triangles = [[0, 1, 4], [1, 5, 4], [1, 2, 5], [2, 6, 5], [2, 3, 6],
(17)                             [3, 7, 6], [4, 5, 8], [5, 9, 8], [5, 6, 9], [6, 10, 9],
(17)                             [6, 7, 10], [7, 11, 10], [8, 9, 12], [9, 13, 12], [9, 10,
13],
(17)                             [10, 14, 13], [10, 11, 14], [11, 15, 14]]
(4)                 mask = np.zeros(len(triangles))
(4)                 mask[8:10] = 1
(4)                 triang = mtri.Triangulation(x, y, triangles, mask)
(4)                 linear_interp = mtri.LinearTriInterpolator(triang, z)
(4)                 cubic_min_E = mtri.CubicTriInterpolator(triang, z)
(4)                 cubic_geom = mtri.CubicTriInterpolator(triang, z, kind='geom')
(4)                 xs = np.linspace(0.25, 2.75, 6)
(4)                 ys = [0.25, 0.75, 2.25, 2.75]
(4)                 xs, ys = np.meshgrid(xs, ys)  # Testing arrays with array.ndim = 2
(4)                 for interp in (linear_interp, cubic_min_E, cubic_geom):
(8)                     zs = interp(xs, ys)
(8)                     assert_array_almost_equal(zs, (1.23*xs - 4.79*ys))
(4)                 xs = [-0.25, 1.25, 1.75, 3.25]
(4)                 ys = xs
(4)                 xs, ys = np.meshgrid(xs, ys)
(4)                 for interp in (linear_interp, cubic_min_E, cubic_geom):
(8)                     zs = linear_interp(xs, ys)
(8)                     assert_array_equal(zs.mask, [[True]*4]*4)
(4)                 xs = np.linspace(0.25, 1.75, 6)
(4)                 ys = [0.25, 0.75, 1.25, 1.75]
(4)                 xs, ys = np.meshgrid(xs, ys)
(4)                 for interp in (linear_interp, cubic_min_E, cubic_geom):
(8)                     zs = interp(xs, ys)
(8)                     matest.assert_array_almost_equal(zs, (1.23*xs - 4.79*ys))
(8)                     mask = (xs >= 1) * (xs <= 2) * (ys >= 1) * (ys <= 2)
(8)                     assert_array_equal(zs.mask, mask)
(4)                 (a, b, c) = (1.23, -4.79, 0.6)
(4)                 def quad(x, y):
(8)                     return a*(x-0.5)**2 + b*(y-0.5)**2 + c*x*y
(4)                 def gradient_quad(x, y):
(8)                     return (2*a*(x-0.5) + c*y, 2*b*(y-0.5) + c*x)
(4)                 x = np.array([0.2, 0.33367, 0.669, 0., 1., 1., 0.])
(4)                 y = np.array([0.3, 0.80755, 0.4335, 0., 0., 1., 1.])
(4)                 triangles = np.array([[0, 1, 2], [3, 0, 4], [4, 0, 2], [4, 2, 5],
(26)                                      [1, 5, 2], [6, 5, 1], [6, 1, 0], [6, 0, 3]])
(4)                 triang = mtri.Triangulation(x, y, triangles)
(4)                 z = quad(x, y)
(4)                 dz = gradient_quad(x, y)
(4)                 xs = np.linspace(0., 1., 5)
(4)                 ys = np.linspace(0., 1., 5)
(4)                 xs, ys = np.meshgrid(xs, ys)
```

```
(4)                 cubic_user = mtri.CubicTriInterpolator(triang, z, kind='user', dz=dz)
(4)                 interp_zs = cubic_user(xs, ys)
(4)                 assert_array_almost_equal(interp_zs, quad(xs, ys))
(4)                 (interp_dzsdx, interp_dzsdy) = cubic_user.gradient(x, y)
(4)                 (dzsdx, dzsdy) = gradient_quad(x, y)
(4)                 assert_array_almost_equal(interp_dzsdx, dzsdx)
(4)                 assert_array_almost_equal(interp_dzsdy, dzsdy)
(4)                 n = 11
(4)                 x, y = np.meshgrid(np.linspace(0., 1., n+1), np.linspace(0., 1., n+1))
(4)                 x = x.ravel()
(4)                 y = y.ravel()
(4)                 z = quad(x, y)
(4)                 triang = mtri.Triangulation(x, y, triangles=meshgrid_triangles(n+1))
(4)                 xs, ys = np.meshgrid(np.linspace(0.1, 0.9, 5), np.linspace(0.1, 0.9, 5))
(4)                 xs = xs.ravel()
(4)                 ys = ys.ravel()
(4)                 linear_interp = mtri.LinearTriInterpolator(triang, z)
(4)                 cubic_min_E = mtri.CubicTriInterpolator(triang, z)
(4)                 cubic_geom = mtri.CubicTriInterpolator(triang, z, kind='geom')
(4)                 zs = quad(xs, ys)
(4)                 diff_lin = np.abs(linear_interp(xs, ys) - zs)
(4)                 for interp in (cubic_min_E, cubic_geom):
(8)                     diff_cubic = np.abs(interp(xs, ys) - zs)
(8)                     assert np.max(diff_lin) >= 10 * np.max(diff_cubic)
(8)                     assert (np.dot(diff_lin, diff_lin) >=
(16)                            100 * np.dot(diff_cubic, diff_cubic))
(0)             def test_triinterpcubic_C1_continuity():
(4)                 def check_continuity(interpolator, loc, values=None):
(8)                     """
(8)                     Checks the continuity of interpolator (and its derivatives) near
(8)                     location loc. Can check the value at loc itself if *values* is
(8)                     provided.
(8)                     *interpolator* TriInterpolator
(8)                     *loc* location to test (x0, y0)
(8)                     *values* (optional) array [z0, dzx0, dzy0] to check the value at *loc*
(8)                     """
(8)                     n_star = 24       # Number of continuity points in a boundary of loc
(8)                     epsilon = 1.e-10  # Distance for loc boundary
(8)                     k = 100.          # Continuity coefficient
(8)                     (loc_x, loc_y) = loc
(8)                     star_x = loc_x + epsilon*np.cos(np.linspace(0., 2*np.pi, n_star))
(8)                     star_y = loc_y + epsilon*np.sin(np.linspace(0., 2*np.pi, n_star))
(8)                     z = interpolator([loc_x], [loc_y])[0]
(8)                     (dzx, dzy) = interpolator.gradient([loc_x], [loc_y])
(8)                     if values is not None:
(12)                        assert_array_almost_equal(z, values[0])
(12)                        assert_array_almost_equal(dzx[0], values[1])
(12)                        assert_array_almost_equal(dzy[0], values[2])
(8)                     diff_z = interpolator(star_x, star_y) - z
(8)                     (tab_dzx, tab_dzy) = interpolator.gradient(star_x, star_y)
(8)                     diff_dzx = tab_dzx - dzx
(8)                     diff_dzy = tab_dzy - dzy
(8)                     assert_array_less(diff_z, epsilon*k)
(8)                     assert_array_less(diff_dzx, epsilon*k)
(8)                     assert_array_less(diff_dzy, epsilon*k)
(4)                 (ax, ay) = (0.2, 0.3)
(4)                 (bx, by) = (0.33367, 0.80755)
(4)                 (cx, cy) = (0.669, 0.4335)
(4)                 x = np.array([ax, bx, cx, 0., 1., 1., 0.])
(4)                 y = np.array([ay, by, cy, 0., 0., 1., 1.])
(4)                 triangles = np.array([[0, 1, 2], [3, 0, 4], [4, 0, 2], [4, 2, 5],
(26)                                      [1, 5, 2], [6, 5, 1], [6, 1, 0], [6, 0, 3]])
(4)                 triang = mtri.Triangulation(x, y, triangles)
(4)                 for idof in range(9):
(8)                     z = np.zeros(7, dtype=np.float64)
(8)                     dzx = np.zeros(7, dtype=np.float64)
(8)                     dzy = np.zeros(7, dtype=np.float64)
(8)                     values = np.zeros([3, 3], dtype=np.float64)
(8)                     case = idof//3
```

```
413: (8)                    values[case, idof % 3] = 1.0
414: (8)                    if case == 0:
415: (12)                       z[idof] = 1.0
416: (8)                    elif case == 1:
417: (12)                       dzx[idof % 3] = 1.0
418: (8)                    elif case == 2:
419: (12)                       dzy[idof % 3] = 1.0
420: (8)                    interp = mtri.CubicTriInterpolator(triang, z, kind='user',
421: (43)                                                      dz=(dzx, dzy))
422: (8)                    check_continuity(interp, (ax, ay), values[:, 0])
423: (8)                    check_continuity(interp, (bx, by), values[:, 1])
424: (8)                    check_continuity(interp, (cx, cy), values[:, 2])
425: (8)                    check_continuity(interp, ((ax+bx)*0.5, (ay+by)*0.5))
426: (8)                    check_continuity(interp, ((ax+cx)*0.5, (ay+cy)*0.5))
427: (8)                    check_continuity(interp, ((cx+bx)*0.5, (cy+by)*0.5))
428: (8)                    check_continuity(interp, ((ax+bx+cx)/3., (ay+by+cy)/3.))
429: (8)                    check_continuity(interp, ((4.*ax+bx+cx)/6., (4.*ay+by+cy)/6.))
430: (8)                    check_continuity(interp, ((ax+4.*bx+cx)/6., (ay+4.*by+cy)/6.))
431: (8)                    check_continuity(interp, ((ax+bx+4.*cx)/6., (ay+by+4.*cy)/6.))
432: (0)            def test_triinterpcubic_cg_solver():
433: (4)                def poisson_sparse_matrix(n, m):
434: (8)                    """
435: (8)                    Return the sparse, (n*m, n*m) matrix in coo format resulting from the
436: (8)                    discretisation of the 2-dimensional Poisson equation according to a
437: (8)                    finite difference numerical scheme on a uniform (n, m) grid.
438: (8)                    """
439: (8)                    l = m*n
440: (8)                    rows = np.concatenate([
441: (12)                       np.arange(l, dtype=np.int32),
442: (12)                       np.arange(l-1, dtype=np.int32), np.arange(1, l, dtype=np.int32),
443: (12)                       np.arange(l-n, dtype=np.int32), np.arange(n, l, dtype=np.int32)])
444: (8)                    cols = np.concatenate([
445: (12)                       np.arange(l, dtype=np.int32),
446: (12)                       np.arange(1, l, dtype=np.int32), np.arange(l-1, dtype=np.int32),
447: (12)                       np.arange(n, l, dtype=np.int32), np.arange(l-n, dtype=np.int32)])
448: (8)                    vals = np.concatenate([
449: (12)                       4*np.ones(l, dtype=np.float64),
450: (12)                       -np.ones(l-1, dtype=np.float64), -np.ones(l-1, dtype=np.float64),
451: (12)                       -np.ones(l-n, dtype=np.float64), -np.ones(l-n, dtype=np.float64)])
452: (8)                    vals[l:2*l-1][m-1::m] = 0.
453: (8)                    vals[2*l-1:3*l-2][m-1::m] = 0.
454: (8)                    return vals, rows, cols, (n*m, n*m)
455: (4)                (n, m) = (12, 4)
456: (4)                mat = mtri._triinterpolate._Sparse_Matrix_coo(*poisson_sparse_matrix(n,
m))
457: (4)                mat.compress_csc()
458: (4)                mat_dense = mat.to_dense()
459: (4)                for itest in range(n*m):
460: (8)                    b = np.zeros(n*m, dtype=np.float64)
461: (8)                    b[itest] = 1.
462: (8)                    x, _ = mtri._triinterpolate._cg(A=mat, b=b, x0=np.zeros(n*m),
463: (40)                                                   tol=1.e-10)
464: (8)                    assert_array_almost_equal(np.dot(mat_dense, x), b)
465: (4)                (i_zero, j_zero) = (12, 49)
466: (4)                vals, rows, cols, _ = poisson_sparse_matrix(n, m)
467: (4)                rows = rows + 1*(rows >= i_zero) + 1*(rows >= j_zero)
468: (4)                cols = cols + 1*(cols >= i_zero) + 1*(cols >= j_zero)
469: (4)                rows = np.concatenate([rows, [i_zero, i_zero-1, j_zero, j_zero-1]])
470: (4)                cols = np.concatenate([cols, [i_zero-1, i_zero, j_zero-1, j_zero]])
471: (4)                vals = np.concatenate([vals, [1., 1., 1., 1.]])
472: (4)                mat = mtri._triinterpolate._Sparse_Matrix_coo(vals, rows, cols,
473: (50)                                                             (n*m + 2, n*m + 2))
474: (4)                mat.compress_csc()
475: (4)                mat_dense = mat.to_dense()
476: (4)                for itest in range(n*m + 2):
477: (8)                    b = np.zeros(n*m + 2, dtype=np.float64)
478: (8)                    b[itest] = 1.
479: (8)                    x, _ = mtri._triinterpolate._cg(A=mat, b=b, x0=np.ones(n * m + 2),
480: (40)                                                   tol=1.e-10)
```

```
481: (8)                    assert_array_almost_equal(np.dot(mat_dense, x), b)
482: (4)                vals = np.ones(17, dtype=np.float64)
483: (4)                rows = np.array([0, 1, 2, 0, 0, 1, 1, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1],
484: (20)                               dtype=np.int32)
485: (4)                cols = np.array([0, 1, 2, 1, 1, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2],
486: (20)                               dtype=np.int32)
487: (4)                dim = (3, 3)
488: (4)                mat = mtri._triinterpolate._Sparse_Matrix_coo(vals, rows, cols, dim)
489: (4)                mat.compress_csc()
490: (4)                mat_dense = mat.to_dense()
491: (4)                assert_array_almost_equal(mat_dense, np.array([
492: (8)                    [1., 2., 0.], [2., 1., 5.], [0., 5., 1.]], dtype=np.float64))
493: (0)            def test_triinterpcubic_geom_weights():
494: (4)                (ax, ay) = (0., 1.687)
495: (4)                x = np.array([ax, 0.5*ax, 0., 1.])
496: (4)                y = np.array([ay, -ay, 0., 0.])
497: (4)                z = np.zeros(4, dtype=np.float64)
498: (4)                triangles = [[0, 2, 3], [1, 3, 2]]
499: (4)                sum_w = np.zeros([4, 2])  # 4 possibilities; 2 triangles
500: (4)                for theta in np.linspace(0., 2*np.pi, 14):  # rotating the figure...
501: (8)                    x_rot = np.cos(theta)*x + np.sin(theta)*y
502: (8)                    y_rot = -np.sin(theta)*x + np.cos(theta)*y
503: (8)                    triang = mtri.Triangulation(x_rot, y_rot, triangles)
504: (8)                    cubic_geom = mtri.CubicTriInterpolator(triang, z, kind='geom')
505: (8)                    dof_estimator = mtri._triinterpolate._DOF_estimator_geom(cubic_geom)
506: (8)                    weights = dof_estimator.compute_geom_weights()
507: (8)                    sum_w[0, :] = np.sum(weights, 1) - 1
508: (8)                    for itri in range(3):
509: (12)                       sum_w[itri+1, :] = np.sum(weights, 1) - 2*weights[:, itri]
510: (8)                    assert_array_almost_equal(np.min(np.abs(sum_w), axis=0),
511: (34)                                             np.array([0., 0.], dtype=np.float64))
512: (0)            def test_triinterp_colinear():
513: (4)                delta = 0.
514: (4)                x0 = np.array([1.5, 0,  1,  2, 3, 1.5,   1.5])
515: (4)                y0 = np.array([-1,  0,  0,  0, 0, delta, 1])
516: (4)                transformations = [[1, 0], [0, 1], [1, 1], [1, 2], [-2, -1], [-2, 1]]
517: (4)                for transformation in transformations:
518: (8)                    x_rot = transformation[0]*x0 + transformation[1]*y0
519: (8)                    y_rot = -transformation[1]*x0 + transformation[0]*y0
520: (8)                    (x, y) = (x_rot, y_rot)
521: (8)                    z = 1.23*x - 4.79*y
522: (8)                    triangles = [[0, 2, 1], [0, 3, 2], [0, 4, 3], [1, 2, 5], [2, 3, 5],
523: (21)                                [3, 4, 5], [1, 5, 6], [4, 6, 5]]
524: (8)                    triang = mtri.Triangulation(x, y, triangles)
525: (8)                    xs = np.linspace(np.min(triang.x), np.max(triang.x), 20)
526: (8)                    ys = np.linspace(np.min(triang.y), np.max(triang.y), 20)
527: (8)                    xs, ys = np.meshgrid(xs, ys)
528: (8)                    xs = xs.ravel()
529: (8)                    ys = ys.ravel()
530: (8)                    mask_out = (triang.get_trifinder()(xs, ys) == -1)
531: (8)                    zs_target = np.ma.array(1.23*xs - 4.79*ys, mask=mask_out)
532: (8)                    linear_interp = mtri.LinearTriInterpolator(triang, z)
533: (8)                    cubic_min_E = mtri.CubicTriInterpolator(triang, z)
534: (8)                    cubic_geom = mtri.CubicTriInterpolator(triang, z, kind='geom')
535: (8)                    for interp in (linear_interp, cubic_min_E, cubic_geom):
536: (12)                       zs = interp(xs, ys)
537: (12)                       assert_array_almost_equal(zs_target, zs)
538: (8)                    itri = 4
539: (8)                    pt1 = triang.triangles[itri, 0]
540: (8)                    pt2 = triang.triangles[itri, 1]
541: (8)                    xs = np.linspace(triang.x[pt1], triang.x[pt2], 10)
542: (8)                    ys = np.linspace(triang.y[pt1], triang.y[pt2], 10)
543: (8)                    zs_target = 1.23*xs - 4.79*ys
544: (8)                    for interp in (linear_interp, cubic_min_E, cubic_geom):
545: (12)                       zs, = interp._interpolate_multikeys(
546: (16)                           xs, ys, tri_index=itri*np.ones(10, dtype=np.int32))
547: (12)                       assert_array_almost_equal(zs_target, zs)
548: (0)            def test_triinterp_transformations():
549: (4)                n_angles = 20
```

```
(4)                 n_radii = 10
(4)                 min_radius = 0.15
(4)                 def z(x, y):
(8)                     r1 = np.hypot(0.5 - x, 0.5 - y)
(8)                     theta1 = np.arctan2(0.5 - x, 0.5 - y)
(8)                     r2 = np.hypot(-x - 0.2, -y - 0.2)
(8)                     theta2 = np.arctan2(-x - 0.2, -y - 0.2)
(8)                     z = -(2*(np.exp((r1/10)**2)-1)*30. * np.cos(7.*theta1) +
(14)                          (np.exp((r2/10)**2)-1)*30. * np.cos(11.*theta2) +
(14)                          0.7*(x**2 + y**2))
(8)                     return (np.max(z)-z)/(np.max(z)-np.min(z))
(4)                 radii = np.linspace(min_radius, 0.95, n_radii)
(4)                 angles = np.linspace(0 + n_angles, 2*np.pi + n_angles,
(25)                                     n_angles, endpoint=False)
(4)                 angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
(4)                 angles[:, 1::2] += np.pi/n_angles
(4)                 x0 = (radii*np.cos(angles)).flatten()
(4)                 y0 = (radii*np.sin(angles)).flatten()
(4)                 triang0 = mtri.Triangulation(x0, y0)  # Delaunay triangulation
(4)                 z0 = z(x0, y0)
(4)                 xs0 = np.linspace(-1., 1., 23)
(4)                 ys0 = np.linspace(-1., 1., 23)
(4)                 xs0, ys0 = np.meshgrid(xs0, ys0)
(4)                 xs0 = xs0.ravel()
(4)                 ys0 = ys0.ravel()
(4)                 interp_z0 = {}
(4)                 for i_angle in range(2):
(8)                     theta = 2*np.pi / n_angles * i_angle
(8)                     x = np.cos(theta)*x0 + np.sin(theta)*y0
(8)                     y = -np.sin(theta)*x0 + np.cos(theta)*y0
(8)                     xs = np.cos(theta)*xs0 + np.sin(theta)*ys0
(8)                     ys = -np.sin(theta)*xs0 + np.cos(theta)*ys0
(8)                     triang = mtri.Triangulation(x, y, triang0.triangles)
(8)                     linear_interp = mtri.LinearTriInterpolator(triang, z0)
(8)                     cubic_min_E = mtri.CubicTriInterpolator(triang, z0)
(8)                     cubic_geom = mtri.CubicTriInterpolator(triang, z0, kind='geom')
(8)                     dic_interp = {'lin': linear_interp,
(22)                                  'min_E': cubic_min_E,
(22)                                  'geom': cubic_geom}
(8)                     for interp_key in ['lin', 'min_E', 'geom']:
(12)                        interp = dic_interp[interp_key]
(12)                        if i_angle == 0:
(16)                            interp_z0[interp_key] = interp(xs0, ys0)  # storage
(12)                        else:
(16)                            interpz = interp(xs, ys)
(16)                            matest.assert_array_almost_equal(interpz,
(49)                                                             interp_z0[interp_key])
(4)                 scale_factor = 987654.3210
(4)                 for scaled_axis in ('x', 'y'):
(8)                     if scaled_axis == 'x':
(12)                        x = scale_factor * x0
(12)                        y = y0
(12)                        xs = scale_factor * xs0
(12)                        ys = ys0
(8)                     else:
(12)                        x = x0
(12)                        y = scale_factor * y0
(12)                        xs = xs0
(12)                        ys = scale_factor * ys0
(8)                     triang = mtri.Triangulation(x, y, triang0.triangles)
(8)                     linear_interp = mtri.LinearTriInterpolator(triang, z0)
(8)                     cubic_min_E = mtri.CubicTriInterpolator(triang, z0)
(8)                     cubic_geom = mtri.CubicTriInterpolator(triang, z0, kind='geom')
(8)                     dic_interp = {'lin': linear_interp,
(22)                                  'min_E': cubic_min_E,
(22)                                  'geom': cubic_geom}
(8)                     for interp_key in ['lin', 'min_E', 'geom']:
(12)                        interpz = dic_interp[interp_key](xs, ys)
(12)                        matest.assert_array_almost_equal(interpz, interp_z0[interp_key])
```

```
(0)            @image_comparison(['tri_smooth_contouring.png'], remove_text=True, tol=0.072)
(0)            def test_tri_smooth_contouring():
(4)                n_angles = 20
(4)                n_radii = 10
(4)                min_radius = 0.15
(4)                def z(x, y):
(8)                    r1 = np.hypot(0.5 - x, 0.5 - y)
(8)                    theta1 = np.arctan2(0.5 - x, 0.5 - y)
(8)                    r2 = np.hypot(-x - 0.2, -y - 0.2)
(8)                    theta2 = np.arctan2(-x - 0.2, -y - 0.2)
(8)                    z = -(2*(np.exp((r1/10)**2)-1)*30. * np.cos(7.*theta1) +
(14)                       (np.exp((r2/10)**2)-1)*30. * np.cos(11.*theta2) +
(14)                       0.7*(x**2 + y**2))
(8)                    return (np.max(z)-z)/(np.max(z)-np.min(z))
(4)                radii = np.linspace(min_radius, 0.95, n_radii)
(4)                angles = np.linspace(0 + n_angles, 2*np.pi + n_angles,
(25)                                    n_angles, endpoint=False)
(4)                angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
(4)                angles[:, 1::2] += np.pi/n_angles
(4)                x0 = (radii*np.cos(angles)).flatten()
(4)                y0 = (radii*np.sin(angles)).flatten()
(4)                triang0 = mtri.Triangulation(x0, y0)  # Delaunay triangulation
(4)                z0 = z(x0, y0)
(4)                triang0.set_mask(np.hypot(x0[triang0.triangles].mean(axis=1),
(30)                                         y0[triang0.triangles].mean(axis=1))
(21)                                 < min_radius)
(4)                refiner = mtri.UniformTriRefiner(triang0)
(4)                tri_refi, z_test_refi = refiner.refine_field(z0, subdiv=4)
(4)                levels = np.arange(0., 1., 0.025)
(4)                plt.triplot(triang0, lw=0.5, color='0.5')
(4)                plt.tricontour(tri_refi, z_test_refi, levels=levels, colors="black")
(0)            @image_comparison(['tri_smooth_gradient.png'], remove_text=True, tol=0.092)
(0)            def test_tri_smooth_gradient():
(4)                def dipole_potential(x, y):
(8)                    """An electric dipole potential V."""
(8)                    r_sq = x**2 + y**2
(8)                    theta = np.arctan2(y, x)
(8)                    z = np.cos(theta)/r_sq
(8)                    return (np.max(z)-z) / (np.max(z)-np.min(z))
(4)                n_angles = 30
(4)                n_radii = 10
(4)                min_radius = 0.2
(4)                radii = np.linspace(min_radius, 0.95, n_radii)
(4)                angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)
(4)                angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
(4)                angles[:, 1::2] += np.pi/n_angles
(4)                x = (radii*np.cos(angles)).flatten()
(4)                y = (radii*np.sin(angles)).flatten()
(4)                V = dipole_potential(x, y)
(4)                triang = mtri.Triangulation(x, y)
(4)                triang.set_mask(np.hypot(x[triang.triangles].mean(axis=1),
(29)                                        y[triang.triangles].mean(axis=1))
(20)                                 < min_radius)
(4)                refiner = mtri.UniformTriRefiner(triang)
(4)                tri_refi, z_test_refi = refiner.refine_field(V, subdiv=3)
(4)                tci = mtri.CubicTriInterpolator(triang, -V)
(4)                Ex, Ey = tci.gradient(triang.x, triang.y)
(4)                E_norm = np.hypot(Ex, Ey)
(4)                plt.figure()
(4)                plt.gca().set_aspect('equal')
(4)                plt.triplot(triang, color='0.8')
(4)                levels = np.arange(0., 1., 0.01)
(4)                cmap = mpl.colormaps['hot']
(4)                plt.tricontour(tri_refi, z_test_refi, levels=levels, cmap=cmap,
(19)                              linewidths=[2.0, 1.0, 1.0, 1.0])
(4)                plt.quiver(triang.x, triang.y, Ex/E_norm, Ey/E_norm,
(15)                          units='xy', scale=10., zorder=3, color='blue',
(15)                          width=0.007, headwidth=3., headlength=4.)
(0)            def test_tritools():
```

```
(4)                 x = np.array([0., 1., 0.5, 0., 2.])
(4)                 y = np.array([0., 0., 0.5*np.sqrt(3.), -1., 1.])
(4)                 triangles = np.array([[0, 1, 2], [0, 1, 3], [1, 2, 4]], dtype=np.int32)
(4)                 mask = np.array([False, False, True], dtype=bool)
(4)                 triang = mtri.Triangulation(x, y, triangles, mask=mask)
(4)                 analyser = mtri.TriAnalyzer(triang)
(4)                 assert_array_almost_equal(analyser.scale_factors, [1, 1/(1+3**.5/2)])
(4)                 assert_array_almost_equal(
(8)                     analyser.circle_ratios(rescale=False),
(8)                     np.ma.masked_array([0.5, 1./(1.+np.sqrt(2.)), np.nan], mask))
(4)                 x = np.array([0., 1., 2.])
(4)                 y = np.array([1., 1.+3., 1.+6.])
(4)                 triangles = np.array([[0, 1, 2]], dtype=np.int32)
(4)                 triang = mtri.Triangulation(x, y, triangles)
(4)                 analyser = mtri.TriAnalyzer(triang)
(4)                 assert_array_almost_equal(analyser.circle_ratios(), np.array([0.]))
(4)                 n = 9
(4)                 def power(x, a):
(8)                     return np.abs(x)**a*np.sign(x)
(4)                 x = np.linspace(-1., 1., n+1)
(4)                 x, y = np.meshgrid(power(x, 2.), power(x, 0.25))
(4)                 x = x.ravel()
(4)                 y = y.ravel()
(4)                 triang = mtri.Triangulation(x, y, triangles=meshgrid_triangles(n+1))
(4)                 analyser = mtri.TriAnalyzer(triang)
(4)                 mask_flat = analyser.get_flat_tri_mask(0.2)
(4)                 verif_mask = np.zeros(162, dtype=bool)
(4)                 corners_index = [0, 1, 2, 3, 14, 15, 16, 17, 18, 19, 34, 35, 126, 127,
(21)                                 142, 143, 144, 145, 146, 147, 158, 159, 160, 161]
(4)                 verif_mask[corners_index] = True
(4)                 assert_array_equal(mask_flat, verif_mask)
(4)                 mask = np.zeros(162, dtype=bool)
(4)                 mask[80] = True
(4)                 triang.set_mask(mask)
(4)                 mask_flat = analyser.get_flat_tri_mask(0.2)
(4)                 center_index = [44, 45, 62, 63, 78, 79, 80, 81, 82, 83, 98, 99, 116, 117]
(4)                 verif_mask[center_index] = True
(4)                 assert_array_equal(mask_flat, verif_mask)
(0)             def test_trirefine():
(4)                 n = 3
(4)                 subdiv = 2
(4)                 x = np.linspace(-1., 1., n+1)
(4)                 x, y = np.meshgrid(x, x)
(4)                 x = x.ravel()
(4)                 y = y.ravel()
(4)                 mask = np.zeros(2*n**2, dtype=bool)
(4)                 mask[n**2:] = True
(4)                 triang = mtri.Triangulation(x, y, triangles=meshgrid_triangles(n+1),
(32)                                            mask=mask)
(4)                 refiner = mtri.UniformTriRefiner(triang)
(4)                 refi_triang = refiner.refine_triangulation(subdiv=subdiv)
(4)                 x_refi = refi_triang.x
(4)                 y_refi = refi_triang.y
(4)                 n_refi = n * subdiv**2
(4)                 x_verif = np.linspace(-1., 1., n_refi+1)
(4)                 x_verif, y_verif = np.meshgrid(x_verif, x_verif)
(4)                 x_verif = x_verif.ravel()
(4)                 y_verif = y_verif.ravel()
(4)                 ind1d = np.isin(np.around(x_verif*(2.5+y_verif), 8),
(20)                                np.around(x_refi*(2.5+y_refi), 8))
(4)                 assert_array_equal(ind1d, True)
(4)                 refi_mask = refi_triang.mask
(4)                 refi_tri_barycenter_x = np.sum(refi_triang.x[refi_triang.triangles],
(35)                                               axis=1) / 3.
(4)                 refi_tri_barycenter_y = np.sum(refi_triang.y[refi_triang.triangles],
(35)                                               axis=1) / 3.
(4)                 tri_finder = triang.get_trifinder()
(4)                 refi_tri_indices = tri_finder(refi_tri_barycenter_x,
(34)                                              refi_tri_barycenter_y)
```

```
757: (4)                refi_tri_mask = triang.mask[refi_tri_indices]
758: (4)                assert_array_equal(refi_mask, refi_tri_mask)
759: (4)                x = np.asarray([0.0, 1.0, 0.0, 1.0])
760: (4)                y = np.asarray([0.0, 0.0, 1.0, 1.0])
761: (4)                triang = [mtri.Triangulation(x, y, [[0, 1, 3], [3, 2, 0]]),
762: (14)                         mtri.Triangulation(x, y, [[0, 1, 3], [2, 0, 3]])]
763: (4)                z = np.hypot(x - 0.3, y - 0.4)
764: (4)                xyz_data = []
765: (4)                for i in range(2):
766: (8)                    refiner = mtri.UniformTriRefiner(triang[i])
767: (8)                    refined_triang, refined_z = refiner.refine_field(z, subdiv=1)
768: (8)                    xyz = np.dstack((refined_triang.x, refined_triang.y, refined_z))[0]
769: (8)                    xyz = xyz[np.lexsort((xyz[:, 1], xyz[:, 0]))]
770: (8)                    xyz_data += [xyz]
771: (4)                assert_array_almost_equal(xyz_data[0], xyz_data[1])
772: (0)            @pytest.mark.parametrize('interpolator',
773: (25)                                     [mtri.LinearTriInterpolator,
774: (26)                                      mtri.CubicTriInterpolator],
775: (25)                                     ids=['linear', 'cubic'])
776: (0)            def test_trirefine_masked(interpolator):
777: (4)                x, y = np.mgrid[:2, :2]
778: (4)                x = np.repeat(x.flatten(), 2)
779: (4)                y = np.repeat(y.flatten(), 2)
780: (4)                z = np.zeros_like(x)
781: (4)                tri = mtri.Triangulation(x, y)
782: (4)                refiner = mtri.UniformTriRefiner(tri)
783: (4)                interp = interpolator(tri, z)
784: (4)                refiner.refine_field(z, triinterpolator=interp, subdiv=2)
785: (0)            def meshgrid_triangles(n):
786: (4)                """
787: (4)                Return (2*(N-1)**2, 3) array of triangles to mesh (N, N)-point
np.meshgrid.
788: (4)                """
789: (4)                tri = []
790: (4)                for i in range(n-1):
791: (8)                    for j in range(n-1):
792: (12)                       a = i + j*n
793: (12)                       b = (i+1) + j*n
794: (12)                       c = i + (j+1)*n
795: (12)                       d = (i+1) + (j+1)*n
796: (12)                       tri += [[a, b, d], [a, d, c]]
797: (4)                return np.array(tri, dtype=np.int32)
798: (0)            def test_triplot_return():
799: (4)                ax = plt.figure().add_subplot()
800: (4)                triang = mtri.Triangulation(
801: (8)                    [0.0, 1.0, 0.0, 1.0], [0.0, 0.0, 1.0, 1.0],
802: (8)                    triangles=[[0, 1, 3], [3, 2, 0]])
803: (4)                assert ax.triplot(triang, "b-") is not None, \
804: (8)                    'triplot should return the artist it adds'
805: (0)            def test_trirefiner_fortran_contiguous_triangles():
806: (4)                triangles1 = np.array([[2, 0, 3], [2, 1, 0]])
807: (4)                assert not np.isfortran(triangles1)
808: (4)                triangles2 = np.array(triangles1, copy=True, order='F')
809: (4)                assert np.isfortran(triangles2)
810: (4)                x = np.array([0.39, 0.59, 0.43, 0.32])
811: (4)                y = np.array([33.99, 34.01, 34.19, 34.18])
812: (4)                triang1 = mtri.Triangulation(x, y, triangles1)
813: (4)                triang2 = mtri.Triangulation(x, y, triangles2)
814: (4)                refiner1 = mtri.UniformTriRefiner(triang1)
815: (4)                refiner2 = mtri.UniformTriRefiner(triang2)
816: (4)                fine_triang1 = refiner1.refine_triangulation(subdiv=1)
817: (4)                fine_triang2 = refiner2.refine_triangulation(subdiv=1)
818: (4)                assert_array_equal(fine_triang1.triangles, fine_triang2.triangles)
819: (0)            def test_qhull_triangle_orientation():
820: (4)                xi = np.linspace(-2, 2, 100)
821: (4)                x, y = map(np.ravel, np.meshgrid(xi, xi))
822: (4)                w = (x > y - 1) & (x < -1.95) & (y > -1.2)
823: (4)                x, y = x[w], y[w]
824: (4)                theta = np.radians(25)
```

```
825: (4)                x1 = x*np.cos(theta) - y*np.sin(theta)
826: (4)                y1 = x*np.sin(theta) + y*np.cos(theta)
827: (4)                triang = mtri.Triangulation(x1, y1)
828: (4)                qhull_neighbors = triang.neighbors
829: (4)                triang._neighbors = None
830: (4)                own_neighbors = triang.neighbors
831: (4)                assert_array_equal(qhull_neighbors, own_neighbors)
832: (0)            def test_trianalyzer_mismatched_indices():
833: (4)                x = np.array([0., 1., 0.5, 0., 2.])
834: (4)                y = np.array([0., 0., 0.5*np.sqrt(3.), -1., 1.])
835: (4)                triangles = np.array([[0, 1, 2], [0, 1, 3], [1, 2, 4]], dtype=np.int32)
836: (4)                mask = np.array([False, False, True], dtype=bool)
837: (4)                triang = mtri.Triangulation(x, y, triangles, mask=mask)
838: (4)                analyser = mtri.TriAnalyzer(triang)
839: (4)                analyser._get_compressed_triangulation()
840: (0)            def test_tricontourf_decreasing_levels():
841: (4)                x = [0.0, 1.0, 1.0]
842: (4)                y = [0.0, 0.0, 1.0]
843: (4)                z = [0.2, 0.4, 0.6]
844: (4)                plt.figure()
845: (4)                with pytest.raises(ValueError):
846: (8)                    plt.tricontourf(x, y, z, [1.0, 0.0])
847: (0)            def test_internal_cpp_api():
848: (4)                from matplotlib import _tri  # noqa: ensure lazy-loaded module *is*
loaded.
849: (4)                with pytest.raises(
850: (12)                       TypeError,
851: (12)                       match=r'__init__\(\): incompatible constructor arguments.'):
852: (8)                    mpl._tri.Triangulation()
853: (4)                with pytest.raises(
854: (12)                       ValueError, match=r'x and y must be 1D arrays of the same
length'):
855: (8)                    mpl._tri.Triangulation([], [1], [[]], (), (), (), False)
856: (4)                x = [0, 1, 1]
857: (4)                y = [0, 0, 1]
858: (4)                with pytest.raises(
859: (12)                       ValueError,
860: (12)                       match=r'triangles must be a 2D array of shape \(\?,3\)'):
861: (8)                    mpl._tri.Triangulation(x, y, [[0, 1]], (), (), (), False)
862: (4)                tris = [[0, 1, 2]]
863: (4)                with pytest.raises(
864: (12)                       ValueError,
865: (12)                       match=r'mask must be a 1D array with the same length as the '
866: (18)                             r'triangles array'):
867: (8)                    mpl._tri.Triangulation(x, y, tris, [0, 1], (), (), False)
868: (4)                with pytest.raises(
869: (12)                       ValueError, match=r'edges must be a 2D array with shape \
(\?,2\)'):
870: (8)                    mpl._tri.Triangulation(x, y, tris, (), [[1]], (), False)
871: (4)                with pytest.raises(
872: (12)                       ValueError,
873: (12)                       match=r'neighbors must be a 2D array with the same shape as the '
874: (18)                             r'triangles array'):
875: (8)                    mpl._tri.Triangulation(x, y, tris, (), (), [[-1]], False)
876: (4)                triang = mpl._tri.Triangulation(x, y, tris, (), (), (), False)
877: (4)                with pytest.raises(
878: (12)                       ValueError,
879: (12)                       match=r'z must be a 1D array with the same length as the '
880: (18)                             r'triangulation x and y arrays'):
881: (8)                    triang.calculate_plane_coefficients([])
882: (4)                for mask in ([0, 1], None):
883: (8)                    with pytest.raises(
884: (16)                           ValueError,
885: (16)                           match=r'mask must be a 1D array with the same length as the '
886: (22)                                 r'triangles array'):
887: (12)                       triang.set_mask(mask)
888: (4)                triang.set_mask([True])
889: (4)                assert_array_equal(triang.get_edges(), np.empty((0, 2)))
890: (4)                triang.set_mask(())  # Equivalent to Python Triangulation mask=None
```

```
891: (4)                assert_array_equal(triang.get_edges(), [[1, 0], [2, 0], [2, 1]])
892: (4)                with pytest.raises(
893: (12)                       TypeError,
894: (12)                       match=r'__init__\(\): incompatible constructor arguments.'):
895: (8)                    mpl._tri.TriContourGenerator()
896: (4)                with pytest.raises(
897: (12)                       ValueError,
898: (12)                       match=r'z must be a 1D array with the same length as the x and y '
899: (18)                             r'arrays'):
900: (8)                    mpl._tri.TriContourGenerator(triang, [1])
901: (4)                z = [0, 1, 2]
902: (4)                tcg = mpl._tri.TriContourGenerator(triang, z)
903: (4)                with pytest.raises(
904: (12)                       ValueError, match=r'filled contour levels must be increasing'):
905: (8)                    tcg.create_filled_contour(1, 0)
906: (4)                with pytest.raises(
907: (12)                       TypeError,
908: (12)                       match=r'__init__\(\): incompatible constructor arguments.'):
909: (8)                    mpl._tri.TrapezoidMapTriFinder()
910: (4)                trifinder = mpl._tri.TrapezoidMapTriFinder(triang)
911: (4)                with pytest.raises(
912: (12)                       ValueError, match=r'x and y must be array-like with same shape'):
913: (8)                    trifinder.find_many([0], [0, 1])
914: (0)            def test_qhull_large_offset():
915: (4)                x = np.asarray([0, 1, 0, 1, 0.5])
916: (4)                y = np.asarray([0, 0, 1, 1, 0.5])
917: (4)                offset = 1e10
918: (4)                triang = mtri.Triangulation(x, y)
919: (4)                triang_offset = mtri.Triangulation(x + offset, y + offset)
920: (4)                assert len(triang.triangles) == len(triang_offset.triangles)
921: (0)            def test_tricontour_non_finite_z():
922: (4)                x = [0, 1, 0, 1]
923: (4)                y = [0, 0, 1, 1]
924: (4)                triang = mtri.Triangulation(x, y)
925: (4)                plt.figure()
926: (4)                with pytest.raises(ValueError, match='z array must not contain non-finite
'
927: (41)                                                   'values within the triangulation'):
928: (8)                    plt.tricontourf(triang, [0, 1, 2, np.inf])
929: (4)                with pytest.raises(ValueError, match='z array must not contain non-finite
'
930: (41)                                                   'values within the triangulation'):
931: (8)                    plt.tricontourf(triang, [0, 1, 2, -np.inf])
932: (4)                with pytest.raises(ValueError, match='z array must not contain non-finite
'
933: (41)                                                   'values within the triangulation'):
934: (8)                    plt.tricontourf(triang, [0, 1, 2, np.nan])
935: (4)                with pytest.raises(ValueError, match='z must not contain masked points '
936: (41)                                                   'within the triangulation'):
937: (8)                    plt.tricontourf(triang, np.ma.array([0, 1, 2, 3], mask=[1, 0, 0, 0]))
938: (0)            def test_tricontourset_reuse():
939: (4)                x = [0.0, 0.5, 1.0]
940: (4)                y = [0.0, 1.0, 0.0]
941: (4)                z = [1.0, 2.0, 3.0]
942: (4)                fig, ax = plt.subplots()
943: (4)                tcs1 = ax.tricontourf(x, y, z)
944: (4)                tcs2 = ax.tricontour(x, y, z)
945: (4)                assert tcs2._contour_generator != tcs1._contour_generator
946: (4)                tcs3 = ax.tricontour(tcs1, z)
947: (4)                assert tcs3._contour_generator == tcs1._contour_generator
948: (0)            @check_figures_equal()
949: (0)            def test_triplot_with_ls(fig_test, fig_ref):
950: (4)                x = [0, 2, 1]
951: (4)                y = [0, 0, 1]
952: (4)                data = [[0, 1, 2]]
953: (4)                fig_test.subplots().triplot(x, y, data, ls='--')
954: (4)                fig_ref.subplots().triplot(x, y, data, linestyle='--')
955: (0)            def test_triplot_label():
956: (4)                x = [0, 2, 1]
```

```
957: (4)                y = [0, 0, 1]
958: (4)                data = [[0, 1, 2]]
959: (4)                fig, ax = plt.subplots()
960: (4)                lines, markers = ax.triplot(x, y, data, label='label')
961: (4)                handles, labels = ax.get_legend_handles_labels()
962: (4)                assert labels == ['label']
963: (4)                assert len(handles) == 1
964: (4)                assert handles[0] is lines
965: (0)            def test_tricontour_path():
966: (4)                x = [0, 4, 4, 0, 2]
967: (4)                y = [0, 0, 4, 4, 2]
968: (4)                triang = mtri.Triangulation(x, y)
969: (4)                _, ax = plt.subplots()
970: (4)                cs = ax.tricontour(triang, [1, 0, 0, 0, 0], levels=[0.5])
971: (4)                paths = cs.get_paths()
972: (4)                assert len(paths) == 1
973: (4)                expected_vertices = [[2, 0], [1, 1], [0, 2]]
974: (4)                assert_array_almost_equal(paths[0].vertices, expected_vertices)
975: (4)                assert_array_equal(paths[0].codes, [1, 2, 2])
976: (4)                assert_array_almost_equal(
977: (8)                    paths[0].to_polygons(closed_only=False), [expected_vertices])
978: (4)                cs = ax.tricontour(triang, [0, 0, 0, 0, 1], levels=[0.5])
979: (4)                paths = cs.get_paths()
980: (4)                assert len(paths) == 1
981: (4)                expected_vertices = [[3, 1], [3, 3], [1, 3], [1, 1], [3, 1]]
982: (4)                assert_array_almost_equal(paths[0].vertices, expected_vertices)
983: (4)                assert_array_equal(paths[0].codes, [1, 2, 2, 2, 79])
984: (4)                assert_array_almost_equal(paths[0].to_polygons(), [expected_vertices])
985: (0)            def test_tricontourf_path():
986: (4)                x = [0, 4, 4, 0, 2]
987: (4)                y = [0, 0, 4, 4, 2]
988: (4)                triang = mtri.Triangulation(x, y)
989: (4)                _, ax = plt.subplots()
990: (4)                cs = ax.tricontourf(triang, [0, 0, 0, 0, 1], levels=[0.5, 1.5])
991: (4)                paths = cs.get_paths()
992: (4)                assert len(paths) == 1
993: (4)                expected_vertices = [[3, 1], [3, 3], [1, 3], [1, 1], [3, 1]]
994: (4)                assert_array_almost_equal(paths[0].vertices, expected_vertices)
995: (4)                assert_array_equal(paths[0].codes, [1, 2, 2, 2, 79])
996: (4)                assert_array_almost_equal(paths[0].to_polygons(), [expected_vertices])
997: (4)                cs = ax.tricontourf(triang, [1, 0, 0, 0, 0], levels=[0.5, 1.5])
998: (4)                paths = cs.get_paths()
999: (4)                assert len(paths) == 1
1000: (4)               expected_vertices = [[2, 0], [1, 1], [0, 2], [0, 0], [2, 0]]
1001: (4)               assert_array_almost_equal(paths[0].vertices, expected_vertices)
1002: (4)               assert_array_equal(paths[0].codes, [1, 2, 2, 2, 79])
1003: (4)               assert_array_almost_equal(paths[0].to_polygons(), [expected_vertices])
1004: (4)               cs = ax.tricontourf(triang, [0, 0, 0, 0, 1], levels=[-0.5, 0.5])
1005: (4)               paths = cs.get_paths()
1006: (4)               assert len(paths) == 1
1007: (4)               expected_vertices = [[0, 0], [4, 0], [4, 4], [0, 4], [0, 0],
1008: (25)                                   [1, 1], [1, 3], [3, 3], [3, 1], [1, 1]]
1009: (4)               assert_array_almost_equal(paths[0].vertices, expected_vertices)
1010: (4)               assert_array_equal(paths[0].codes, [1, 2, 2, 2, 79, 1, 2, 2, 2, 79])
1011: (4)               assert_array_almost_equal(paths[0].to_polygons(),
np.split(expected_vertices, [5]))

----------------------------------------

File 226 - test_ttconv.py:

1: (0)              from pathlib import Path
2: (0)              import matplotlib
3: (0)              from matplotlib.testing.decorators import image_comparison
4: (0)              import matplotlib.pyplot as plt
5: (0)              @image_comparison(["truetype-conversion.pdf"])
6: (0)              def test_truetype_conversion(recwarn):
7: (4)                  matplotlib.rcParams['pdf.fonttype'] = 3
8: (4)                  fig, ax = plt.subplots()
```

```
9: (4)                  ax.text(0, 0, "ABCDE",
10: (12)                        font=Path(__file__).with_name("mpltest.ttf"), fontsize=80)
11: (4)                 ax.set_xticks([])
12: (4)                 ax.set_yticks([])
```

File 227 - test_units.py:

```
1: (0)              from datetime import datetime, timezone, timedelta
2: (0)              import platform
3: (0)              from unittest.mock import MagicMock
4: (0)              import matplotlib.pyplot as plt
5: (0)              from matplotlib.testing.decorators import check_figures_equal,
image_comparison
6: (0)              import matplotlib.units as munits
7: (0)              from matplotlib.category import UnitData
8: (0)              import numpy as np
9: (0)              import pytest
10: (0)             class Quantity:
11: (4)                 def __init__(self, data, units):
12: (8)                     self.magnitude = data
13: (8)                     self.units = units
14: (4)                 def to(self, new_units):
15: (8)                     factors = {('hours', 'seconds'): 3600, ('minutes', 'hours'): 1 / 60,
16: (19)                               ('minutes', 'seconds'): 60, ('feet', 'miles'): 1 / 5280.,
17: (19)                               ('feet', 'inches'): 12, ('miles', 'inches'): 12 * 5280}
18: (8)                     if self.units != new_units:
19: (12)                        mult = factors[self.units, new_units]
20: (12)                        return Quantity(mult * self.magnitude, new_units)
21: (8)                     else:
22: (12)                        return Quantity(self.magnitude, self.units)
23: (4)                 def __copy__(self):
24: (8)                     return Quantity(self.magnitude, self.units)
25: (4)                 def __getattr__(self, attr):
26: (8)                     return getattr(self.magnitude, attr)
27: (4)                 def __getitem__(self, item):
28: (8)                     if np.iterable(self.magnitude):
29: (12)                        return Quantity(self.magnitude[item], self.units)
30: (8)                     else:
31: (12)                        return Quantity(self.magnitude, self.units)
32: (4)                 def __array__(self):
33: (8)                     return np.asarray(self.magnitude)
34: (0)             @pytest.fixture
35: (0)             def quantity_converter():
36: (4)                 qc = munits.ConversionInterface()
37: (4)                 def convert(value, unit, axis):
38: (8)                     if hasattr(value, 'units'):
39: (12)                        return value.to(unit).magnitude
40: (8)                     elif np.iterable(value):
41: (12)                        try:
42: (16)                            return [v.to(unit).magnitude for v in value]
43: (12)                        except AttributeError:
44: (16)                            return [Quantity(v, axis.get_units()).to(unit).magnitude
45: (24)                                    for v in value]
46: (8)                     else:
47: (12)                        return Quantity(value, axis.get_units()).to(unit).magnitude
48: (4)                 def default_units(value, axis):
49: (8)                     if hasattr(value, 'units'):
50: (12)                        return value.units
51: (8)                     elif np.iterable(value):
52: (12)                        for v in value:
53: (16)                            if hasattr(v, 'units'):
54: (20)                                return v.units
55: (12)                        return None
56: (4)                 qc.convert = MagicMock(side_effect=convert)
57: (4)                 qc.axisinfo = MagicMock(side_effect=lambda u, a:
58: (28)                                        munits.AxisInfo(label=u, default_limits=(0, 100)))
59: (4)                 qc.default_units = MagicMock(side_effect=default_units)
```

```
(4)                 return qc
(0)             @image_comparison(['plot_pint.png'], style='mpl20',
(18)                            tol=0 if platform.machine() == 'x86_64' else 0.03)
(0)             def test_numpy_facade(quantity_converter):
(4)                 plt.rcParams['axes.formatter.limits'] = -7, 7
(4)                 munits.registry[Quantity] = quantity_converter
(4)                 y = Quantity(np.linspace(0, 30), 'miles')
(4)                 x = Quantity(np.linspace(0, 5), 'hours')
(4)                 fig, ax = plt.subplots()
(4)                 fig.subplots_adjust(left=0.15)  # Make space for label
(4)                 ax.plot(x, y, 'tab:blue')
(4)                 ax.axhline(Quantity(26400, 'feet'), color='tab:red')
(4)                 ax.axvline(Quantity(120, 'minutes'), color='tab:green')
(4)                 ax.yaxis.set_units('inches')
(4)                 ax.xaxis.set_units('seconds')
(4)                 assert quantity_converter.convert.called
(4)                 assert quantity_converter.axisinfo.called
(4)                 assert quantity_converter.default_units.called
(0)             @image_comparison(['plot_masked_units.png'], remove_text=True, style='mpl20',
(18)                            tol=0 if platform.machine() == 'x86_64' else 0.02)
(0)             def test_plot_masked_units():
(4)                 data = np.linspace(-5, 5)
(4)                 data_masked = np.ma.array(data, mask=(data > -2) & (data < 2))
(4)                 data_masked_units = Quantity(data_masked, 'meters')
(4)                 fig, ax = plt.subplots()
(4)                 ax.plot(data_masked_units)
(0)             def test_empty_set_limits_with_units(quantity_converter):
(4)                 munits.registry[Quantity] = quantity_converter
(4)                 fig, ax = plt.subplots()
(4)                 ax.set_xlim(Quantity(-1, 'meters'), Quantity(6, 'meters'))
(4)                 ax.set_ylim(Quantity(-1, 'hours'), Quantity(16, 'hours'))
(0)             @image_comparison(['jpl_bar_units.png'],
(18)                            savefig_kwarg={'dpi': 120}, style='mpl20')
(0)             def test_jpl_bar_units():
(4)                 import matplotlib.testing.jpl_units as units
(4)                 units.register()
(4)                 day = units.Duration("ET", 24.0 * 60.0 * 60.0)
(4)                 x = [0 * units.km, 1 * units.km, 2 * units.km]
(4)                 w = [1 * day, 2 * day, 3 * day]
(4)                 b = units.Epoch("ET", dt=datetime(2009, 4, 26))
(4)                 fig, ax = plt.subplots()
(4)                 ax.bar(x, w, bottom=b)
(4)                 ax.set_ylim([b - 1 * day, b + w[-1] + (1.001) * day])
(0)             @image_comparison(['jpl_barh_units.png'],
(18)                            savefig_kwarg={'dpi': 120}, style='mpl20')
(0)             def test_jpl_barh_units():
(4)                 import matplotlib.testing.jpl_units as units
(4)                 units.register()
(4)                 day = units.Duration("ET", 24.0 * 60.0 * 60.0)
(4)                 x = [0 * units.km, 1 * units.km, 2 * units.km]
(4)                 w = [1 * day, 2 * day, 3 * day]
(4)                 b = units.Epoch("ET", dt=datetime(2009, 4, 26))
(4)                 fig, ax = plt.subplots()
(4)                 ax.barh(x, w, left=b)
(4)                 ax.set_xlim([b - 1 * day, b + w[-1] + (1.001) * day])
(0)             def test_jpl_datetime_units_consistent():
(4)                 import matplotlib.testing.jpl_units as units
(4)                 units.register()
(4)                 dt = datetime(2009, 4, 26)
(4)                 jpl = units.Epoch("ET", dt=dt)
(4)                 dt_conv = munits.registry.get_converter(dt).convert(dt, None, None)
(4)                 jpl_conv = munits.registry.get_converter(jpl).convert(jpl, None, None)
(4)                 assert dt_conv == jpl_conv
(0)             def test_empty_arrays():
(4)                 plt.scatter(np.array([], dtype='datetime64[ns]'), np.array([]))
(0)             def test_scatter_element0_masked():
(4)                 times = np.arange('2005-02', '2005-03', dtype='datetime64[D]')
(4)                 y = np.arange(len(times), dtype=float)
(4)                 y[0] = np.nan
```

```
(4)                 fig, ax = plt.subplots()
(4)                 ax.scatter(times, y)
(4)                 fig.canvas.draw()
(0)             def test_errorbar_mixed_units():
(4)                 x = np.arange(10)
(4)                 y = [datetime(2020, 5, i * 2 + 1) for i in x]
(4)                 fig, ax = plt.subplots()
(4)                 ax.errorbar(x, y, timedelta(days=0.5))
(4)                 fig.canvas.draw()
(0)             @check_figures_equal(extensions=["png"])
(0)             def test_subclass(fig_test, fig_ref):
(4)                 class subdate(datetime):
(8)                     pass
(4)                 fig_test.subplots().plot(subdate(2000, 1, 1), 0, "o")
(4)                 fig_ref.subplots().plot(datetime(2000, 1, 1), 0, "o")
(0)             def test_shared_axis_quantity(quantity_converter):
(4)                 munits.registry[Quantity] = quantity_converter
(4)                 x = Quantity(np.linspace(0, 1, 10), "hours")
(4)                 y1 = Quantity(np.linspace(1, 2, 10), "feet")
(4)                 y2 = Quantity(np.linspace(3, 4, 10), "feet")
(4)                 fig, (ax1, ax2) = plt.subplots(2, 1, sharex='all', sharey='all')
(4)                 ax1.plot(x, y1)
(4)                 ax2.plot(x, y2)
(4)                 assert ax1.xaxis.get_units() == ax2.xaxis.get_units() == "hours"
(4)                 assert ax2.yaxis.get_units() == ax2.yaxis.get_units() == "feet"
(4)                 ax1.xaxis.set_units("seconds")
(4)                 ax2.yaxis.set_units("inches")
(4)                 assert ax1.xaxis.get_units() == ax2.xaxis.get_units() == "seconds"
(4)                 assert ax1.yaxis.get_units() == ax2.yaxis.get_units() == "inches"
(0)             def test_shared_axis_datetime():
(4)                 y1 = [datetime(2020, i, 1, tzinfo=timezone.utc) for i in range(1, 13)]
(4)                 y2 = [datetime(2021, i, 1, tzinfo=timezone.utc) for i in range(1, 13)]
(4)                 fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
(4)                 ax1.plot(y1)
(4)                 ax2.plot(y2)
(4)                 ax1.yaxis.set_units(timezone(timedelta(hours=5)))
(4)                 assert ax2.yaxis.units == timezone(timedelta(hours=5))
(0)             def test_shared_axis_categorical():
(4)                 d1 = {"a": 1, "b": 2}
(4)                 d2 = {"a": 3, "b": 4}
(4)                 fig, (ax1, ax2) = plt.subplots(1, 2, sharex=True, sharey=True)
(4)                 ax1.plot(d1.keys(), d1.values())
(4)                 ax2.plot(d2.keys(), d2.values())
(4)                 ax1.xaxis.set_units(UnitData(["c", "d"]))
(4)                 assert "c" in ax2.xaxis.get_units()._mapping.keys()
(0)             def test_empty_default_limits(quantity_converter):
(4)                 munits.registry[Quantity] = quantity_converter
(4)                 fig, ax1 = plt.subplots()
(4)                 ax1.xaxis.update_units(Quantity([10], "miles"))
(4)                 fig.draw_without_rendering()
(4)                 assert ax1.get_xlim() == (0, 100)
(4)                 ax1.yaxis.update_units(Quantity([10], "miles"))
(4)                 fig.draw_without_rendering()
(4)                 assert ax1.get_ylim() == (0, 100)
(4)                 fig, ax = plt.subplots()
(4)                 ax.axhline(30)
(4)                 ax.plot(Quantity(np.arange(0, 3), "miles"),
(12)                        Quantity(np.arange(0, 6, 2), "feet"))
(4)                 fig.draw_without_rendering()
(4)                 assert ax.get_xlim() == (0, 2)
(4)                 assert ax.get_ylim() == (0, 30)
(4)                 fig, ax = plt.subplots()
(4)                 ax.axvline(30)
(4)                 ax.plot(Quantity(np.arange(0, 3), "miles"),
(12)                        Quantity(np.arange(0, 6, 2), "feet"))
(4)                 fig.draw_without_rendering()
(4)                 assert ax.get_xlim() == (0, 30)
(4)                 assert ax.get_ylim() == (0, 4)
(4)                 fig, ax = plt.subplots()
```

```
198: (4)                ax.xaxis.update_units(Quantity([10], "miles"))
199: (4)                ax.axhline(30)
200: (4)                fig.draw_without_rendering()
201: (4)                assert ax.get_xlim() == (0, 100)
202: (4)                assert ax.get_ylim() == (28.5, 31.5)
203: (4)                fig, ax = plt.subplots()
204: (4)                ax.yaxis.update_units(Quantity([10], "miles"))
205: (4)                ax.axvline(30)
206: (4)                fig.draw_without_rendering()
207: (4)                assert ax.get_ylim() == (0, 100)
208: (4)                assert ax.get_xlim() == (28.5, 31.5)
209: (0)            class Kernel:
210: (4)                def __init__(self, array):
211: (8)                    self._array = np.asanyarray(array)
212: (4)                def __array__(self, dtype=None, copy=None):
213: (8)                    if dtype is not None and dtype != self._array.dtype:
214: (12)                       if copy is not None and not copy:
215: (16)                           raise ValueError(
216: (20)                               f"Converting array from {self._array.dtype} to "
217: (20)                               f"{dtype} requires a copy"
218: (16)                           )
219: (8)                    arr = np.asarray(self._array, dtype=dtype)
220: (8)                    return (arr if not copy else np.copy(arr))
221: (4)                @property
222: (4)                def shape(self):
223: (8)                    return self._array.shape
224: (0)            def test_plot_kernel():
225: (4)                kernel = Kernel([1, 2, 3, 4, 5])
226: (4)                plt.plot(kernel)

----------------------------------------

File 228 - test_type1font.py:

1: (0)              import matplotlib._type1font as t1f
2: (0)              import os.path
3: (0)              import difflib
4: (0)              import pytest
5: (0)              def test_Type1Font():
6: (4)                  filename = os.path.join(os.path.dirname(__file__), 'cmr10.pfb')
7: (4)                  font = t1f.Type1Font(filename)
8: (4)                  slanted = font.transform({'slant': 1})
9: (4)                  condensed = font.transform({'extend': 0.5})
10: (4)                 with open(filename, 'rb') as fd:
11: (8)                     rawdata = fd.read()
12: (4)                 assert font.parts[0] == rawdata[0x0006:0x10c5]
13: (4)                 assert font.parts[1] == rawdata[0x10cb:0x897f]
14: (4)                 assert font.parts[2] == rawdata[0x8985:0x8ba6]
15: (4)                 assert font.decrypted.startswith(b'dup\n/Private 18 dict dup begin')
16: (4)                 assert font.decrypted.endswith(b'mark currentfile closefile\n')
17: (4)                 assert slanted.decrypted.startswith(b'dup\n/Private 18 dict dup begin')
18: (4)                 assert slanted.decrypted.endswith(b'mark currentfile closefile\n')
19: (4)                 assert b'UniqueID 5000793' in font.parts[0]
20: (4)                 assert b'UniqueID 5000793' in font.decrypted
21: (4)                 assert font._pos['UniqueID'] == [(797, 818), (4483, 4504)]
22: (4)                 len0 = len(font.parts[0])
23: (4)                 for key in font._pos.keys():
24: (8)                     for pos0, pos1 in font._pos[key]:
25: (12)                        if pos0 < len0:
26: (16)                            data = font.parts[0][pos0:pos1]
27: (12)                        else:
28: (16)                            data = font.decrypted[pos0-len0:pos1-len0]
29: (12)                        assert data.startswith(f'/{key}'.encode('ascii'))
30: (4)                 assert {'FontType', 'FontMatrix', 'PaintType', 'ItalicAngle', 'RD'
31: (12)                        } < set(font._pos.keys())
32: (4)                 assert b'UniqueID 5000793' not in slanted.parts[0]
33: (4)                 assert b'UniqueID 5000793' not in slanted.decrypted
34: (4)                 assert 'UniqueID' not in slanted._pos
35: (4)                 assert font.prop['Weight'] == 'Medium'
```

```
36: (4)                 assert not font.prop['isFixedPitch']
37: (4)                 assert font.prop['ItalicAngle'] == 0
38: (4)                 assert slanted.prop['ItalicAngle'] == -45
39: (4)                 assert font.prop['Encoding'][5] == 'Pi'
40: (4)                 assert isinstance(font.prop['CharStrings']['Pi'], bytes)
41: (4)                 assert font._abbr['ND'] == 'ND'
42: (4)                 differ = difflib.Differ()
43: (4)                 diff = list(differ.compare(
44: (8)                     font.parts[0].decode('latin-1').splitlines(),
45: (8)                     slanted.parts[0].decode('latin-1').splitlines()))
46: (4)                 for line in (
47: (9)                      '- /UniqueID 5000793 def',
48: (9)                      '- /FontName /CMR10 def',
49: (9)                      '+ /FontName/CMR10_Slant_1000 def',
50: (9)                      '- /FontMatrix [0.001 0 0 0.001 0 0 ]readonly def',
51: (9)                      '+ /FontMatrix [0.001 0 0.001 0.001 0 0] readonly def',
52: (9)                      '-  /ItalicAngle 0 def',
53: (9)                      '+  /ItalicAngle -45.0 def'):
54: (8)                     assert line in diff, 'diff to slanted font must contain %s' % line
55: (4)                 diff = list(differ.compare(
56: (8)                     font.parts[0].decode('latin-1').splitlines(),
57: (8)                     condensed.parts[0].decode('latin-1').splitlines()))
58: (4)                 for line in (
59: (9)                      '- /UniqueID 5000793 def',
60: (9)                      '- /FontName /CMR10 def',
61: (9)                      '+ /FontName/CMR10_Extend_500 def',
62: (9)                      '- /FontMatrix [0.001 0 0 0.001 0 0 ]readonly def',
63: (9)                      '+ /FontMatrix [0.0005 0 0 0.001 0 0] readonly def'):
64: (8)                     assert line in diff, 'diff to condensed font must contain %s' % line
65: (0)             def test_Type1Font_2():
66: (4)                 filename = os.path.join(os.path.dirname(__file__),
67: (28)                                        'Courier10PitchBT-Bold.pfb')
68: (4)                 font = t1f.Type1Font(filename)
69: (4)                 assert font.prop['Weight'] == 'Bold'
70: (4)                 assert font.prop['isFixedPitch']
71: (4)                 assert font.prop['Encoding'][65] == 'A'  # the font uses StandardEncoding
72: (4)                 (pos0, pos1), = font._pos['Encoding']
73: (4)                 assert font.parts[0][pos0:pos1] == b'/Encoding StandardEncoding'
74: (4)                 assert font._abbr['ND'] == '|-'
75: (0)             def test_tokenize():
76: (4)                 data = (b'1234/abc false -9.81  Foo <<[0 1 2]<0 1ef a\t>>>\n'
77: (12)                        b'(string with(nested\t\\) par)ens\\\\)')
78: (4)                 n, w, num, kw, d = 'name', 'whitespace', 'number', 'keyword', 'delimiter'
79: (4)                 b, s = 'boolean', 'string'
80: (4)                 correct = [
81: (8)                     (num, 1234), (n, 'abc'), (w, ' '), (b, False), (w, ' '), (num, -9.81),
82: (8)                     (w, '  '), (kw, 'Foo'), (w, ' '), (d, '<<'), (d, '['), (num, 0),
83: (8)                     (w, ' '), (num, 1), (w, ' '), (num, 2), (d, ']'), (s,
b'\x01\xef\xa0'),
84: (8)                     (d, '>>'), (w, '\n'), (s, 'string with(nested\t) par)ens\\')
85: (4)                 ]
86: (4)                 correct_no_ws = [x for x in correct if x[0] != w]
87: (4)                 def convert(tokens):
88: (8)                     return [(t.kind, t.value()) for t in tokens]
89: (4)                 assert convert(t1f._tokenize(data, False)) == correct
90: (4)                 assert convert(t1f._tokenize(data, True)) == correct_no_ws
91: (4)                 def bin_after(n):
92: (8)                     tokens = t1f._tokenize(data, True)
93: (8)                     result = []
94: (8)                     for _ in range(n):
95: (12)                        result.append(next(tokens))
96: (8)                     result.append(tokens.send(10))
97: (8)                     return convert(result)
98: (4)                 for n in range(1, len(correct_no_ws)):
99: (8)                     result = bin_after(n)
100: (8)                    assert result[:-1] == correct_no_ws[:n]
101: (8)                    assert result[-1][0] == 'binary'
102: (8)                    assert isinstance(result[-1][1], bytes)
103: (0)            def test_tokenize_errors():
```

```
104: (4)                with pytest.raises(ValueError):
105: (8)                    list(t1f._tokenize(b'1234 (this (string) is unterminated\\)', True))
106: (4)                with pytest.raises(ValueError):
107: (8)                    list(t1f._tokenize(b'/Foo<01234', True))
108: (4)                with pytest.raises(ValueError):
109: (8)                    list(t1f._tokenize(b'/Foo<01234abcg>/Bar', True))
110: (0)            def test_overprecision():
111: (4)                filename = os.path.join(os.path.dirname(__file__), 'cmr10.pfb')
112: (4)                font = t1f.Type1Font(filename)
113: (4)                slanted = font.transform({'slant': .167})
114: (4)                lines = slanted.parts[0].decode('ascii').splitlines()
115: (4)                matrix, = [line[line.index('[')+1:line.index(']')]
116: (15)                           for line in lines if '/FontMatrix' in line]
117: (4)                angle, = [word
118: (14)                          for line in lines if '/ItalicAngle' in line
119: (14)                          for word in line.split() if word[0] in '-0123456789']
120: (4)                assert matrix == '0.001 0 0.000167 0.001 0 0'
121: (4)                assert angle == '-9.4809'
122: (0)            def test_encrypt_decrypt_roundtrip():
123: (4)                data = b'this is my plaintext \0\1\2\3'
124: (4)                encrypted = t1f.Type1Font._encrypt(data, 'eexec')
125: (4)                decrypted = t1f.Type1Font._decrypt(encrypted, 'eexec')
126: (4)                assert encrypted != decrypted
127: (4)                assert data == decrypted
```

File 229 - test_usetex.py:

```
1: (0)              from tempfile import TemporaryFile
2: (0)              import numpy as np
3: (0)              from packaging.version import parse as parse_version
4: (0)              import pytest
5: (0)              import matplotlib as mpl
6: (0)              from matplotlib import dviread
7: (0)              from matplotlib.testing import _has_tex_package
8: (0)              from matplotlib.testing.decorators import check_figures_equal,
image_comparison
9: (0)              from matplotlib.testing._markers import needs_usetex
10: (0)             import matplotlib.pyplot as plt
11: (0)             pytestmark = needs_usetex
12: (0)             @image_comparison(
13: (4)                 baseline_images=['test_usetex'],
14: (4)                 extensions=['pdf', 'png'],
15: (4)                 style="mpl20")
16: (0)             def test_usetex():
17: (4)                 mpl.rcParams['text.usetex'] = True
18: (4)                 fig, ax = plt.subplots()
19: (4)                 kwargs = {"verticalalignment": "baseline", "size": 24,
20: (14)                          "bbox": dict(pad=0, edgecolor="k", facecolor="none")}
21: (4)                 ax.text(0.2, 0.7,
22: (12)                        r'\LaTeX\ $\left[\int\limits_e^{2e}'
23: (12)                        r'\sqrt\frac{\log^3 x}{x}\,\mathrm{d}x \right\}$',
24: (12)                        **kwargs)
25: (4)                 ax.text(0.2, 0.3, "lg", **kwargs)
26: (4)                 ax.text(0.4, 0.3, r"$\frac{1}{2}\pi$", **kwargs)
27: (4)                 ax.text(0.6, 0.3, "$p^{3^A}$", **kwargs)
28: (4)                 ax.text(0.8, 0.3, "$p_{3_2}$", **kwargs)
29: (4)                 for x in {t.get_position()[0] for t in ax.texts}:
30: (8)                     ax.axvline(x)
31: (4)                 for y in {t.get_position()[1] for t in ax.texts}:
32: (8)                     ax.axhline(y)
33: (4)                 ax.set_axis_off()
34: (0)             @check_figures_equal()
35: (0)             def test_empty(fig_test, fig_ref):
36: (4)                 mpl.rcParams['text.usetex'] = True
37: (4)                 fig_test.text(.5, .5, "% a comment")
38: (0)             @check_figures_equal()
39: (0)             def test_unicode_minus(fig_test, fig_ref):
```

```
 40: (4)                 mpl.rcParams['text.usetex'] = True
 41: (4)                 fig_test.text(.5, .5, "$-$")
 42: (4)                 fig_ref.text(.5, .5, "\N{MINUS SIGN}")
 43: (0)             def test_mathdefault():
 44: (4)                 plt.rcParams["axes.formatter.use_mathtext"] = True
 45: (4)                 fig = plt.figure()
 46: (4)                 fig.add_subplot().set_xlim(-1, 1)
 47: (4)                 mpl.rcParams['text.usetex'] = True
 48: (4)                 fig.canvas.draw()
 49: (0)             @image_comparison(['eqnarray.png'])
 50: (0)             def test_multiline_eqnarray():
 51: (4)                 text = (
 52: (8)                     r'\begin{eqnarray*}'
 53: (8)                     r'foo\\'
 54: (8)                     r'bar\\'
 55: (8)                     r'baz\\'
 56: (8)                     r'\end{eqnarray*}'
 57: (4)                 )
 58: (4)                 fig = plt.figure(figsize=(1, 1))
 59: (4)                 fig.text(0.5, 0.5, text, usetex=True,
 60: (13)                         horizontalalignment='center', verticalalignment='center')
 61: (0)             @pytest.mark.parametrize("fontsize", [8, 10, 12])
 62: (0)             def test_minus_no_descent(fontsize):
 63: (4)                 mpl.style.use("mpl20")
 64: (4)                 mpl.rcParams['font.size'] = fontsize
 65: (4)                 heights = {}
 66: (4)                 fig = plt.figure()
 67: (4)                 for vals in [(1,), (-1,), (-1, 1)]:
 68: (8)                     fig.clear()
 69: (8)                     for x in vals:
 70: (12)                        fig.text(.5, .5, f"${x}$", usetex=True)
 71: (8)                     fig.canvas.draw()
 72: (8)                     heights[vals] = ((np.array(fig.canvas.buffer_rgba())[..., 0] != 255)
 73: (25)                                     .any(axis=1).sum())
 74: (4)                 assert len({*heights.values()}) == 1
 75: (0)             @pytest.mark.parametrize('pkg', ['xcolor', 'chemformula'])
 76: (0)             def test_usetex_packages(pkg):
 77: (4)                 if not _has_tex_package(pkg):
 78: (8)                     pytest.skip(f'{pkg} is not available')
 79: (4)                 mpl.rcParams['text.usetex'] = True
 80: (4)                 fig = plt.figure()
 81: (4)                 text = fig.text(0.5, 0.5, "Some text 0123456789")
 82: (4)                 fig.canvas.draw()
 83: (4)                 mpl.rcParams['text.latex.preamble'] = (
 84: (8)                     r'\PassOptionsToPackage{dvipsnames}{xcolor}\usepackage{%s}' % pkg)
 85: (4)                 fig = plt.figure()
 86: (4)                 text2 = fig.text(0.5, 0.5, "Some text 0123456789")
 87: (4)                 fig.canvas.draw()
 88: (4)                 np.testing.assert_array_equal(text2.get_window_extent(),
 89: (34)                                              text.get_window_extent())
 90: (0)             @pytest.mark.parametrize(
 91: (4)                 "preamble",
 92: (4)                 [r"\usepackage[full]{textcomp}", r"\usepackage{underscore}"],
 93: (0)             )
 94: (0)             def test_latex_pkg_already_loaded(preamble):
 95: (4)                 plt.rcParams["text.latex.preamble"] = preamble
 96: (4)                 fig = plt.figure()
 97: (4)                 fig.text(.5, .5, "hello, world", usetex=True)
 98: (4)                 fig.canvas.draw()
 99: (0)             def test_usetex_with_underscore():
100: (4)                 plt.rcParams["text.usetex"] = True
101: (4)                 df = {"a_b": range(5)[::-1], "c": range(5)}
102: (4)                 fig, ax = plt.subplots()
103: (4)                 ax.plot("c", "a_b", data=df)
104: (4)                 ax.legend()
105: (4)                 ax.text(0, 0, "foo_bar", usetex=True)
106: (4)                 plt.draw()
107: (0)             @pytest.mark.flaky(reruns=3)  # Tends to hit a TeX cache lock on AppVeyor.
108: (0)             @pytest.mark.parametrize("fmt", ["pdf", "svg"])
```

```
109: (0)            def test_missing_psfont(fmt, monkeypatch):
110: (4)                """An error is raised if a TeX font lacks a Type-1 equivalent"""
111: (4)                monkeypatch.setattr(
112: (8)                    dviread.PsfontsMap, '__getitem__',
113: (8)                    lambda self, k: dviread.PsFont(
114: (12)                       texname=b'texfont', psname=b'Some Font',
115: (12)                       effects=None, encoding=None, filename=None))
116: (4)                mpl.rcParams['text.usetex'] = True
117: (4)                fig, ax = plt.subplots()
118: (4)                ax.text(0.5, 0.5, 'hello')
119: (4)                with TemporaryFile() as tmpfile, pytest.raises(ValueError):
120: (8)                    fig.savefig(tmpfile, format=fmt)
121: (0)            try:
122: (4)                _old_gs_version = mpl._get_executable_info('gs').version <
parse_version('9.55')
123: (0)            except mpl.ExecutableNotFoundError:
124: (4)                _old_gs_version = True
125: (0)            @image_comparison(baseline_images=['rotation'], extensions=['eps', 'pdf',
'png', 'svg'],
126: (18)                             style='mpl20', tol=3.91 if _old_gs_version else 0)
127: (0)            def test_rotation():
128: (4)                mpl.rcParams['text.usetex'] = True
129: (4)                fig = plt.figure()
130: (4)                ax = fig.add_axes([0, 0, 1, 1])
131: (4)                ax.set(xlim=[-0.5, 5], xticks=[], ylim=[-0.5, 3], yticks=[],
frame_on=False)
132: (4)                text = {val: val[0] for val in ['top', 'center', 'bottom', 'left',
'right']}
133: (4)                text['baseline'] = 'B'
134: (4)                text['center_baseline'] = 'C'
135: (4)                for i, va in enumerate(['top', 'center', 'bottom', 'baseline',
'center_baseline']):
136: (8)                    for j, ha in enumerate(['left', 'center', 'right']):
137: (12)                       for k, angle in enumerate([0, 90, 180, 270]):
138: (16)                           k //= 2
139: (16)                           x = i + k / 2
140: (16)                           y = j + k / 2
141: (16)                           ax.plot(x, y, '+', c=f'C{k}', markersize=20,
markeredgewidth=0.5)
142: (16)                           ax.text(x, y, f"$\\mathrm{{My {text[ha]}{text[va]}
{angle}}}$",
143: (24)                                   rotation=angle, horizontalalignment=ha,
verticalalignment=va)
```

File 230 - test_widgets.py:

```
1: (0)              import functools
2: (0)              import io
3: (0)              from unittest import mock
4: (0)              import matplotlib as mpl
5: (0)              from matplotlib.backend_bases import MouseEvent
6: (0)              import matplotlib.colors as mcolors
7: (0)              import matplotlib.widgets as widgets
8: (0)              import matplotlib.pyplot as plt
9: (0)              from matplotlib.testing.decorators import check_figures_equal,
image_comparison
10: (0)             from matplotlib.testing.widgets import (click_and_drag, do_event, get_ax,
11: (40)                                                    mock_event, noop)
12: (0)             import numpy as np
13: (0)             from numpy.testing import assert_allclose
14: (0)             import pytest
15: (0)             @pytest.fixture
16: (0)             def ax():
17: (4)                 return get_ax()
18: (0)             def test_save_blitted_widget_as_pdf():
19: (4)                 from matplotlib.widgets import CheckButtons, RadioButtons
20: (4)                 from matplotlib.cbook import _get_running_interactive_framework
```

```
 21: (4)                if _get_running_interactive_framework() not in ['headless', None]:
 22: (8)                    pytest.xfail("Callback exceptions are not raised otherwise.")
 23: (4)                fig, ax = plt.subplots(
 24: (8)                    nrows=2, ncols=2, figsize=(5, 2), width_ratios=[1, 2]
 25: (4)                )
 26: (4)                default_rb = RadioButtons(ax[0, 0], ['Apples', 'Oranges'])
 27: (4)                styled_rb = RadioButtons(
 28: (8)                    ax[0, 1], ['Apples', 'Oranges'],
 29: (8)                    label_props={'color': ['red', 'orange'],
 30: (21)                                 'fontsize': [16, 20]},
 31: (8)                    radio_props={'edgecolor': ['red', 'orange'],
 32: (21)                                 'facecolor': ['mistyrose', 'peachpuff']}
 33: (4)                )
 34: (4)                default_cb = CheckButtons(ax[1, 0], ['Apples', 'Oranges'],
 35: (30)                                          actives=[True, True])
 36: (4)                styled_cb = CheckButtons(
 37: (8)                    ax[1, 1], ['Apples', 'Oranges'],
 38: (8)                    actives=[True, True],
 39: (8)                    label_props={'color': ['red', 'orange'],
 40: (21)                                 'fontsize': [16, 20]},
 41: (8)                    frame_props={'edgecolor': ['red', 'orange'],
 42: (21)                                 'facecolor': ['mistyrose', 'peachpuff']},
 43: (8)                    check_props={'color': ['darkred', 'darkorange']}
 44: (4)                )
 45: (4)                ax[0, 0].set_title('Default')
 46: (4)                ax[0, 1].set_title('Stylized')
 47: (4)                fig.canvas.draw()
 48: (4)                with io.BytesIO() as result_after:
 49: (8)                    fig.savefig(result_after, format='pdf')
 50: (0)            @pytest.mark.parametrize('kwargs', [
 51: (4)                dict(),
 52: (4)                dict(useblit=True, button=1),
 53: (4)                dict(minspanx=10, minspany=10, spancoords='pixels'),
 54: (4)                dict(props=dict(fill=True)),
 55: (0)            ])
 56: (0)            def test_rectangle_selector(ax, kwargs):
 57: (4)                onselect = mock.Mock(spec=noop, return_value=None)
 58: (4)                tool = widgets.RectangleSelector(ax, onselect, **kwargs)
 59: (4)                do_event(tool, 'press', xdata=100, ydata=100, button=1)
 60: (4)                do_event(tool, 'onmove', xdata=199, ydata=199, button=1)
 61: (4)                do_event(tool, 'release', xdata=250, ydata=250, button=1)
 62: (4)                if kwargs.get('drawtype', None) not in ['line', 'none']:
 63: (8)                    assert_allclose(tool.geometry,
 64: (24)                                   [[100., 100, 199, 199, 100],
 65: (25)                                    [100, 199, 199, 100, 100]],
 66: (24)                                   err_msg=tool.geometry)
 67: (4)                onselect.assert_called_once()
 68: (4)                (epress, erelease), kwargs = onselect.call_args
 69: (4)                assert epress.xdata == 100
 70: (4)                assert epress.ydata == 100
 71: (4)                assert erelease.xdata == 199
 72: (4)                assert erelease.ydata == 199
 73: (4)                assert kwargs == {}
 74: (0)            @pytest.mark.parametrize('spancoords', ['data', 'pixels'])
 75: (0)            @pytest.mark.parametrize('minspanx, x1', [[0, 10], [1, 10.5], [1, 11]])
 76: (0)            @pytest.mark.parametrize('minspany, y1', [[0, 10], [1, 10.5], [1, 11]])
 77: (0)            def test_rectangle_minspan(ax, spancoords, minspanx, x1, minspany, y1):
 78: (4)                onselect = mock.Mock(spec=noop, return_value=None)
 79: (4)                x0, y0 = (10, 10)
 80: (4)                if spancoords == 'pixels':
 81: (8)                    minspanx, minspany = (ax.transData.transform((x1, y1)) -
 82: (30)                                          ax.transData.transform((x0, y0)))
 83: (4)                tool = widgets.RectangleSelector(ax, onselect, interactive=True,
 84: (37)                                                spancoords=spancoords,
 85: (37)                                                minspanx=minspanx, minspany=minspany)
 86: (4)                click_and_drag(tool, start=(x0, x1), end=(y0, y1))
 87: (4)                assert not tool._selection_completed
 88: (4)                onselect.assert_not_called()
 89: (4)                click_and_drag(tool, start=(20, 20), end=(30, 30))
```

```
 90: (4)                assert tool._selection_completed
 91: (4)                onselect.assert_called_once()
 92: (4)                onselect.reset_mock()
 93: (4)                click_and_drag(tool, start=(x0, y0), end=(x1, y1))
 94: (4)                assert not tool._selection_completed
 95: (4)                onselect.assert_called_once()
 96: (4)                (epress, erelease), kwargs = onselect.call_args
 97: (4)                assert epress.xdata == x0
 98: (4)                assert epress.ydata == y0
 99: (4)                assert erelease.xdata == x1
100: (4)                assert erelease.ydata == y1
101: (4)                assert kwargs == {}
102: (0)            def test_deprecation_selector_visible_attribute(ax):
103: (4)                tool = widgets.RectangleSelector(ax, lambda *args: None)
104: (4)                assert tool.get_visible()
105: (4)                with pytest.warns(mpl.MatplotlibDeprecationWarning,
106: (22)                                  match="was deprecated in Matplotlib 3.8"):
107: (8)                    tool.visible
108: (0)            @pytest.mark.parametrize('drag_from_anywhere, new_center',
109: (25)                                    [[True, (60, 75)],
110: (26)                                     [False, (30, 20)]])
111: (0)            def test_rectangle_drag(ax, drag_from_anywhere, new_center):
112: (4)                tool = widgets.RectangleSelector(ax, onselect=noop, interactive=True,
113: (37)                                                 drag_from_anywhere=drag_from_anywhere)
114: (4)                click_and_drag(tool, start=(0, 10), end=(100, 120))
115: (4)                assert tool.center == (50, 65)
116: (4)                click_and_drag(tool, start=(25, 15), end=(35, 25))
117: (4)                assert tool.center == new_center
118: (4)                click_and_drag(tool, start=(175, 185), end=(185, 195))
119: (4)                assert tool.center == (180, 190)
120: (0)            def test_rectangle_selector_set_props_handle_props(ax):
121: (4)                tool = widgets.RectangleSelector(ax, onselect=noop, interactive=True,
122: (37)                                                 props=dict(facecolor='b', alpha=0.2),
123: (37)                                                 handle_props=dict(alpha=0.5))
124: (4)                click_and_drag(tool, start=(0, 10), end=(100, 120))
125: (4)                artist = tool._selection_artist
126: (4)                assert artist.get_facecolor() == mcolors.to_rgba('b', alpha=0.2)
127: (4)                tool.set_props(facecolor='r', alpha=0.3)
128: (4)                assert artist.get_facecolor() == mcolors.to_rgba('r', alpha=0.3)
129: (4)                for artist in tool._handles_artists:
130: (8)                    assert artist.get_markeredgecolor() == 'black'
131: (8)                    assert artist.get_alpha() == 0.5
132: (4)                tool.set_handle_props(markeredgecolor='r', alpha=0.3)
133: (4)                for artist in tool._handles_artists:
134: (8)                    assert artist.get_markeredgecolor() == 'r'
135: (8)                    assert artist.get_alpha() == 0.3
136: (0)            def test_rectangle_resize(ax):
137: (4)                tool = widgets.RectangleSelector(ax, onselect=noop, interactive=True)
138: (4)                click_and_drag(tool, start=(0, 10), end=(100, 120))
139: (4)                assert tool.extents == (0.0, 100.0, 10.0, 120.0)
140: (4)                extents = tool.extents
141: (4)                xdata, ydata = extents[1], extents[3]
142: (4)                xdata_new, ydata_new = xdata + 10, ydata + 5
143: (4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new))
144: (4)                assert tool.extents == (extents[0], xdata_new, extents[2], ydata_new)
145: (4)                extents = tool.extents
146: (4)                xdata, ydata = extents[1], extents[2] + (extents[3] - extents[2]) / 2
147: (4)                xdata_new, ydata_new = xdata + 10, ydata
148: (4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new))
149: (4)                assert tool.extents == (extents[0], xdata_new, extents[2], extents[3])
150: (4)                extents = tool.extents
151: (4)                xdata, ydata = extents[0], extents[2] + (extents[3] - extents[2]) / 2
152: (4)                xdata_new, ydata_new = xdata + 15, ydata
153: (4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new))
154: (4)                assert tool.extents == (xdata_new, extents[1], extents[2], extents[3])
155: (4)                extents = tool.extents
156: (4)                xdata, ydata = extents[0], extents[2]
157: (4)                xdata_new, ydata_new = xdata + 20, ydata + 25
158: (4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new))
```

```
159: (4)                assert tool.extents == (xdata_new, extents[1], ydata_new, extents[3])
160: (0)            def test_rectangle_add_state(ax):
161: (4)                tool = widgets.RectangleSelector(ax, onselect=noop, interactive=True)
162: (4)                click_and_drag(tool, start=(70, 65), end=(125, 130))
163: (4)                with pytest.raises(ValueError):
164: (8)                    tool.add_state('unsupported_state')
165: (4)                with pytest.raises(ValueError):
166: (8)                    tool.add_state('clear')
167: (4)                tool.add_state('move')
168: (4)                tool.add_state('square')
169: (4)                tool.add_state('center')
170: (0)            @pytest.mark.parametrize('add_state', [True, False])
171: (0)            def test_rectangle_resize_center(ax, add_state):
172: (4)                tool = widgets.RectangleSelector(ax, onselect=noop, interactive=True)
173: (4)                click_and_drag(tool, start=(70, 65), end=(125, 130))
174: (4)                assert tool.extents == (70.0, 125.0, 65.0, 130.0)
175: (4)                if add_state:
176: (8)                    tool.add_state('center')
177: (8)                    use_key = None
178: (4)                else:
179: (8)                    use_key = 'control'
180: (4)                extents = tool.extents
181: (4)                xdata, ydata = extents[1], extents[3]
182: (4)                xdiff, ydiff = 10, 5
183: (4)                xdata_new, ydata_new = xdata + xdiff, ydata + ydiff
184: (4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new),
185: (19)                               key=use_key)
186: (4)                assert tool.extents == (extents[0] - xdiff, xdata_new,
187: (28)                                       extents[2] - ydiff, ydata_new)
188: (4)                extents = tool.extents
189: (4)                xdata, ydata = extents[1], extents[2] + (extents[3] - extents[2]) / 2
190: (4)                xdiff = 10
191: (4)                xdata_new, ydata_new = xdata + xdiff, ydata
192: (4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new),
193: (19)                               key=use_key)
194: (4)                assert tool.extents == (extents[0] - xdiff, xdata_new,
195: (28)                                       extents[2], extents[3])
196: (4)                extents = tool.extents
197: (4)                xdata, ydata = extents[1], extents[2] + (extents[3] - extents[2]) / 2
198: (4)                xdiff = -20
199: (4)                xdata_new, ydata_new = xdata + xdiff, ydata
200: (4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new),
201: (19)                               key=use_key)
202: (4)                assert tool.extents == (extents[0] - xdiff, xdata_new,
203: (28)                                       extents[2], extents[3])
204: (4)                extents = tool.extents
205: (4)                xdata, ydata = extents[0], extents[2] + (extents[3] - extents[2]) / 2
206: (4)                xdiff = 15
207: (4)                xdata_new, ydata_new = xdata + xdiff, ydata
208: (4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new),
209: (19)                               key=use_key)
210: (4)                assert tool.extents == (xdata_new, extents[1] - xdiff,
211: (28)                                       extents[2], extents[3])
212: (4)                extents = tool.extents
213: (4)                xdata, ydata = extents[0], extents[2] + (extents[3] - extents[2]) / 2
214: (4)                xdiff = -25
215: (4)                xdata_new, ydata_new = xdata + xdiff, ydata
216: (4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new),
217: (19)                               key=use_key)
218: (4)                assert tool.extents == (xdata_new, extents[1] - xdiff,
219: (28)                                       extents[2], extents[3])
220: (4)                extents = tool.extents
221: (4)                xdata, ydata = extents[0], extents[2]
222: (4)                xdiff, ydiff = 20, 25
223: (4)                xdata_new, ydata_new = xdata + xdiff, ydata + ydiff
224: (4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new),
225: (19)                               key=use_key)
226: (4)                assert tool.extents == (xdata_new, extents[1] - xdiff,
227: (28)                                       ydata_new, extents[3] - ydiff)
```

```
(0)            @pytest.mark.parametrize('add_state', [True, False])
(0)            def test_rectangle_resize_square(ax, add_state):
(4)                tool = widgets.RectangleSelector(ax, onselect=noop, interactive=True)
(4)                click_and_drag(tool, start=(70, 65), end=(120, 115))
(4)                assert tool.extents == (70.0, 120.0, 65.0, 115.0)
(4)                if add_state:
(8)                    tool.add_state('square')
(8)                    use_key = None
(4)                else:
(8)                    use_key = 'shift'
(4)                extents = tool.extents
(4)                xdata, ydata = extents[1], extents[3]
(4)                xdiff, ydiff = 10, 5
(4)                xdata_new, ydata_new = xdata + xdiff, ydata + ydiff
(4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new),
(19)                              key=use_key)
(4)                assert tool.extents == (extents[0], xdata_new,
(28)                                       extents[2], extents[3] + xdiff)
(4)                extents = tool.extents
(4)                xdata, ydata = extents[1], extents[2] + (extents[3] - extents[2]) / 2
(4)                xdiff = 10
(4)                xdata_new, ydata_new = xdata + xdiff, ydata
(4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new),
(19)                              key=use_key)
(4)                assert tool.extents == (extents[0], xdata_new,
(28)                                       extents[2], extents[3] + xdiff)
(4)                extents = tool.extents
(4)                xdata, ydata = extents[1], extents[2] + (extents[3] - extents[2]) / 2
(4)                xdiff = -20
(4)                xdata_new, ydata_new = xdata + xdiff, ydata
(4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new),
(19)                              key=use_key)
(4)                assert tool.extents == (extents[0], xdata_new,
(28)                                       extents[2], extents[3] + xdiff)
(4)                extents = tool.extents
(4)                xdata, ydata = extents[0], extents[2] + (extents[3] - extents[2]) / 2
(4)                xdiff = 15
(4)                xdata_new, ydata_new = xdata + xdiff, ydata
(4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new),
(19)                              key=use_key)
(4)                assert tool.extents == (xdata_new, extents[1],
(28)                                       extents[2], extents[3] - xdiff)
(4)                extents = tool.extents
(4)                xdata, ydata = extents[0], extents[2] + (extents[3] - extents[2]) / 2
(4)                xdiff = -25
(4)                xdata_new, ydata_new = xdata + xdiff, ydata
(4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new),
(19)                              key=use_key)
(4)                assert tool.extents == (xdata_new, extents[1],
(28)                                       extents[2], extents[3] - xdiff)
(4)                extents = tool.extents
(4)                xdata, ydata = extents[0], extents[2]
(4)                xdiff, ydiff = 20, 25
(4)                xdata_new, ydata_new = xdata + xdiff, ydata + ydiff
(4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new),
(19)                              key=use_key)
(4)                assert tool.extents == (extents[0] + ydiff, extents[1],
(28)                                       ydata_new, extents[3])
(0)            def test_rectangle_resize_square_center(ax):
(4)                tool = widgets.RectangleSelector(ax, onselect=noop, interactive=True)
(4)                click_and_drag(tool, start=(70, 65), end=(120, 115))
(4)                tool.add_state('square')
(4)                tool.add_state('center')
(4)                assert_allclose(tool.extents, (70.0, 120.0, 65.0, 115.0))
(4)                extents = tool.extents
(4)                xdata, ydata = extents[1], extents[3]
(4)                xdiff, ydiff = 10, 5
(4)                xdata_new, ydata_new = xdata + xdiff, ydata + ydiff
(4)                click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new))
```

```
(4)                 assert_allclose(tool.extents, (extents[0] - xdiff, xdata_new,
(35)                                                extents[2] - xdiff, extents[3] + xdiff))
(4)                 extents = tool.extents
(4)                 xdata, ydata = extents[1], extents[2] + (extents[3] - extents[2]) / 2
(4)                 xdiff = 10
(4)                 xdata_new, ydata_new = xdata + xdiff, ydata
(4)                 click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new))
(4)                 assert_allclose(tool.extents, (extents[0] - xdiff, xdata_new,
(35)                                                extents[2] - xdiff, extents[3] + xdiff))
(4)                 extents = tool.extents
(4)                 xdata, ydata = extents[1], extents[2] + (extents[3] - extents[2]) / 2
(4)                 xdiff = -20
(4)                 xdata_new, ydata_new = xdata + xdiff, ydata
(4)                 click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new))
(4)                 assert_allclose(tool.extents, (extents[0] - xdiff, xdata_new,
(35)                                                extents[2] - xdiff, extents[3] + xdiff))
(4)                 extents = tool.extents
(4)                 xdata, ydata = extents[0], extents[2] + (extents[3] - extents[2]) / 2
(4)                 xdiff = 5
(4)                 xdata_new, ydata_new = xdata + xdiff, ydata
(4)                 click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new))
(4)                 assert_allclose(tool.extents, (xdata_new, extents[1] - xdiff,
(35)                                                extents[2] + xdiff, extents[3] - xdiff))
(4)                 extents = tool.extents
(4)                 xdata, ydata = extents[0], extents[2] + (extents[3] - extents[2]) / 2
(4)                 xdiff = -25
(4)                 xdata_new, ydata_new = xdata + xdiff, ydata
(4)                 click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new))
(4)                 assert_allclose(tool.extents, (xdata_new, extents[1] - xdiff,
(35)                                                extents[2] + xdiff, extents[3] - xdiff))
(4)                 extents = tool.extents
(4)                 xdata, ydata = extents[0], extents[2]
(4)                 xdiff, ydiff = 20, 25
(4)                 xdata_new, ydata_new = xdata + xdiff, ydata + ydiff
(4)                 click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new))
(4)                 assert_allclose(tool.extents, (extents[0] + ydiff, extents[1] - ydiff,
(35)                                                ydata_new, extents[3] - ydiff))
(0)             @pytest.mark.parametrize('selector_class',
(25)                                     [widgets.RectangleSelector, widgets.EllipseSelector])
(0)             def test_rectangle_rotate(ax, selector_class):
(4)                 tool = selector_class(ax, onselect=noop, interactive=True)
(4)                 click_and_drag(tool, start=(100, 100), end=(130, 140))
(4)                 assert tool.extents == (100, 130, 100, 140)
(4)                 assert len(tool._state) == 0
(4)                 do_event(tool, 'on_key_press', key='r')
(4)                 assert tool._state == {'rotate'}
(4)                 assert len(tool._state) == 1
(4)                 click_and_drag(tool, start=(130, 140), end=(120, 145))
(4)                 do_event(tool, 'on_key_press', key='r')
(4)                 assert len(tool._state) == 0
(4)                 assert tool.extents == (100, 130, 100, 140)
(4)                 assert_allclose(tool.rotation, 25.56, atol=0.01)
(4)                 tool.rotation = 45
(4)                 assert tool.rotation == 45
(4)                 assert_allclose(tool.corners,
(20)                                np.array([[118.53, 139.75, 111.46, 90.25],
(30)                                          [95.25, 116.46, 144.75, 123.54]]), atol=0.01)
(4)                 click_and_drag(tool, start=(110, 145), end=(110, 160))
(4)                 assert_allclose(tool.extents, (100, 139.75, 100, 151.82), atol=0.01)
(4)                 if selector_class == widgets.RectangleSelector:
(8)                     with pytest.raises(ValueError):
(12)                        tool._selection_artist.rotation_point = 'unvalid_value'
(0)             def test_rectangle_add_remove_set(ax):
(4)                 tool = widgets.RectangleSelector(ax, onselect=noop, interactive=True)
(4)                 click_and_drag(tool, start=(100, 100), end=(130, 140))
(4)                 assert tool.extents == (100, 130, 100, 140)
(4)                 assert len(tool._state) == 0
(4)                 for state in ['rotate', 'square', 'center']:
(8)                     tool.add_state(state)
```

```
366: (8)                    assert len(tool._state) == 1
367: (8)                    tool.remove_state(state)
368: (8)                    assert len(tool._state) == 0
369: (0)            @pytest.mark.parametrize('use_data_coordinates', [False, True])
370: (0)            def test_rectangle_resize_square_center_aspect(ax, use_data_coordinates):
371: (4)                ax.set_aspect(0.8)
372: (4)                tool = widgets.RectangleSelector(ax, onselect=noop, interactive=True,
373: (37)
use_data_coordinates=use_data_coordinates)
374: (4)                click_and_drag(tool, start=(70, 65), end=(120, 115))
375: (4)                assert tool.extents == (70.0, 120.0, 65.0, 115.0)
376: (4)                tool.add_state('square')
377: (4)                tool.add_state('center')
378: (4)                if use_data_coordinates:
379: (8)                    extents = tool.extents
380: (8)                    xdata, ydata, width = extents[1], extents[3], extents[1] - extents[0]
381: (8)                    xdiff, ycenter = 10,  extents[2] + (extents[3] - extents[2]) / 2
382: (8)                    xdata_new, ydata_new = xdata + xdiff, ydata
383: (8)                    ychange = width / 2 + xdiff
384: (8)                    click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new))
385: (8)                    assert_allclose(tool.extents, [extents[0] - xdiff, xdata_new,
386: (39)                                                   ycenter - ychange, ycenter + ychange])
387: (4)                else:
388: (8)                    extents = tool.extents
389: (8)                    xdata, ydata = extents[1], extents[3]
390: (8)                    xdiff = 10
391: (8)                    xdata_new, ydata_new = xdata + xdiff, ydata
392: (8)                    ychange = xdiff * 1 / tool._aspect_ratio_correction
393: (8)                    click_and_drag(tool, start=(xdata, ydata), end=(xdata_new, ydata_new))
394: (8)                    assert_allclose(tool.extents, [extents[0] - xdiff, xdata_new,
395: (39)                                                   46.25, 133.75])
396: (0)            def test_ellipse(ax):
397: (4)                """For ellipse, test out the key modifiers"""
398: (4)                tool = widgets.EllipseSelector(ax, onselect=noop,
399: (35)                                               grab_range=10, interactive=True)
400: (4)                tool.extents = (100, 150, 100, 150)
401: (4)                click_and_drag(tool, start=(125, 125), end=(145, 145))
402: (4)                assert tool.extents == (120, 170, 120, 170)
403: (4)                click_and_drag(tool, start=(100, 100), end=(125, 125), key='control')
404: (4)                assert tool.extents == (75, 125, 75, 125)
405: (4)                click_and_drag(tool, start=(10, 10), end=(35, 30), key='shift')
406: (4)                extents = [int(e) for e in tool.extents]
407: (4)                assert extents == [10, 35, 10, 35]
408: (4)                click_and_drag(tool, start=(100, 100), end=(125, 130), key='ctrl+shift')
409: (4)                extents = [int(e) for e in tool.extents]
410: (4)                assert extents == [70, 130, 70, 130]
411: (4)                assert tool.geometry.shape == (2, 73)
412: (4)                assert_allclose(tool.geometry[:, 0], [70., 100])
413: (0)            def test_rectangle_handles(ax):
414: (4)                tool = widgets.RectangleSelector(ax, onselect=noop,
415: (37)                                                grab_range=10,
416: (37)                                                interactive=True,
417: (37)                                                handle_props={'markerfacecolor': 'r',
418: (51)                                                              'markeredgecolor': 'b'})
419: (4)                tool.extents = (100, 150, 100, 150)
420: (4)                assert_allclose(tool.corners, ((100, 150, 150, 100), (100, 100, 150,
150)))
421: (4)                assert tool.extents == (100, 150, 100, 150)
422: (4)                assert_allclose(tool.edge_centers,
423: (20)                               ((100, 125.0, 150, 125.0), (125.0, 100, 125.0, 150)))
424: (4)                assert tool.extents == (100, 150, 100, 150)
425: (4)                click_and_drag(tool, start=(100, 100), end=(120, 120))
426: (4)                assert tool.extents == (120, 150, 120, 150)
427: (4)                click_and_drag(tool, start=(132, 132), end=(120, 120))
428: (4)                assert tool.extents == (108, 138, 108, 138)
429: (4)                click_and_drag(tool, start=(10, 10), end=(100, 100))
430: (4)                assert tool.extents == (10, 100, 10, 100)
431: (4)                assert mcolors.same_color(
432: (8)                    tool._corner_handles.artists[0].get_markerfacecolor(), 'r')
```

```
433: (4)                assert mcolors.same_color(
434: (8)                    tool._corner_handles.artists[0].get_markeredgecolor(), 'b')
435: (0)            @pytest.mark.parametrize('interactive', [True, False])
436: (0)            def test_rectangle_selector_onselect(ax, interactive):
437: (4)                onselect = mock.Mock(spec=noop, return_value=None)
438: (4)                tool = widgets.RectangleSelector(ax, onselect, interactive=interactive)
439: (4)                click_and_drag(tool, start=(100, 110), end=(150, 120))
440: (4)                onselect.assert_called_once()
441: (4)                assert tool.extents == (100.0, 150.0, 110.0, 120.0)
442: (4)                onselect.reset_mock()
443: (4)                click_and_drag(tool, start=(10, 100), end=(10, 100))
444: (4)                onselect.assert_called_once()
445: (0)            @pytest.mark.parametrize('ignore_event_outside', [True, False])
446: (0)            def test_rectangle_selector_ignore_outside(ax, ignore_event_outside):
447: (4)                onselect = mock.Mock(spec=noop, return_value=None)
448: (4)                tool = widgets.RectangleSelector(ax, onselect,
449: (37)
ignore_event_outside=ignore_event_outside)
450: (4)                click_and_drag(tool, start=(100, 110), end=(150, 120))
451: (4)                onselect.assert_called_once()
452: (4)                assert tool.extents == (100.0, 150.0, 110.0, 120.0)
453: (4)                onselect.reset_mock()
454: (4)                click_and_drag(tool, start=(150, 150), end=(160, 160))
455: (4)                if ignore_event_outside:
456: (8)                    onselect.assert_not_called()
457: (8)                    assert tool.extents == (100.0, 150.0, 110.0, 120.0)
458: (4)                else:
459: (8)                    onselect.assert_called_once()
460: (8)                    assert tool.extents == (150.0, 160.0, 150.0, 160.0)
461: (0)            @pytest.mark.parametrize('orientation, onmove_callback, kwargs', [
462: (4)                ('horizontal', False, dict(minspan=10, useblit=True)),
463: (4)                ('vertical', True, dict(button=1)),
464: (4)                ('horizontal', False, dict(props=dict(fill=True))),
465: (4)                ('horizontal', False, dict(interactive=True)),
466: (0)            ])
467: (0)            def test_span_selector(ax, orientation, onmove_callback, kwargs):
468: (4)                onselect = mock.Mock(spec=noop, return_value=None)
469: (4)                onmove = mock.Mock(spec=noop, return_value=None)
470: (4)                if onmove_callback:
471: (8)                    kwargs['onmove_callback'] = onmove
472: (4)                ax.set_aspect("auto")
473: (4)                tax = ax.twinx()
474: (4)                tool = widgets.SpanSelector(ax, onselect, orientation, **kwargs)
475: (4)                do_event(tool, 'press', xdata=100, ydata=100, button=1)
476: (4)                do_event(tool, 'onmove', xdata=199, ydata=199, button=1)
477: (4)                do_event(tool, 'release', xdata=250, ydata=250, button=1)
478: (4)                onselect.assert_called_once_with(100, 199)
479: (4)                if onmove_callback:
480: (8)                    onmove.assert_called_once_with(100, 199)
481: (0)            @pytest.mark.parametrize('interactive', [True, False])
482: (0)            def test_span_selector_onselect(ax, interactive):
483: (4)                onselect = mock.Mock(spec=noop, return_value=None)
484: (4)                tool = widgets.SpanSelector(ax, onselect, 'horizontal',
485: (32)                                            interactive=interactive)
486: (4)                click_and_drag(tool, start=(100, 100), end=(150, 100))
487: (4)                onselect.assert_called_once()
488: (4)                assert tool.extents == (100, 150)
489: (4)                onselect.reset_mock()
490: (4)                click_and_drag(tool, start=(10, 100), end=(10, 100))
491: (4)                onselect.assert_called_once()
492: (0)            @pytest.mark.parametrize('ignore_event_outside', [True, False])
493: (0)            def test_span_selector_ignore_outside(ax, ignore_event_outside):
494: (4)                onselect = mock.Mock(spec=noop, return_value=None)
495: (4)                onmove = mock.Mock(spec=noop, return_value=None)
496: (4)                tool = widgets.SpanSelector(ax, onselect, 'horizontal',
497: (32)                                            onmove_callback=onmove,
498: (32)                                            ignore_event_outside=ignore_event_outside)
499: (4)                click_and_drag(tool, start=(100, 100), end=(125, 125))
500: (4)                onselect.assert_called_once()
```

```
(4)                 onmove.assert_called_once()
(4)                 assert tool.extents == (100, 125)
(4)                 onselect.reset_mock()
(4)                 onmove.reset_mock()
(4)                 click_and_drag(tool, start=(150, 150), end=(160, 160))
(4)                 if ignore_event_outside:
(8)                     onselect.assert_not_called()
(8)                     onmove.assert_not_called()
(8)                     assert tool.extents == (100, 125)
(4)                 else:
(8)                     onselect.assert_called_once()
(8)                     onmove.assert_called_once()
(8)                     assert tool.extents == (150, 160)
(0)             @pytest.mark.parametrize('drag_from_anywhere', [True, False])
(0)             def test_span_selector_drag(ax, drag_from_anywhere):
(4)                 tool = widgets.SpanSelector(ax, onselect=noop, direction='horizontal',
(32)                                            interactive=True,
(32)                                            drag_from_anywhere=drag_from_anywhere)
(4)                 click_and_drag(tool, start=(10, 10), end=(100, 120))
(4)                 assert tool.extents == (10, 100)
(4)                 click_and_drag(tool, start=(25, 15), end=(35, 25))
(4)                 if drag_from_anywhere:
(8)                     assert tool.extents == (20, 110)
(4)                 else:
(8)                     assert tool.extents == (25, 35)
(4)                 click_and_drag(tool, start=(175, 185), end=(185, 195))
(4)                 assert tool.extents == (175, 185)
(0)             def test_span_selector_direction(ax):
(4)                 tool = widgets.SpanSelector(ax, onselect=noop, direction='horizontal',
(32)                                            interactive=True)
(4)                 assert tool.direction == 'horizontal'
(4)                 assert tool._edge_handles.direction == 'horizontal'
(4)                 with pytest.raises(ValueError):
(8)                     tool = widgets.SpanSelector(ax, onselect=noop,
(36)                                                direction='invalid_direction')
(4)                 tool.direction = 'vertical'
(4)                 assert tool.direction == 'vertical'
(4)                 assert tool._edge_handles.direction == 'vertical'
(4)                 with pytest.raises(ValueError):
(8)                     tool.direction = 'invalid_string'
(0)             def test_span_selector_set_props_handle_props(ax):
(4)                 tool = widgets.SpanSelector(ax, onselect=noop, direction='horizontal',
(32)                                            interactive=True,
(32)                                            props=dict(facecolor='b', alpha=0.2),
(32)                                            handle_props=dict(alpha=0.5))
(4)                 click_and_drag(tool, start=(0, 10), end=(100, 120))
(4)                 artist = tool._selection_artist
(4)                 assert artist.get_facecolor() == mcolors.to_rgba('b', alpha=0.2)
(4)                 tool.set_props(facecolor='r', alpha=0.3)
(4)                 assert artist.get_facecolor() == mcolors.to_rgba('r', alpha=0.3)
(4)                 for artist in tool._handles_artists:
(8)                     assert artist.get_color() == 'b'
(8)                     assert artist.get_alpha() == 0.5
(4)                 tool.set_handle_props(color='r', alpha=0.3)
(4)                 for artist in tool._handles_artists:
(8)                     assert artist.get_color() == 'r'
(8)                     assert artist.get_alpha() == 0.3
(0)             @pytest.mark.parametrize('selector', ['span', 'rectangle'])
(0)             def test_selector_clear(ax, selector):
(4)                 kwargs = dict(ax=ax, onselect=noop, interactive=True)
(4)                 if selector == 'span':
(8)                     Selector = widgets.SpanSelector
(8)                     kwargs['direction'] = 'horizontal'
(4)                 else:
(8)                     Selector = widgets.RectangleSelector
(4)                 tool = Selector(**kwargs)
(4)                 click_and_drag(tool, start=(10, 10), end=(100, 120))
(4)                 click_and_drag(tool, start=(130, 130), end=(130, 130))
(4)                 assert not tool._selection_completed
```

```
(4)                 kwargs['ignore_event_outside'] = True
(4)                 tool = Selector(**kwargs)
(4)                 assert tool.ignore_event_outside
(4)                 click_and_drag(tool, start=(10, 10), end=(100, 120))
(4)                 click_and_drag(tool, start=(130, 130), end=(130, 130))
(4)                 assert tool._selection_completed
(4)                 do_event(tool, 'on_key_press', key='escape')
(4)                 assert not tool._selection_completed
(0)             @pytest.mark.parametrize('selector', ['span', 'rectangle'])
(0)             def test_selector_clear_method(ax, selector):
(4)                 if selector == 'span':
(8)                     tool = widgets.SpanSelector(ax, onselect=noop, direction='horizontal',
(36)                                                 interactive=True,
(36)                                                 ignore_event_outside=True)
(4)                 else:
(8)                     tool = widgets.RectangleSelector(ax, onselect=noop, interactive=True)
(4)                 click_and_drag(tool, start=(10, 10), end=(100, 120))
(4)                 assert tool._selection_completed
(4)                 assert tool.get_visible()
(4)                 if selector == 'span':
(8)                     assert tool.extents == (10, 100)
(4)                 tool.clear()
(4)                 assert not tool._selection_completed
(4)                 assert not tool.get_visible()
(4)                 click_and_drag(tool, start=(10, 10), end=(50, 120))
(4)                 assert tool._selection_completed
(4)                 assert tool.get_visible()
(4)                 if selector == 'span':
(8)                     assert tool.extents == (10, 50)
(0)             def test_span_selector_add_state(ax):
(4)                 tool = widgets.SpanSelector(ax, noop, 'horizontal',
(32)                                            interactive=True)
(4)                 with pytest.raises(ValueError):
(8)                     tool.add_state('unsupported_state')
(4)                 with pytest.raises(ValueError):
(8)                     tool.add_state('center')
(4)                 with pytest.raises(ValueError):
(8)                     tool.add_state('square')
(4)                 tool.add_state('move')
(0)             def test_tool_line_handle(ax):
(4)                 positions = [20, 30, 50]
(4)                 tool_line_handle = widgets.ToolLineHandles(ax, positions, 'horizontal',
(47)                                                           useblit=False)
(4)                 for artist in tool_line_handle.artists:
(8)                     assert not artist.get_animated()
(8)                     assert not artist.get_visible()
(4)                 tool_line_handle.set_visible(True)
(4)                 tool_line_handle.set_animated(True)
(4)                 for artist in tool_line_handle.artists:
(8)                     assert artist.get_animated()
(8)                     assert artist.get_visible()
(4)                 assert tool_line_handle.positions == positions
(0)             @pytest.mark.parametrize('direction', ("horizontal", "vertical"))
(0)             def test_span_selector_bound(direction):
(4)                 fig, ax = plt.subplots(1, 1)
(4)                 ax.plot([10, 20], [10, 30])
(4)                 ax.figure.canvas.draw()
(4)                 x_bound = ax.get_xbound()
(4)                 y_bound = ax.get_ybound()
(4)                 tool = widgets.SpanSelector(ax, print, direction, interactive=True)
(4)                 assert ax.get_xbound() == x_bound
(4)                 assert ax.get_ybound() == y_bound
(4)                 bound = x_bound if direction == 'horizontal' else y_bound
(4)                 assert tool._edge_handles.positions == list(bound)
(4)                 press_data = (10.5, 11.5)
(4)                 move_data = (11, 13)  # Updating selector is done in onmove
(4)                 release_data = move_data
(4)                 click_and_drag(tool, start=press_data, end=move_data)
(4)                 assert ax.get_xbound() == x_bound
```

```
639: (4)                assert ax.get_ybound() == y_bound
640: (4)                index = 0 if direction == 'horizontal' else 1
641: (4)                handle_positions = [press_data[index], release_data[index]]
642: (4)                assert tool._edge_handles.positions == handle_positions
643: (0)            @pytest.mark.backend('QtAgg', skip_on_importerror=True)
644: (0)            def test_span_selector_animated_artists_callback():
645: (4)                """Check that the animated artists changed in callbacks are updated."""
646: (4)                x = np.linspace(0, 2 * np.pi, 100)
647: (4)                values = np.sin(x)
648: (4)                fig, ax = plt.subplots()
649: (4)                ln, = ax.plot(x, values, animated=True)
650: (4)                ln2, = ax.plot([], animated=True)
651: (4)                plt.pause(0.1)
652: (4)                ax.draw_artist(ln)
653: (4)                fig.canvas.blit(fig.bbox)
654: (4)                def mean(vmin, vmax):
655: (8)                    indmin, indmax = np.searchsorted(x, (vmin, vmax))
656: (8)                    v = values[indmin:indmax].mean()
657: (8)                    ln2.set_data(x, np.full_like(x, v))
658: (4)                span = widgets.SpanSelector(ax, mean, direction='horizontal',
659: (32)                                            onmove_callback=mean,
660: (32)                                            interactive=True,
661: (32)                                            drag_from_anywhere=True,
662: (32)                                            useblit=True)
663: (4)                press_data = [1, 2]
664: (4)                move_data = [2, 2]
665: (4)                do_event(span, 'press', xdata=press_data[0], ydata=press_data[1],
button=1)
666: (4)                do_event(span, 'onmove', xdata=move_data[0], ydata=move_data[1], button=1)
667: (4)                assert span._get_animated_artists() == (ln, ln2)
668: (4)                assert ln.stale is False
669: (4)                assert ln2.stale
670: (4)                assert_allclose(ln2.get_ydata(), 0.9547335049088455)
671: (4)                span.update()
672: (4)                assert ln2.stale is False
673: (4)                press_data = [4, 0]
674: (4)                move_data = [5, 2]
675: (4)                release_data = [5, 2]
676: (4)                do_event(span, 'press', xdata=press_data[0], ydata=press_data[1],
button=1)
677: (4)                do_event(span, 'onmove', xdata=move_data[0], ydata=move_data[1], button=1)
678: (4)                assert ln.stale is False
679: (4)                assert ln2.stale
680: (4)                assert_allclose(ln2.get_ydata(), -0.9424150707548072)
681: (4)                do_event(span, 'release', xdata=release_data[0],
682: (13)                        ydata=release_data[1], button=1)
683: (4)                assert ln2.stale is False
684: (0)            def test_snapping_values_span_selector(ax):
685: (4)                def onselect(*args):
686: (8)                    pass
687: (4)                tool = widgets.SpanSelector(ax, onselect, direction='horizontal',)
688: (4)                snap_function = tool._snap
689: (4)                snap_values = np.linspace(0, 5, 11)
690: (4)                values = np.array([-0.1, 0.1, 0.2, 0.5, 0.6, 0.7, 0.9, 4.76, 5.0, 5.5])
691: (4)                expect = np.array([00.0, 0.0, 0.0, 0.5, 0.5, 0.5, 1.0, 5.00, 5.0, 5.0])
692: (4)                values = snap_function(values, snap_values)
693: (4)                assert_allclose(values, expect)
694: (0)            def test_span_selector_snap(ax):
695: (4)                def onselect(vmin, vmax):
696: (8)                    ax._got_onselect = True
697: (4)                snap_values = np.arange(50) * 4
698: (4)                tool = widgets.SpanSelector(ax, onselect, direction='horizontal',
699: (32)                                            snap_values=snap_values)
700: (4)                tool.extents = (17, 35)
701: (4)                assert tool.extents == (16, 36)
702: (4)                tool.snap_values = None
703: (4)                assert tool.snap_values is None
704: (4)                tool.extents = (17, 35)
705: (4)                assert tool.extents == (17, 35)
```

```
(0)  def test_span_selector_extents(ax):
(4)      tool = widgets.SpanSelector(
(8)          ax, lambda a, b: None, "horizontal", ignore_event_outside=True
(8)          )
(4)      tool.extents = (5, 10)
(4)      assert tool.extents == (5, 10)
(4)      assert tool._selection_completed
(4)      press_data = (12, 14)
(4)      release_data = (20, 14)
(4)      click_and_drag(tool, start=press_data, end=release_data)
(4)      assert tool.extents == (5, 10)
(0)  @pytest.mark.parametrize('kwargs', [
(4)      dict(),
(4)      dict(useblit=False, props=dict(color='red')),
(4)      dict(useblit=True, button=1),
(0)  ])
(0)  def test_lasso_selector(ax, kwargs):
(4)      onselect = mock.Mock(spec=noop, return_value=None)
(4)      tool = widgets.LassoSelector(ax, onselect, **kwargs)
(4)      do_event(tool, 'press', xdata=100, ydata=100, button=1)
(4)      do_event(tool, 'onmove', xdata=125, ydata=125, button=1)
(4)      do_event(tool, 'release', xdata=150, ydata=150, button=1)
(4)      onselect.assert_called_once_with([(100, 100), (125, 125), (150, 150)])
(0)  def test_lasso_selector_set_props(ax):
(4)      onselect = mock.Mock(spec=noop, return_value=None)
(4)      tool = widgets.LassoSelector(ax, onselect, props=dict(color='b',
alpha=0.2))
(4)      artist = tool._selection_artist
(4)      assert mcolors.same_color(artist.get_color(), 'b')
(4)      assert artist.get_alpha() == 0.2
(4)      tool.set_props(color='r', alpha=0.3)
(4)      assert mcolors.same_color(artist.get_color(), 'r')
(4)      assert artist.get_alpha() == 0.3
(0)  def test_lasso_set_props(ax):
(4)      onselect = mock.Mock(spec=noop, return_value=None)
(4)      tool = widgets.Lasso(ax, (100, 100), onselect)
(4)      line = tool.line
(4)      assert mcolors.same_color(line.get_color(), 'black')
(4)      assert line.get_linestyle() == '-'
(4)      assert line.get_lw() == 2
(4)      tool = widgets.Lasso(ax, (100, 100), onselect, props=dict(
(8)          linestyle='-', color='darkblue', alpha=0.2, lw=1))
(4)      line = tool.line
(4)      assert mcolors.same_color(line.get_color(), 'darkblue')
(4)      assert line.get_alpha() == 0.2
(4)      assert line.get_lw() == 1
(4)      assert line.get_linestyle() == '-'
(4)      line.set_color('r')
(4)      line.set_alpha(0.3)
(4)      assert mcolors.same_color(line.get_color(), 'r')
(4)      assert line.get_alpha() == 0.3
(0)  def test_CheckButtons(ax):
(4)      labels = ('a', 'b', 'c')
(4)      check = widgets.CheckButtons(ax, labels, (True, False, True))
(4)      assert check.get_status() == [True, False, True]
(4)      check.set_active(0)
(4)      assert check.get_status() == [False, False, True]
(4)      assert check.get_checked_labels() == ['c']
(4)      check.clear()
(4)      assert check.get_status() == [False, False, False]
(4)      assert check.get_checked_labels() == []
(4)      for invalid_index in [-1, len(labels), len(labels)+5]:
(8)          with pytest.raises(ValueError):
(12)             check.set_active(index=invalid_index)
(4)      for invalid_value in ['invalid', -1]:
(8)          with pytest.raises(TypeError):
(12)             check.set_active(1, state=invalid_value)
(4)      cid = check.on_clicked(lambda: None)
(4)      check.disconnect(cid)
```

```
774: (0)            @pytest.mark.parametrize("toolbar", ["none", "toolbar2", "toolmanager"])
775: (0)            def test_TextBox(ax, toolbar):
776: (4)                plt.rcParams._set("toolbar", toolbar)
777: (4)                submit_event = mock.Mock(spec=noop, return_value=None)
778: (4)                text_change_event = mock.Mock(spec=noop, return_value=None)
779: (4)                tool = widgets.TextBox(ax, '')
780: (4)                tool.on_submit(submit_event)
781: (4)                tool.on_text_change(text_change_event)
782: (4)                assert tool.text == ''
783: (4)                do_event(tool, '_click')
784: (4)                tool.set_val('x**2')
785: (4)                assert tool.text == 'x**2'
786: (4)                assert text_change_event.call_count == 1
787: (4)                tool.begin_typing()
788: (4)                tool.stop_typing()
789: (4)                assert submit_event.call_count == 2
790: (4)                do_event(tool, '_click', xdata=.5, ydata=.5)  # Ensure the click is in the
axes.
791: (4)                do_event(tool, '_keypress', key='+')
792: (4)                do_event(tool, '_keypress', key='5')
793: (4)                assert text_change_event.call_count == 3
794: (0)            def test_RadioButtons(ax):
795: (4)                radio = widgets.RadioButtons(ax, ('Radio 1', 'Radio 2', 'Radio 3'))
796: (4)                radio.set_active(1)
797: (4)                assert radio.value_selected == 'Radio 2'
798: (4)                assert radio.index_selected == 1
799: (4)                radio.clear()
800: (4)                assert radio.value_selected == 'Radio 1'
801: (4)                assert radio.index_selected == 0
802: (0)            @image_comparison(['check_radio_buttons.png'], style='mpl20',
remove_text=True)
803: (0)            def test_check_radio_buttons_image():
804: (4)                ax = get_ax()
805: (4)                fig = ax.figure
806: (4)                fig.subplots_adjust(left=0.3)
807: (4)                rax1 = fig.add_axes((0.05, 0.7, 0.2, 0.15))
808: (4)                rb1 = widgets.RadioButtons(rax1, ('Radio 1', 'Radio 2', 'Radio 3'))
809: (4)                rax2 = fig.add_axes((0.05, 0.5, 0.2, 0.15))
810: (4)                cb1 = widgets.CheckButtons(rax2, ('Check 1', 'Check 2', 'Check 3'),
811: (31)                                           (False, True, True))
812: (4)                rax3 = fig.add_axes((0.05, 0.3, 0.2, 0.15))
813: (4)                rb3 = widgets.RadioButtons(
814: (8)                    rax3, ('Radio 1', 'Radio 2', 'Radio 3'),
815: (8)                    label_props={'fontsize': [8, 12, 16],
816: (21)                                 'color': ['red', 'green', 'blue']},
817: (8)                    radio_props={'edgecolor': ['red', 'green', 'blue'],
818: (21)                                 'facecolor': ['mistyrose', 'palegreen', 'lightblue']})
819: (4)                rax4 = fig.add_axes((0.05, 0.1, 0.2, 0.15))
820: (4)                cb4 = widgets.CheckButtons(
821: (8)                    rax4, ('Check 1', 'Check 2', 'Check 3'), (False, True, True),
822: (8)                    label_props={'fontsize': [8, 12, 16],
823: (21)                                 'color': ['red', 'green', 'blue']},
824: (8)                    frame_props={'edgecolor': ['red', 'green', 'blue'],
825: (21)                                 'facecolor': ['mistyrose', 'palegreen', 'lightblue']},
826: (8)                    check_props={'color': ['red', 'green', 'blue']})
827: (0)            @check_figures_equal(extensions=["png"])
828: (0)            def test_radio_buttons(fig_test, fig_ref):
829: (4)                widgets.RadioButtons(fig_test.subplots(), ["tea", "coffee"])
830: (4)                ax = fig_ref.add_subplot(xticks=[], yticks=[])
831: (4)                ax.scatter([.15, .15], [2/3, 1/3], transform=ax.transAxes,
832: (15)                           s=(plt.rcParams["font.size"] / 2) ** 2, c=["C0", "none"])
833: (4)                ax.text(.25, 2/3, "tea", transform=ax.transAxes, va="center")
834: (4)                ax.text(.25, 1/3, "coffee", transform=ax.transAxes, va="center")
835: (0)            @check_figures_equal(extensions=['png'])
836: (0)            def test_radio_buttons_props(fig_test, fig_ref):
837: (4)                label_props = {'color': ['red'], 'fontsize': [24]}
838: (4)                radio_props = {'facecolor': 'green', 'edgecolor': 'blue', 'linewidth': 2}
839: (4)                widgets.RadioButtons(fig_ref.subplots(), ['tea', 'coffee'],
840: (25)                                     label_props=label_props, radio_props=radio_props)
```

```
841: (4)                cb = widgets.RadioButtons(fig_test.subplots(), ['tea', 'coffee'])
842: (4)                cb.set_label_props(label_props)
843: (4)                cb.set_radio_props({**radio_props, 's': (24 / 2)**2})
844: (0)            def test_radio_button_active_conflict(ax):
845: (4)                with pytest.warns(UserWarning,
846: (22)                                  match=r'Both the \*activecolor\* parameter'):
847: (8)                    rb = widgets.RadioButtons(ax, ['tea', 'coffee'], activecolor='red',
848: (34)                                              radio_props={'facecolor': 'green'})
849: (4)                assert mcolors.same_color(rb._buttons.get_facecolor(), ['green', 'none'])
850: (0)            @check_figures_equal(extensions=['png'])
851: (0)            def test_radio_buttons_activecolor_change(fig_test, fig_ref):
852: (4)                widgets.RadioButtons(fig_ref.subplots(), ['tea', 'coffee'],
853: (25)                                    activecolor='green')
854: (4)                cb = widgets.RadioButtons(fig_test.subplots(), ['tea', 'coffee'],
855: (30)                                          activecolor='red')
856: (4)                cb.activecolor = 'green'
857: (0)            @check_figures_equal(extensions=["png"])
858: (0)            def test_check_buttons(fig_test, fig_ref):
859: (4)                widgets.CheckButtons(fig_test.subplots(), ["tea", "coffee"], [True, True])
860: (4)                ax = fig_ref.add_subplot(xticks=[], yticks=[])
861: (4)                ax.scatter([.15, .15], [2/3, 1/3], marker='s', transform=ax.transAxes,
862: (15)                           s=(plt.rcParams["font.size"] / 2) ** 2, c=["none", "none"])
863: (4)                ax.scatter([.15, .15], [2/3, 1/3], marker='x', transform=ax.transAxes,
864: (15)                           s=(plt.rcParams["font.size"] / 2) ** 2, c=["k", "k"])
865: (4)                ax.text(.25, 2/3, "tea", transform=ax.transAxes, va="center")
866: (4)                ax.text(.25, 1/3, "coffee", transform=ax.transAxes, va="center")
867: (0)            @check_figures_equal(extensions=['png'])
868: (0)            def test_check_button_props(fig_test, fig_ref):
869: (4)                label_props = {'color': ['red'], 'fontsize': [24]}
870: (4)                frame_props = {'facecolor': 'green', 'edgecolor': 'blue', 'linewidth': 2}
871: (4)                check_props = {'facecolor': 'red', 'linewidth': 2}
872: (4)                widgets.CheckButtons(fig_ref.subplots(), ['tea', 'coffee'], [True, True],
873: (25)                                    label_props=label_props, frame_props=frame_props,
874: (25)                                    check_props=check_props)
875: (4)                cb = widgets.CheckButtons(fig_test.subplots(), ['tea', 'coffee'],
876: (30)                                          [True, True])
877: (4)                cb.set_label_props(label_props)
878: (4)                cb.set_frame_props({**frame_props, 's': (24 / 2)**2})
879: (4)                check_props['edgecolor'] = check_props.pop('facecolor')
880: (4)                cb.set_check_props({**check_props, 's': (24 / 2)**2})
881: (0)            def test_slider_slidermin_slidermax_invalid():
882: (4)                fig, ax = plt.subplots()
883: (4)                with pytest.raises(ValueError):
884: (8)                    widgets.Slider(ax=ax, label='', valmin=0.0, valmax=24.0,
885: (23)                                  slidermin=10.0)
886: (4)                with pytest.raises(ValueError):
887: (8)                    widgets.Slider(ax=ax, label='', valmin=0.0, valmax=24.0,
888: (23)                                  slidermax=10.0)
889: (0)            def test_slider_slidermin_slidermax():
890: (4)                fig, ax = plt.subplots()
891: (4)                slider_ = widgets.Slider(ax=ax, label='', valmin=0.0, valmax=24.0,
892: (29)                                        valinit=5.0)
893: (4)                slider = widgets.Slider(ax=ax, label='', valmin=0.0, valmax=24.0,
894: (28)                                       valinit=1.0, slidermin=slider_)
895: (4)                assert slider.val == slider_.val
896: (4)                slider = widgets.Slider(ax=ax, label='', valmin=0.0, valmax=24.0,
897: (28)                                       valinit=10.0, slidermax=slider_)
898: (4)                assert slider.val == slider_.val
899: (0)            def test_slider_valmin_valmax():
900: (4)                fig, ax = plt.subplots()
901: (4)                slider = widgets.Slider(ax=ax, label='', valmin=0.0, valmax=24.0,
902: (28)                                       valinit=-10.0)
903: (4)                assert slider.val == slider.valmin
904: (4)                slider = widgets.Slider(ax=ax, label='', valmin=0.0, valmax=24.0,
905: (28)                                       valinit=25.0)
906: (4)                assert slider.val == slider.valmax
907: (0)            def test_slider_valstep_snapping():
908: (4)                fig, ax = plt.subplots()
909: (4)                slider = widgets.Slider(ax=ax, label='', valmin=0.0, valmax=24.0,
```

```
(28)                                    valinit=11.4, valstep=1)
(4)                 assert slider.val == 11
(4)                 slider = widgets.Slider(ax=ax, label='', valmin=0.0, valmax=24.0,
(28)                                    valinit=11.4, valstep=[0, 1, 5.5, 19.7])
(4)                 assert slider.val == 5.5
(0)             def test_slider_horizontal_vertical():
(4)                 fig, ax = plt.subplots()
(4)                 slider = widgets.Slider(ax=ax, label='', valmin=0, valmax=24,
(28)                                    valinit=12, orientation='horizontal')
(4)                 slider.set_val(10)
(4)                 assert slider.val == 10
(4)                 box = slider.poly.get_extents().transformed(ax.transAxes.inverted())
(4)                 assert_allclose(box.bounds, [0, .25, 10/24, .5])
(4)                 fig, ax = plt.subplots()
(4)                 slider = widgets.Slider(ax=ax, label='', valmin=0, valmax=24,
(28)                                    valinit=12, orientation='vertical')
(4)                 slider.set_val(10)
(4)                 assert slider.val == 10
(4)                 box = slider.poly.get_extents().transformed(ax.transAxes.inverted())
(4)                 assert_allclose(box.bounds, [.25, 0, .5, 10/24])
(0)             def test_slider_reset():
(4)                 fig, ax = plt.subplots()
(4)                 slider = widgets.Slider(ax=ax, label='', valmin=0, valmax=1, valinit=.5)
(4)                 slider.set_val(0.75)
(4)                 slider.reset()
(4)                 assert slider.val == 0.5
(0)             @pytest.mark.parametrize("orientation", ["horizontal", "vertical"])
(0)             def test_range_slider(orientation):
(4)                 if orientation == "vertical":
(8)                     idx = [1, 0, 3, 2]
(4)                 else:
(8)                     idx = [0, 1, 2, 3]
(4)                 fig, ax = plt.subplots()
(4)                 slider = widgets.RangeSlider(
(8)                     ax=ax, label="", valmin=0.0, valmax=1.0, orientation=orientation,
(8)                     valinit=[0.1, 0.34]
(4)                 )
(4)                 box = slider.poly.get_extents().transformed(ax.transAxes.inverted())
(4)                 assert_allclose(box.get_points().flatten()[idx], [0.1, 0.25, 0.34, 0.75])
(4)                 assert_allclose(slider.val, (0.1, 0.34))
(4)                 def handle_positions(slider):
(8)                     if orientation == "vertical":
(12)                        return [h.get_ydata()[0] for h in slider._handles]
(8)                     else:
(12)                        return [h.get_xdata()[0] for h in slider._handles]
(4)                 slider.set_val((0.4, 0.6))
(4)                 assert_allclose(slider.val, (0.4, 0.6))
(4)                 assert_allclose(handle_positions(slider), (0.4, 0.6))
(4)                 box = slider.poly.get_extents().transformed(ax.transAxes.inverted())
(4)                 assert_allclose(box.get_points().flatten()[idx], [0.4, .25, 0.6, .75])
(4)                 slider.set_val((0.2, 0.1))
(4)                 assert_allclose(slider.val, (0.1, 0.2))
(4)                 assert_allclose(handle_positions(slider), (0.1, 0.2))
(4)                 slider.set_val((-1, 10))
(4)                 assert_allclose(slider.val, (0, 1))
(4)                 assert_allclose(handle_positions(slider), (0, 1))
(4)                 slider.reset()
(4)                 assert_allclose(slider.val, (0.1, 0.34))
(4)                 assert_allclose(handle_positions(slider), (0.1, 0.34))
(0)             @pytest.mark.parametrize("orientation", ["horizontal", "vertical"])
(0)             def test_range_slider_same_init_values(orientation):
(4)                 if orientation == "vertical":
(8)                     idx = [1, 0, 3, 2]
(4)                 else:
(8)                     idx = [0, 1, 2, 3]
(4)                 fig, ax = plt.subplots()
(4)                 slider = widgets.RangeSlider(
(9)                      ax=ax, label="", valmin=0.0, valmax=1.0, orientation=orientation,
(9)                      valinit=[0, 0]
```

```
(5)                 )
(4)             box = slider.poly.get_extents().transformed(ax.transAxes.inverted())
(4)             assert_allclose(box.get_points().flatten()[idx], [0, 0.25, 0, 0.75])
(0)         def check_polygon_selector(event_sequence, expected_result, selections_count,
(27)                                    **kwargs):
(4)             """
(4)             Helper function to test Polygon Selector.
(4)             Parameters
(4)             ----------
(4)             event_sequence : list of tuples (etype, dict())
(8)                 A sequence of events to perform. The sequence is a list of tuples
(8)                 where the first element of the tuple is an etype (e.g., 'onmove',
(8)                 'press', etc.), and the second element of the tuple is a dictionary of
(9)                  the arguments for the event (e.g., xdata=5, key='shift', etc.).
(4)             expected_result : list of vertices (xdata, ydata)
(8)                 The list of vertices that are expected to result from the event
(8)                 sequence.
(4)             selections_count : int
(8)                 Wait for the tool to call its `onselect` function `selections_count`
(8)                 times, before comparing the result to the `expected_result`
(4)             **kwargs
(8)                 Keyword arguments are passed to PolygonSelector.
(4)             """
(4)             ax = get_ax()
(4)             onselect = mock.Mock(spec=noop, return_value=None)
(4)             tool = widgets.PolygonSelector(ax, onselect, **kwargs)
(4)             for (etype, event_args) in event_sequence:
(8)                 do_event(tool, etype, **event_args)
(4)             assert onselect.call_count == selections_count
(4)             assert onselect.call_args == ((expected_result, ), {})
(0)         def polygon_place_vertex(xdata, ydata):
(4)             return [('onmove', dict(xdata=xdata, ydata=ydata)),
(12)                    ('press', dict(xdata=xdata, ydata=ydata)),
(12)                    ('release', dict(xdata=xdata, ydata=ydata))]
(0)         def polygon_remove_vertex(xdata, ydata):
(4)             return [('onmove', dict(xdata=xdata, ydata=ydata)),
(12)                    ('press', dict(xdata=xdata, ydata=ydata, button=3)),
(12)                    ('release', dict(xdata=xdata, ydata=ydata, button=3))]
(0)         @pytest.mark.parametrize('draw_bounding_box', [False, True])
(0)         def test_polygon_selector(draw_bounding_box):
(4)             check_selector = functools.partial(
(8)                 check_polygon_selector, draw_bounding_box=draw_bounding_box)
(4)             expected_result = [(50, 50), (150, 50), (50, 150)]
(4)             event_sequence = [
(8)                 *polygon_place_vertex(50, 50),
(8)                 *polygon_place_vertex(150, 50),
(8)                 *polygon_place_vertex(50, 150),
(8)                 *polygon_place_vertex(50, 50),
(4)             ]
(4)             check_selector(event_sequence, expected_result, 1)
(4)             expected_result = [(75, 50), (150, 50), (50, 150)]
(4)             event_sequence = [
(8)                 *polygon_place_vertex(50, 50),
(8)                 *polygon_place_vertex(150, 50),
(8)                 ('on_key_press', dict(key='control')),
(8)                 ('onmove', dict(xdata=50, ydata=50)),
(8)                 ('press', dict(xdata=50, ydata=50)),
(8)                 ('onmove', dict(xdata=75, ydata=50)),
(8)                 ('release', dict(xdata=75, ydata=50)),
(8)                 ('on_key_release', dict(key='control')),
(8)                 *polygon_place_vertex(50, 150),
(8)                 *polygon_place_vertex(75, 50),
(4)             ]
(4)             check_selector(event_sequence, expected_result, 1)
(4)             expected_result = [(50, 75), (150, 75), (50, 150)]
(4)             event_sequence = [
(8)                 *polygon_place_vertex(50, 50),
(8)                 *polygon_place_vertex(150, 50),
(8)                 ('on_key_press', dict(key='shift')),
```

```
(8)                     ('onmove', dict(xdata=100, ydata=100)),
(8)                     ('press', dict(xdata=100, ydata=100)),
(8)                     ('onmove', dict(xdata=100, ydata=125)),
(8)                     ('release', dict(xdata=100, ydata=125)),
(8)                     ('on_key_release', dict(key='shift')),
(8)                     *polygon_place_vertex(50, 150),
(8)                     *polygon_place_vertex(50, 75),
(4)                 ]
(4)                 check_selector(event_sequence, expected_result, 1)
(4)                 expected_result = [(75, 50), (150, 50), (50, 150)]
(4)                 event_sequence = [
(8)                     *polygon_place_vertex(50, 50),
(8)                     *polygon_place_vertex(150, 50),
(8)                     *polygon_place_vertex(50, 150),
(8)                     *polygon_place_vertex(50, 50),
(8)                     ('onmove', dict(xdata=50, ydata=50)),
(8)                     ('press', dict(xdata=50, ydata=50)),
(8)                     ('onmove', dict(xdata=75, ydata=50)),
(8)                     ('release', dict(xdata=75, ydata=50)),
(4)                 ]
(4)                 check_selector(event_sequence, expected_result, 2)
(4)                 expected_result = [(75, 75), (175, 75), (75, 175)]
(4)                 event_sequence = [
(8)                     *polygon_place_vertex(50, 50),
(8)                     *polygon_place_vertex(150, 50),
(8)                     *polygon_place_vertex(50, 150),
(8)                     *polygon_place_vertex(50, 50),
(8)                     ('on_key_press', dict(key='shift')),
(8)                     ('onmove', dict(xdata=100, ydata=100)),
(8)                     ('press', dict(xdata=100, ydata=100)),
(8)                     ('onmove', dict(xdata=125, ydata=125)),
(8)                     ('release', dict(xdata=125, ydata=125)),
(8)                     ('on_key_release', dict(key='shift')),
(4)                 ]
(4)                 check_selector(event_sequence, expected_result, 2)
(4)                 expected_result = [(50, 50), (150, 50), (50, 150)]
(4)                 event_sequence = [
(8)                     ('on_key_press', dict(key='control')),
(8)                     ('onmove', dict(xdata=100, ydata=100)),
(8)                     ('press', dict(xdata=100, ydata=100)),
(8)                     ('onmove', dict(xdata=125, ydata=125)),
(8)                     ('release', dict(xdata=125, ydata=125)),
(8)                     ('on_key_release', dict(key='control')),
(8)                     ('on_key_press', dict(key='shift')),
(8)                     ('onmove', dict(xdata=100, ydata=100)),
(8)                     ('press', dict(xdata=100, ydata=100)),
(8)                     ('onmove', dict(xdata=125, ydata=125)),
(8)                     ('release', dict(xdata=125, ydata=125)),
(8)                     ('on_key_release', dict(key='shift')),
(8)                     *polygon_place_vertex(50, 50),
(8)                     *polygon_place_vertex(150, 50),
(8)                     *polygon_place_vertex(50, 150),
(8)                     *polygon_place_vertex(50, 50),
(4)                 ]
(4)                 check_selector(event_sequence, expected_result, 1)
(4)                 expected_result = [(50, 50), (150, 50), (50, 150)]
(4)                 event_sequence = [
(8)                     *polygon_place_vertex(50, 50),
(8)                     *polygon_place_vertex(250, 50),
(8)                     ('on_key_press', dict(key='escape')),
(8)                     ('on_key_release', dict(key='escape')),
(8)                     *polygon_place_vertex(50, 50),
(8)                     *polygon_place_vertex(150, 50),
(8)                     *polygon_place_vertex(50, 150),
(8)                     *polygon_place_vertex(50, 50),
(4)                 ]
(4)                 check_selector(event_sequence, expected_result, 1)
(0)             @pytest.mark.parametrize('draw_bounding_box', [False, True])
(0)             def test_polygon_selector_set_props_handle_props(ax, draw_bounding_box):
```

```
(4)             tool = widgets.PolygonSelector(ax, onselect=noop,
(35)                                           props=dict(color='b', alpha=0.2),
(35)                                           handle_props=dict(alpha=0.5),
(35)                                           draw_bounding_box=draw_bounding_box)
(4)             event_sequence = [
(8)                 *polygon_place_vertex(50, 50),
(8)                 *polygon_place_vertex(150, 50),
(8)                 *polygon_place_vertex(50, 150),
(8)                 *polygon_place_vertex(50, 50),
(4)             ]
(4)             for (etype, event_args) in event_sequence:
(8)                 do_event(tool, etype, **event_args)
(4)             artist = tool._selection_artist
(4)             assert artist.get_color() == 'b'
(4)             assert artist.get_alpha() == 0.2
(4)             tool.set_props(color='r', alpha=0.3)
(4)             assert artist.get_color() == 'r'
(4)             assert artist.get_alpha() == 0.3
(4)             for artist in tool._handles_artists:
(8)                 assert artist.get_color() == 'b'
(8)                 assert artist.get_alpha() == 0.5
(4)             tool.set_handle_props(color='r', alpha=0.3)
(4)             for artist in tool._handles_artists:
(8)                 assert artist.get_color() == 'r'
(8)                 assert artist.get_alpha() == 0.3
(0)         @check_figures_equal()
(0)         def test_rect_visibility(fig_test, fig_ref):
(4)             ax_test = fig_test.subplots()
(4)             _ = fig_ref.subplots()
(4)             tool = widgets.RectangleSelector(ax_test, onselect=noop,
(37)                                             props={'visible': False})
(4)             tool.extents = (0.2, 0.8, 0.3, 0.7)
(0)         @pytest.mark.parametrize('idx', [1, 2, 3])
(0)         @pytest.mark.parametrize('draw_bounding_box', [False, True])
(0)         def test_polygon_selector_remove(idx, draw_bounding_box):
(4)             verts = [(50, 50), (150, 50), (50, 150)]
(4)             event_sequence = [polygon_place_vertex(*verts[0]),
(22)                              polygon_place_vertex(*verts[1]),
(22)                              polygon_place_vertex(*verts[2]),
(22)                              polygon_place_vertex(*verts[0])]
(4)             event_sequence.insert(idx, polygon_place_vertex(200, 200))
(4)             event_sequence.append(polygon_remove_vertex(200, 200))
(4)             event_sequence = sum(event_sequence, [])
(4)             check_polygon_selector(event_sequence, verts, 2,
(27)                                   draw_bounding_box=draw_bounding_box)
(0)         @pytest.mark.parametrize('draw_bounding_box', [False, True])
(0)         def test_polygon_selector_remove_first_point(draw_bounding_box):
(4)             verts = [(50, 50), (150, 50), (50, 150)]
(4)             event_sequence = [
(8)                 *polygon_place_vertex(*verts[0]),
(8)                 *polygon_place_vertex(*verts[1]),
(8)                 *polygon_place_vertex(*verts[2]),
(8)                 *polygon_place_vertex(*verts[0]),
(8)                 *polygon_remove_vertex(*verts[0]),
(4)             ]
(4)             check_polygon_selector(event_sequence, verts[1:], 2,
(27)                                   draw_bounding_box=draw_bounding_box)
(0)         @pytest.mark.parametrize('draw_bounding_box', [False, True])
(0)         def test_polygon_selector_redraw(ax, draw_bounding_box):
(4)             verts = [(50, 50), (150, 50), (50, 150)]
(4)             event_sequence = [
(8)                 *polygon_place_vertex(*verts[0]),
(8)                 *polygon_place_vertex(*verts[1]),
(8)                 *polygon_place_vertex(*verts[2]),
(8)                 *polygon_place_vertex(*verts[0]),
(8)                 *polygon_remove_vertex(*verts[1]),
(8)                 *polygon_remove_vertex(*verts[2]),
(8)                 *polygon_place_vertex(*verts[1]),
(4)             ]
```

```
(4)             tool = widgets.PolygonSelector(ax, onselect=noop,
(35)                                           draw_bounding_box=draw_bounding_box)
(4)             for (etype, event_args) in event_sequence:
(8)                 do_event(tool, etype, **event_args)
(4)             assert tool.verts == verts[0:2]
(0)         @pytest.mark.parametrize('draw_bounding_box', [False, True])
(0)         @check_figures_equal(extensions=['png'])
(0)         def test_polygon_selector_verts_setter(fig_test, fig_ref, draw_bounding_box):
(4)             verts = [(0.1, 0.4), (0.5, 0.9), (0.3, 0.2)]
(4)             ax_test = fig_test.add_subplot()
(4)             tool_test = widgets.PolygonSelector(
(8)                 ax_test, onselect=noop, draw_bounding_box=draw_bounding_box)
(4)             tool_test.verts = verts
(4)             assert tool_test.verts == verts
(4)             ax_ref = fig_ref.add_subplot()
(4)             tool_ref = widgets.PolygonSelector(
(8)                 ax_ref, onselect=noop, draw_bounding_box=draw_bounding_box)
(4)             event_sequence = [
(8)                 *polygon_place_vertex(*verts[0]),
(8)                 *polygon_place_vertex(*verts[1]),
(8)                 *polygon_place_vertex(*verts[2]),
(8)                 *polygon_place_vertex(*verts[0]),
(4)             ]
(4)             for (etype, event_args) in event_sequence:
(8)                 do_event(tool_ref, etype, **event_args)
(0)         def test_polygon_selector_box(ax):
(4)             ax.set(xlim=(-10, 50), ylim=(-10, 50))
(4)             verts = [(20, 0), (0, 20), (20, 40), (40, 20)]
(4)             event_sequence = [
(8)                 *polygon_place_vertex(*verts[0]),
(8)                 *polygon_place_vertex(*verts[1]),
(8)                 *polygon_place_vertex(*verts[2]),
(8)                 *polygon_place_vertex(*verts[3]),
(8)                 *polygon_place_vertex(*verts[0]),
(4)             ]
(4)             tool = widgets.PolygonSelector(ax, onselect=noop, draw_bounding_box=True)
(4)             for (etype, event_args) in event_sequence:
(8)                 do_event(tool, etype, **event_args)
(4)             t = ax.transData
(4)             canvas = ax.figure.canvas
(4)             MouseEvent(
(8)                 "button_press_event", canvas, *t.transform((40, 40)), 1)._process()
(4)             MouseEvent(
(8)                 "motion_notify_event", canvas, *t.transform((20, 20)))._process()
(4)             MouseEvent(
(8)                 "button_release_event", canvas, *t.transform((20, 20)), 1)._process()
(4)             np.testing.assert_allclose(
(8)                 tool.verts, [(10, 0), (0, 10), (10, 20), (20, 10)])
(4)             MouseEvent(
(8)                 "button_press_event", canvas, *t.transform((10, 10)), 1)._process()
(4)             MouseEvent(
(8)                 "motion_notify_event", canvas, *t.transform((30, 30)))._process()
(4)             MouseEvent(
(8)                 "button_release_event", canvas, *t.transform((30, 30)), 1)._process()
(4)             np.testing.assert_allclose(
(8)                 tool.verts, [(30, 20), (20, 30), (30, 40), (40, 30)])
(4)             np.testing.assert_allclose(
(8)                 tool._box.extents, (20.0, 40.0, 20.0, 40.0))
(4)             MouseEvent(
(8)                 "button_press_event", canvas, *t.transform((30, 20)), 3)._process()
(4)             MouseEvent(
(8)                 "button_release_event", canvas, *t.transform((30, 20)), 3)._process()
(4)             np.testing.assert_allclose(
(8)                 tool.verts, [(20, 30), (30, 40), (40, 30)])
(4)             np.testing.assert_allclose(
(8)                 tool._box.extents, (20.0, 40.0, 30.0, 40.0))
(0)         def test_polygon_selector_clear_method(ax):
(4)             onselect = mock.Mock(spec=noop, return_value=None)
(4)             tool = widgets.PolygonSelector(ax, onselect)
```

```
1255: (4)               for result in ([(50, 50), (150, 50), (50, 150), (50, 50)],
1256: (19)                             [(50, 50), (100, 50), (50, 150), (50, 50)]):
1257: (8)                   for x, y in result:
1258: (12)                      for etype, event_args in polygon_place_vertex(x, y):
1259: (16)                          do_event(tool, etype, **event_args)
1260: (8)                   artist = tool._selection_artist
1261: (8)                   assert tool._selection_completed
1262: (8)                   assert tool.get_visible()
1263: (8)                   assert artist.get_visible()
1264: (8)                   np.testing.assert_equal(artist.get_xydata(), result)
1265: (8)                   assert onselect.call_args == ((result[:-1],), {})
1266: (8)                   tool.clear()
1267: (8)                   assert not tool._selection_completed
1268: (8)                   np.testing.assert_equal(artist.get_xydata(), [(0, 0)])
1269: (0)           @pytest.mark.parametrize("horizOn", [False, True])
1270: (0)           @pytest.mark.parametrize("vertOn", [False, True])
1271: (0)           def test_MultiCursor(horizOn, vertOn):
1272: (4)               (ax1, ax3) = plt.figure().subplots(2, sharex=True)
1273: (4)               ax2 = plt.figure().subplots()
1274: (4)               multi = widgets.MultiCursor(
1275: (8)                   None, (ax1, ax2), useblit=False, horizOn=horizOn, vertOn=vertOn
1276: (4)               )
1277: (4)               assert len(multi.vlines) == 2
1278: (4)               assert len(multi.hlines) == 2
1279: (4)               event = mock_event(ax1, xdata=.5, ydata=.25)
1280: (4)               multi.onmove(event)
1281: (4)               ax1.figure.canvas.draw()
1282: (4)               for l in multi.vlines:
1283: (8)                   assert l.get_xdata() == (.5, .5)
1284: (4)               for l in multi.hlines:
1285: (8)                   assert l.get_ydata() == (.25, .25)
1286: (4)               assert len([line for line in multi.vlines if line.get_visible()]) == (
1287: (8)                   2 if vertOn else 0)
1288: (4)               assert len([line for line in multi.hlines if line.get_visible()]) == (
1289: (8)                   2 if horizOn else 0)
1290: (4)               multi.horizOn = not multi.horizOn
1291: (4)               multi.vertOn = not multi.vertOn
1292: (4)               event = mock_event(ax1, xdata=.5, ydata=.25)
1293: (4)               multi.onmove(event)
1294: (4)               assert len([line for line in multi.vlines if line.get_visible()]) == (
1295: (8)                   0 if vertOn else 2)
1296: (4)               assert len([line for line in multi.hlines if line.get_visible()]) == (
1297: (8)                   0 if horizOn else 2)
1298: (4)               event = mock_event(ax3, xdata=.75, ydata=.75)
1299: (4)               multi.onmove(event)
1300: (4)               for l in multi.vlines:
1301: (8)                   assert l.get_xdata() == (.5, .5)
1302: (4)               for l in multi.hlines:
1303: (8)                   assert l.get_ydata() == (.25, .25)
```

File 231 - __init__.py:

```
1: (0)              """
2: (0)              Unstructured triangular grid functions.
3: (0)              """
4: (0)              from ._triangulation import Triangulation
5: (0)              from ._tricontour import TriContourSet, tricontour, tricontourf
6: (0)              from ._trifinder import TriFinder, TrapezoidMapTriFinder
7: (0)              from ._triinterpolate import (TriInterpolator, LinearTriInterpolator,
8: (30)                                           CubicTriInterpolator)
9: (0)              from ._tripcolor import tripcolor
10: (0)             from ._triplot import triplot
11: (0)             from ._trirefine import TriRefiner, UniformTriRefiner
12: (0)             from ._tritools import TriAnalyzer
13: (0)             __all__ = ["Triangulation",
14: (11)                       "TriContourSet", "tricontour", "tricontourf",
15: (11)                       "TriFinder", "TrapezoidMapTriFinder",
```

```
16: (11)                        "TriInterpolator", "LinearTriInterpolator", "CubicTriInterpolator",
17: (11)                        "tripcolor",
18: (11)                        "triplot",
19: (11)                        "TriRefiner", "UniformTriRefiner",
20: (11)                        "TriAnalyzer"]
```

File 232 - _triangulation.py:

```
1: (0)              import sys
2: (0)              import numpy as np
3: (0)              from matplotlib import _api
4: (0)              class Triangulation:
5: (4)                  """
6: (4)                  An unstructured triangular grid consisting of npoints points and
7: (4)                  ntri triangles.  The triangles can either be specified by the user
8: (4)                  or automatically generated using a Delaunay triangulation.
9: (4)                  Parameters
10: (4)                 ----------
11: (4)                 x, y : (npoints,) array-like
12: (8)                     Coordinates of grid points.
13: (4)                 triangles : (ntri, 3) array-like of int, optional
14: (8)                     For each triangle, the indices of the three points that make
15: (8)                     up the triangle, ordered in an anticlockwise manner.  If not
16: (8)                     specified, the Delaunay triangulation is calculated.
17: (4)                 mask : (ntri,) array-like of bool, optional
18: (8)                     Which triangles are masked out.
19: (4)                 Attributes
20: (4)                 ----------
21: (4)                 triangles : (ntri, 3) array of int
22: (8)                     For each triangle, the indices of the three points that make
23: (8)                     up the triangle, ordered in an anticlockwise manner. If you want to
24: (8)                     take the *mask* into account, use `get_masked_triangles` instead.
25: (4)                 mask : (ntri, 3) array of bool or None
26: (8)                     Masked out triangles.
27: (4)                 is_delaunay : bool
28: (8)                     Whether the Triangulation is a calculated Delaunay
29: (8)                     triangulation (where *triangles* was not specified) or not.
30: (4)                 Notes
31: (4)                 -----
32: (4)                 For a Triangulation to be valid it must not have duplicate points,
33: (4)                 triangles formed from colinear points, or overlapping triangles.
34: (4)                 """
35: (4)                 def __init__(self, x, y, triangles=None, mask=None):
36: (8)                     from matplotlib import _qhull
37: (8)                     self.x = np.asarray(x, dtype=np.float64)
38: (8)                     self.y = np.asarray(y, dtype=np.float64)
39: (8)                     if self.x.shape != self.y.shape or self.x.ndim != 1:
40: (12)                        raise ValueError("x and y must be equal-length 1D arrays, but "
41: (29)                                         f"found shapes {self.x.shape!r} and "
42: (29)                                         f"{self.y.shape!r}")
43: (8)                     self.mask = None
44: (8)                     self._edges = None
45: (8)                     self._neighbors = None
46: (8)                     self.is_delaunay = False
47: (8)                     if triangles is None:
48: (12)                        self.triangles, self._neighbors = _qhull.delaunay(x, y,
sys.flags.verbose)
49: (12)                        self.is_delaunay = True
50: (8)                     else:
51: (12)                        try:
52: (16)                            self.triangles = np.array(triangles, dtype=np.int32,
order='C')
53: (12)                        except ValueError as e:
54: (16)                            raise ValueError('triangles must be a (N, 3) int array, not '
55: (33)                                             f'{triangles!r}') from e
56: (12)                        if self.triangles.ndim != 2 or self.triangles.shape[1] != 3:
57: (16)                            raise ValueError(
```

```
 58: (20)                                'triangles must be a (N, 3) int array, but found shape '
 59: (20)                                f'{self.triangles.shape!r}')
 60: (12)                            if self.triangles.max() >= len(self.x):
 61: (16)                                raise ValueError(
 62: (20)                                    'triangles are indices into the points and must be in the
 '
 63: (20)                                    f'range 0 <= i < {len(self.x)} but found value '
 64: (20)                                    f'{self.triangles.max()}')
 65: (12)                            if self.triangles.min() < 0:
 66: (16)                                raise ValueError(
 67: (20)                                    'triangles are indices into the points and must be in the
 '
 68: (20)                                    f'range 0 <= i < {len(self.x)} but found value '
 69: (20)                                    f'{self.triangles.min()}')
 70: (8)                         self._cpp_triangulation = None
 71: (8)                         self._trifinder = None
 72: (8)                         self.set_mask(mask)
 73: (4)                     def calculate_plane_coefficients(self, z):
 74: (8)                         """
 75: (8)                         Calculate plane equation coefficients for all unmasked triangles from
 76: (8)                         the point (x, y) coordinates and specified z-array of shape (npoints).
 77: (8)                         The returned array has shape (npoints, 3) and allows z-value at (x, y)
 78: (8)                         position in triangle tri to be calculated using
 79: (8)                         ``z = array[tri, 0] * x  + array[tri, 1] * y + array[tri, 2]``.
 80: (8)                         """
 81: (8)                         return self.get_cpp_triangulation().calculate_plane_coefficients(z)
 82: (4)                     @property
 83: (4)                     def edges(self):
 84: (8)                         """
 85: (8)                         Return integer array of shape (nedges, 2) containing all edges of
 86: (8)                         non-masked triangles.
 87: (8)                         Each row defines an edge by its start point index and end point
 88: (8)                         index.  Each edge appears only once, i.e. for an edge between points
 89: (8)                         *i*  and *j*, there will only be either *(i, j)* or *(j, i)*.
 90: (8)                         """
 91: (8)                         if self._edges is None:
 92: (12)                            self._edges = self.get_cpp_triangulation().get_edges()
 93: (8)                         return self._edges
 94: (4)                     def get_cpp_triangulation(self):
 95: (8)                         """
 96: (8)                         Return the underlying C++ Triangulation object, creating it
 97: (8)                         if necessary.
 98: (8)                         """
 99: (8)                         from matplotlib import _tri
100: (8)                         if self._cpp_triangulation is None:
101: (12)                            self._cpp_triangulation = _tri.Triangulation(
102: (16)                                self.x, self.y, self.triangles,
103: (16)                                self.mask if self.mask is not None else (),
104: (16)                                self._edges if self._edges is not None else (),
105: (16)                                self._neighbors if self._neighbors is not None else (),
106: (16)                                not self.is_delaunay)
107: (8)                         return self._cpp_triangulation
108: (4)                     def get_masked_triangles(self):
109: (8)                         """
110: (8)                         Return an array of triangles taking the mask into account.
111: (8)                         """
112: (8)                         if self.mask is not None:
113: (12)                            return self.triangles[~self.mask]
114: (8)                         else:
115: (12)                            return self.triangles
116: (4)                     @staticmethod
117: (4)                     def get_from_args_and_kwargs(*args, **kwargs):
118: (8)                         """
119: (8)                         Return a Triangulation object from the args and kwargs, and
120: (8)                         the remaining args and kwargs with the consumed values removed.
121: (8)                         There are two alternatives: either the first argument is a
122: (8)                         Triangulation object, in which case it is returned, or the args
123: (8)                         and kwargs are sufficient to create a new Triangulation to
124: (8)                         return.  In the latter case, see Triangulation.__init__ for
```

```
(8)                     the possible args and kwargs.
(8)                     """
(8)                     if isinstance(args[0], Triangulation):
(12)                        triangulation, *args = args
(12)                        if 'triangles' in kwargs:
(16)                            _api.warn_external(
(20)                                "Passing the keyword 'triangles' has no effect when also "
(20)                                "passing a Triangulation")
(12)                        if 'mask' in kwargs:
(16)                            _api.warn_external(
(20)                                "Passing the keyword 'mask' has no effect when also "
(20)                                "passing a Triangulation")
(8)                     else:
(12)                        x, y, triangles, mask, args, kwargs = \
(16)                            Triangulation._extract_triangulation_params(args, kwargs)
(12)                        triangulation = Triangulation(x, y, triangles, mask)
(8)                     return triangulation, args, kwargs
(4)                 @staticmethod
(4)                 def _extract_triangulation_params(args, kwargs):
(8)                     x, y, *args = args
(8)                     triangles = kwargs.pop('triangles', None)
(8)                     from_args = False
(8)                     if triangles is None and args:
(12)                        triangles = args[0]
(12)                        from_args = True
(8)                     if triangles is not None:
(12)                        try:
(16)                            triangles = np.asarray(triangles, dtype=np.int32)
(12)                        except ValueError:
(16)                            triangles = None
(8)                     if triangles is not None and (triangles.ndim != 2 or
(38)                                                  triangles.shape[1] != 3):
(12)                        triangles = None
(8)                     if triangles is not None and from_args:
(12)                        args = args[1:]  # Consumed first item in args.
(8)                     mask = kwargs.pop('mask', None)
(8)                     return x, y, triangles, mask, args, kwargs
(4)                 def get_trifinder(self):
(8)                     """
(8)                     Return the default `matplotlib.tri.TriFinder` of this
(8)                     triangulation, creating it if necessary.  This allows the same
(8)                     TriFinder object to be easily shared.
(8)                     """
(8)                     if self._trifinder is None:
(12)                        from matplotlib.tri._trifinder import TrapezoidMapTriFinder
(12)                        self._trifinder = TrapezoidMapTriFinder(self)
(8)                     return self._trifinder
(4)                 @property
(4)                 def neighbors(self):
(8)                     """
(8)                     Return integer array of shape (ntri, 3) containing neighbor triangles.
(8)                     For each triangle, the indices of the three triangles that
(8)                     share the same edges, or -1 if there is no such neighboring
(8)                     triangle.  ``neighbors[i, j]`` is the triangle that is the neighbor
(8)                     to the edge from point index ``triangles[i, j]`` to point index
(8)                     ``triangles[i, (j+1)%3]``.
(8)                     """
(8)                     if self._neighbors is None:
(12)                        self._neighbors = self.get_cpp_triangulation().get_neighbors()
(8)                     return self._neighbors
(4)                 def set_mask(self, mask):
(8)                     """
(8)                     Set or clear the mask array.
(8)                     Parameters
(8)                     ----------
(8)                     mask : None or bool array of length ntri
(8)                     """
(8)                     if mask is None:
(12)                        self.mask = None
```

```
194: (8)                    else:
195: (12)                       self.mask = np.asarray(mask, dtype=bool)
196: (12)                       if self.mask.shape != (self.triangles.shape[0],):
197: (16)                           raise ValueError('mask array must have same length as '
198: (33)                                            'triangles array')
199: (8)                    if self._cpp_triangulation is not None:
200: (12)                       self._cpp_triangulation.set_mask(
201: (16)                           self.mask if self.mask is not None else ())
202: (8)                    self._edges = None
203: (8)                    self._neighbors = None
204: (8)                    if self._trifinder is not None:
205: (12)                       self._trifinder._initialize()
```

File 233 - _tricontour.py:

```
1: (0)              import numpy as np
2: (0)              from matplotlib import _docstring
3: (0)              from matplotlib.contour import ContourSet
4: (0)              from matplotlib.tri._triangulation import Triangulation
5: (0)              @_docstring.dedent_interpd
6: (0)              class TriContourSet(ContourSet):
7: (4)                  """
8: (4)                  Create and store a set of contour lines or filled regions for
9: (4)                  a triangular grid.
10: (4)                 This class is typically not instantiated directly by the user but by
11: (4)                 `~.Axes.tricontour` and `~.Axes.tricontourf`.
12: (4)                 %(contour_set_attributes)s
13: (4)                 """
14: (4)                 def __init__(self, ax, *args, **kwargs):
15: (8)                     """
16: (8)                     Draw triangular grid contour lines or filled regions,
17: (8)                     depending on whether keyword arg *filled* is False
18: (8)                     (default) or True.
19: (8)                     The first argument of the initializer must be an `~.axes.Axes`
20: (8)                     object.  The remaining arguments and keyword arguments
21: (8)                     are described in the docstring of `~.Axes.tricontour`.
22: (8)                     """
23: (8)                     super().__init__(ax, *args, **kwargs)
24: (4)                 def _process_args(self, *args, **kwargs):
25: (8)                     """
26: (8)                     Process args and kwargs.
27: (8)                     """
28: (8)                     if isinstance(args[0], TriContourSet):
29: (12)                        C = args[0]._contour_generator
30: (12)                        if self.levels is None:
31: (16)                            self.levels = args[0].levels
32: (12)                        self.zmin = args[0].zmin
33: (12)                        self.zmax = args[0].zmax
34: (12)                        self._mins = args[0]._mins
35: (12)                        self._maxs = args[0]._maxs
36: (8)                     else:
37: (12)                        from matplotlib import _tri
38: (12)                        tri, z = self._contour_args(args, kwargs)
39: (12)                        C = _tri.TriContourGenerator(tri.get_cpp_triangulation(), z)
40: (12)                        self._mins = [tri.x.min(), tri.y.min()]
41: (12)                        self._maxs = [tri.x.max(), tri.y.max()]
42: (8)                     self._contour_generator = C
43: (8)                     return kwargs
44: (4)                 def _contour_args(self, args, kwargs):
45: (8)                     tri, args, kwargs = Triangulation.get_from_args_and_kwargs(*args,
46: (67)                                                                               **kwargs)
47: (8)                     z, *args = args
48: (8)                     z = np.ma.asarray(z)
49: (8)                     if z.shape != tri.x.shape:
50: (12)                        raise ValueError('z array must have same length as triangulation
x'
51: (29)                                         ' and y arrays')
```

```
52: (8)                     z_check = z[np.unique(tri.get_masked_triangles())]
53: (8)                     if np.ma.is_masked(z_check):
54: (12)                        raise ValueError('z must not contain masked points within the '
55: (29)                                         'triangulation')
56: (8)                     if not np.isfinite(z_check).all():
57: (12)                        raise ValueError('z array must not contain non-finite values '
58: (29)                                         'within the triangulation')
59: (8)                     z = np.ma.masked_invalid(z, copy=False)
60: (8)                     self.zmax = float(z_check.max())
61: (8)                     self.zmin = float(z_check.min())
62: (8)                     if self.logscale and self.zmin <= 0:
63: (12)                        func = 'contourf' if self.filled else 'contour'
64: (12)                        raise ValueError(f'Cannot {func} log of negative values.')
65: (8)                     self._process_contour_level_args(args, z.dtype)
66: (8)                     return (tri, z)
67: (0)             _docstring.interpd.update(_tricontour_doc="""
68: (0)             Draw contour %%(type)s on an unstructured triangular grid.
69: (0)             Call signatures::
70: (4)                 %%(func)s(triangulation, z, [levels], ...)
71: (4)                 %%(func)s(x, y, z, [levels], *, [triangles=triangles], [mask=mask], ...)
72: (0)             The triangular grid can be specified either by passing a `.Triangulation`
73: (0)             object as the first parameter, or by passing the points *x*, *y* and
74: (0)             optionally the *triangles* and a *mask*. See `.Triangulation` for an
75: (0)             explanation of these parameters. If neither of *triangulation* or
76: (0)             *triangles* are given, the triangulation is calculated on the fly.
77: (0)             It is possible to pass *triangles* positionally, i.e.
78: (0)             ``%%(func)s(x, y, triangles, z, ...)``. However, this is discouraged. For more
79: (0)             clarity, pass *triangles* via keyword argument.
80: (0)             Parameters
81: (0)             ----------
82: (0)             triangulation : `.Triangulation`, optional
83: (4)                 An already created triangular grid.
84: (0)             x, y, triangles, mask
85: (4)                 Parameters defining the triangular grid. See `.Triangulation`.
86: (4)                 This is mutually exclusive with specifying *triangulation*.
87: (0)             z : array-like
88: (4)                 The height values over which the contour is drawn.  Color-mapping is
89: (4)                 controlled by *cmap*, *norm*, *vmin*, and *vmax*.
90: (4)                 .. note::
91: (8)                     All values in *z* must be finite. Hence, nan and inf values must
92: (8)                     either be removed or `~.Triangulation.set_mask` be used.
93: (0)             levels : int or array-like, optional
94: (4)                 Determines the number and positions of the contour lines / regions.
95: (4)                 If an int *n*, use `~matplotlib.ticker.MaxNLocator`, which tries to
96: (4)                 automatically choose no more than *n+1* "nice" contour levels between
97: (4)                 between minimum and maximum numeric values of *Z*.
98: (4)                 If array-like, draw contour lines at the specified levels.  The values
must
99: (4)                 be in increasing order.
100: (0)            Returns
101: (0)            -------
102: (0)            `~matplotlib.tri.TriContourSet`
103: (0)            Other Parameters
104: (0)            ----------------
105: (0)            colors : :mpltype:`color` or list of :mpltype:`color`, optional
106: (4)                The colors of the levels, i.e., the contour %%(type)s.
107: (4)                The sequence is cycled for the levels in ascending order. If the sequence
108: (4)                is shorter than the number of levels, it is repeated.
109: (4)                As a shortcut, single color strings may be used in place of one-element
110: (4)                lists, i.e. ``'red'`` instead of ``['red']`` to color all levels with the
111: (4)                same color. This shortcut does only work for color strings, not for other
112: (4)                ways of specifying colors.
113: (4)                By default (value *None*), the colormap specified by *cmap* will be used.
114: (0)            alpha : float, default: 1
115: (4)                The alpha blending value, between 0 (transparent) and 1 (opaque).
116: (0)            %(cmap_doc)s
117: (4)                This parameter is ignored if *colors* is set.
118: (0)            %(norm_doc)s
119: (4)                This parameter is ignored if *colors* is set.
```

```
(0)             %(vmin_vmax_doc)s
(4)                 If *vmin* or *vmax* are not given, the default color scaling is based on
(4)                 *levels*.
(4)                 This parameter is ignored if *colors* is set.
(0)             origin : {*None*, 'upper', 'lower', 'image'}, default: None
(4)                 Determines the orientation and exact position of *z* by specifying the
(4)                 position of ``z[0, 0]``.  This is only relevant, if *X*, *Y* are not
given.
(4)                 - *None*: ``z[0, 0]`` is at X=0, Y=0 in the lower left corner.
(4)                 - 'lower': ``z[0, 0]`` is at X=0.5, Y=0.5 in the lower left corner.
(4)                 - 'upper': ``z[0, 0]`` is at X=N+0.5, Y=0.5 in the upper left corner.
(4)                 - 'image': Use the value from :rc:`image.origin`.
(0)             extent : (x0, x1, y0, y1), optional
(4)                 If *origin* is not *None*, then *extent* is interpreted as in `.imshow`:
it
(4)                 gives the outer pixel boundaries. In this case, the position of z[0, 0] is
(4)                 the center of the pixel, not a corner. If *origin* is *None*, then
(4)                 (*x0*, *y0*) is the position of z[0, 0], and (*x1*, *y1*) is the position
(4)                 of z[-1, -1].
(4)                 This argument is ignored if *X* and *Y* are specified in the call to
(4)                 contour.
(0)             locator : ticker.Locator subclass, optional
(4)                 The locator is used to determine the contour levels if they are not given
(4)                 explicitly via *levels*.
(4)                 Defaults to `~.ticker.MaxNLocator`.
(0)             extend : {'neither', 'both', 'min', 'max'}, default: 'neither'
(4)                 Determines the ``%%(func)s``-coloring of values that are outside the
(4)                 *levels* range.
(4)                 If 'neither', values outside the *levels* range are not colored.  If
'min',
(4)                 'max' or 'both', color the values below, above or below and above the
(4)                 *levels* range.
(4)                 Values below ``min(levels)`` and above ``max(levels)`` are mapped to the
(4)                 under/over values of the `.Colormap`. Note that most colormaps do not have
(4)                 dedicated colors for these by default, so that the over and under values
(4)                 are the edge values of the colormap.  You may want to set these values
(4)                 explicitly using `.Colormap.set_under` and `.Colormap.set_over`.
(4)                 .. note::
(8)                     An existing `.TriContourSet` does not get notified if properties of
its
(8)                     colormap are changed. Therefore, an explicit call to
(8)                     `.ContourSet.changed()` is needed after modifying the colormap. The
(8)                     explicit call can be left out, if a colorbar is assigned to the
(8)                     `.TriContourSet` because it internally calls `.ContourSet.changed()`.
(0)             xunits, yunits : registered units, optional
(4)                 Override axis units by specifying an instance of a
(4)                 :class:`matplotlib.units.ConversionInterface`.
(0)             antialiased : bool, optional
(4)                 Enable antialiasing, overriding the defaults.  For
(4)                 filled contours, the default is *True*.  For line contours,
(4)                 it is taken from :rc:`lines.antialiased`."""% _docstring.interpd.params)
(0)             @_docstring.Substitution(func='tricontour', type='lines')
(0)             @_docstring.dedent_interpd
(0)             def tricontour(ax, *args, **kwargs):
(4)                 """
(4)                 %(_tricontour_doc)s
(4)                 linewidths : float or array-like, default: :rc:`contour.linewidth`
(8)                     The line width of the contour lines.
(8)                     If a number, all levels will be plotted with this linewidth.
(8)                     If a sequence, the levels in ascending order will be plotted with
(8)                     the linewidths in the order specified.
(8)                     If None, this falls back to :rc:`lines.linewidth`.
(4)                 linestyles : {*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional
(8)                     If *linestyles* is *None*, the default is 'solid' unless the lines are
(8)                     monochrome.  In that case, negative contours will take their linestyle
(8)                     from :rc:`contour.negative_linestyle` setting.
(8)                     *linestyles* can also be an iterable of the above strings specifying a
(8)                     set of linestyles to be used. If this iterable is shorter than the
(8)                     number of contour levels it will be repeated as necessary.
```

```
185: (4)                """
186: (4)                kwargs['filled'] = False
187: (4)                return TriContourSet(ax, *args, **kwargs)
188: (0)            @_docstring.Substitution(func='tricontourf', type='regions')
189: (0)            @_docstring.dedent_interpd
190: (0)            def tricontourf(ax, *args, **kwargs):
191: (4)                """
192: (4)                %(_tricontour_doc)s
193: (4)                hatches : list[str], optional
194: (8)                    A list of crosshatch patterns to use on the filled areas.
195: (8)                    If None, no hatching will be added to the contour.
196: (4)                Notes
197: (4)                -----
198: (4)                `.tricontourf` fills intervals that are closed at the top; that is, for
199: (4)                boundaries *z1* and *z2*, the filled region is::
200: (8)                    z1 < Z <= z2
201: (4)                except for the lowest interval, which is closed on both sides (i.e. it
202: (4)                includes the lowest value).
203: (4)                """
204: (4)                kwargs['filled'] = True
205: (4)                return TriContourSet(ax, *args, **kwargs)
```

File 234 - _trifinder.py:

```
1: (0)              import numpy as np
2: (0)              from matplotlib import _api
3: (0)              from matplotlib.tri import Triangulation
4: (0)              class TriFinder:
5: (4)                  """
6: (4)                  Abstract base class for classes used to find the triangles of a
7: (4)                  Triangulation in which (x, y) points lie.
8: (4)                  Rather than instantiate an object of a class derived from TriFinder, it is
9: (4)                  usually better to use the function `.Triangulation.get_trifinder`.
10: (4)                 Derived classes implement __call__(x, y) where x and y are array-like
point
11: (4)                 coordinates of the same shape.
12: (4)                 """
13: (4)                 def __init__(self, triangulation):
14: (8)                     _api.check_isinstance(Triangulation, triangulation=triangulation)
15: (8)                     self._triangulation = triangulation
16: (4)                 def __call__(self, x, y):
17: (8)                     raise NotImplementedError
18: (0)             class TrapezoidMapTriFinder(TriFinder):
19: (4)                 """
20: (4)                 `~matplotlib.tri.TriFinder` class implemented using the trapezoid
21: (4)                 map algorithm from the book "Computational Geometry, Algorithms and
22: (4)                 Applications", second edition, by M. de Berg, M. van Kreveld, M. Overmars
23: (4)                 and O. Schwarzkopf.
24: (4)                 The triangulation must be valid, i.e. it must not have duplicate points,
25: (4)                 triangles formed from colinear points, or overlapping triangles.  The
26: (4)                 algorithm has some tolerance to triangles formed from colinear points, but
27: (4)                 this should not be relied upon.
28: (4)                 """
29: (4)                 def __init__(self, triangulation):
30: (8)                     from matplotlib import _tri
31: (8)                     super().__init__(triangulation)
32: (8)                     self._cpp_trifinder = _tri.TrapezoidMapTriFinder(
33: (12)                        triangulation.get_cpp_triangulation())
34: (8)                     self._initialize()
35: (4)                 def __call__(self, x, y):
36: (8)                     """
37: (8)                     Return an array containing the indices of the triangles in which the
38: (8)                     specified *x*, *y* points lie, or -1 for points that do not lie within
39: (8)                     a triangle.
40: (8)                     *x*, *y* are array-like x and y coordinates of the same shape and any
41: (8)                     number of dimensions.
42: (8)                     Returns integer array with the same shape and *x* and *y*.
```

```
43: (8)                     """
44: (8)                     x = np.asarray(x, dtype=np.float64)
45: (8)                     y = np.asarray(y, dtype=np.float64)
46: (8)                     if x.shape != y.shape:
47: (12)                        raise ValueError("x and y must be array-like with the same shape")
48: (8)                     indices = (self._cpp_trifinder.find_many(x.ravel(), y.ravel())
49: (19)                               .reshape(x.shape))
50: (8)                     return indices
51: (4)                 def _get_tree_stats(self):
52: (8)                     """
53: (8)                     Return a python list containing the statistics about the node tree:
54: (12)                        0: number of nodes (tree size)
55: (12)                        1: number of unique nodes
56: (12)                        2: number of trapezoids (tree leaf nodes)
57: (12)                        3: number of unique trapezoids
58: (12)                        4: maximum parent count (max number of times a node is repeated in
59: (19)                               tree)
60: (12)                        5: maximum depth of tree (one more than the maximum number of
61: (19)                               comparisons needed to search through the tree)
62: (12)                        6: mean of all trapezoid depths (one more than the average number
63: (19)                               of comparisons needed to search through the tree)
64: (8)                     """
65: (8)                     return self._cpp_trifinder.get_tree_stats()
66: (4)                 def _initialize(self):
67: (8)                     """
68: (8)                     Initialize the underlying C++ object.  Can be called multiple times
if,
69: (8)                     for example, the triangulation is modified.
70: (8)                     """
71: (8)                     self._cpp_trifinder.initialize()
72: (4)                 def _print_tree(self):
73: (8)                     """
74: (8)                     Print a text representation of the node tree, which is useful for
75: (8)                     debugging purposes.
76: (8)                     """
77: (8)                     self._cpp_trifinder.print_tree()
```

File 235 - _triinterpolate.py:

```
1: (0)              """
2: (0)              Interpolation inside triangular grids.
3: (0)              """
4: (0)              import numpy as np
5: (0)              from matplotlib import _api
6: (0)              from matplotlib.tri import Triangulation
7: (0)              from matplotlib.tri._trifinder import TriFinder
8: (0)              from matplotlib.tri._tritools import TriAnalyzer
9: (0)              __all__ = ('TriInterpolator', 'LinearTriInterpolator', 'CubicTriInterpolator')
10: (0)             class TriInterpolator:
11: (4)                 """
12: (4)                 Abstract base class for classes used to interpolate on a triangular grid.
13: (4)                 Derived classes implement the following methods:
14: (4)                 - ``__call__(x, y)``,
15: (6)                   where x, y are array-like point coordinates of the same shape, and
16: (6)                   that returns a masked array of the same shape containing the
17: (6)                   interpolated z-values.
18: (4)                 - ``gradient(x, y)``,
19: (6)                   where x, y are array-like point coordinates of the same
20: (6)                   shape, and that returns a list of 2 masked arrays of the same shape
21: (6)                   containing the 2 derivatives of the interpolator (derivatives of
22: (6)                   interpolated z values with respect to x and y).
23: (4)                 """
24: (4)                 def __init__(self, triangulation, z, trifinder=None):
25: (8)                     _api.check_isinstance(Triangulation, triangulation=triangulation)
26: (8)                     self._triangulation = triangulation
27: (8)                     self._z = np.asarray(z)
28: (8)                     if self._z.shape != self._triangulation.x.shape:
```

```
(12)                        raise ValueError("z array must have same length as triangulation
x"
(29)                                            " and y arrays")
(8)                     _api.check_isinstance((TriFinder, None), trifinder=trifinder)
(8)                     self._trifinder = trifinder or self._triangulation.get_trifinder()
(8)                     self._unit_x = 1.0
(8)                     self._unit_y = 1.0
(8)                     self._tri_renum = None
(4)                 _docstring__call__ = """
(8)                     Returns a masked array containing interpolated values at the specified
(8)                     (x, y) points.
(8)                     Parameters
(8)                     ----------
(8)                     x, y : array-like
(12)                        x and y coordinates of the same shape and any number of
(12)                        dimensions.
(8)                     Returns
(8)                     -------
(8)                     np.ma.array
(12)                        Masked array of the same shape as *x* and *y*; values
corresponding
(12)                        to (*x*, *y*) points outside of the triangulation are masked out.
(8)                     """
(4)                 _docstringgradient = r"""
(8)                     Returns a list of 2 masked arrays containing interpolated derivatives
(8)                     at the specified (x, y) points.
(8)                     Parameters
(8)                     ----------
(8)                     x, y : array-like
(12)                        x and y coordinates of the same shape and any number of
(12)                        dimensions.
(8)                     Returns
(8)                     -------
(8)                     dzdx, dzdy : np.ma.array
(12)                        2 masked arrays of the same shape as *x* and *y*; values
(12)                        corresponding to (x, y) points outside of the triangulation
(12)                        are masked out.
(12)                        The first returned array contains the values of
(12)                        :math:`\frac{\partial z}{\partial x}` and the second those of
(12)                        :math:`\frac{\partial z}{\partial y}`.
(8)                     """
(4)                 def _interpolate_multikeys(self, x, y, tri_index=None,
(31)                                           return_keys=('z',)):
(8)                     """
(8)                     Versatile (private) method defined for all TriInterpolators.
(8)                     :meth:`_interpolate_multikeys` is a wrapper around method
(8)                     :meth:`_interpolate_single_key` (to be defined in the child
(8)                     subclasses).
(8)                     :meth:`_interpolate_single_key actually performs the interpolation,
(8)                     but only for 1-dimensional inputs and at valid locations (inside
(8)                     unmasked triangles of the triangulation).
(8)                     The purpose of :meth:`_interpolate_multikeys` is to implement the
(8)                     following common tasks needed in all subclasses implementations:
(8)                     - calculation of containing triangles
(8)                     - dealing with more than one interpolation request at the same
(10)                      location (e.g., if the 2 derivatives are requested, it is
(10)                      unnecessary to compute the containing triangles twice)
(8)                     - scaling according to self._unit_x, self._unit_y
(8)                     - dealing with points outside of the grid (with fill value np.nan)
(8)                     - dealing with multi-dimensional *x*, *y* arrays: flattening for
(10)                      :meth:`_interpolate_params` call and final reshaping.
(8)                     (Note that np.vectorize could do most of those things very well for
(8)                     you, but it does it by function evaluations over successive tuples of
(8)                     the input arrays. Therefore, this tends to be more time-consuming than
(8)                     using optimized numpy functions - e.g., np.dot - which can be used
(8)                     easily on the flattened inputs, in the child-subclass methods
(8)                     :meth:`_interpolate_single_key`.)
(8)                     It is guaranteed that the calls to :meth:`_interpolate_single_key`
(8)                     will be done with flattened (1-d) array-like input parameters *x*, *y*
```

```
 96: (8)                    and with flattened, valid `tri_index` arrays (no -1 index allowed).
 97: (8)                    Parameters
 98: (8)                    ----------
 99: (8)                    x, y : array-like
100: (12)                       x and y coordinates where interpolated values are requested.
101: (8)                    tri_index : array-like of int, optional
102: (12)                       Array of the containing triangle indices, same shape as
103: (12)                       *x* and *y*. Defaults to None. If None, these indices
104: (12)                       will be computed by a TriFinder instance.
105: (12)                       (Note: For point outside the grid, tri_index[ipt] shall be -1).
106: (8)                    return_keys : tuple of keys from {'z', 'dzdx', 'dzdy'}
107: (12)                       Defines the interpolation arrays to return, and in which order.
108: (8)                    Returns
109: (8)                    -------
110: (8)                    list of arrays
111: (12)                       Each array-like contains the expected interpolated values in the
112: (12)                       order defined by *return_keys* parameter.
113: (8)                    """
114: (8)                    x = np.asarray(x, dtype=np.float64)
115: (8)                    y = np.asarray(y, dtype=np.float64)
116: (8)                    sh_ret = x.shape
117: (8)                    if x.shape != y.shape:
118: (12)                       raise ValueError("x and y shall have same shapes."
119: (29)                                        f" Given: {x.shape} and {y.shape}")
120: (8)                    x = np.ravel(x)
121: (8)                    y = np.ravel(y)
122: (8)                    x_scaled = x/self._unit_x
123: (8)                    y_scaled = y/self._unit_y
124: (8)                    size_ret = np.size(x_scaled)
125: (8)                    if tri_index is None:
126: (12)                       tri_index = self._trifinder(x, y)
127: (8)                    else:
128: (12)                       if tri_index.shape != sh_ret:
129: (16)                           raise ValueError(
130: (20)                               "tri_index array is provided and shall"
131: (20)                               " have same shape as x and y. Given: "
132: (20)                               f"{tri_index.shape} and {sh_ret}")
133: (12)                       tri_index = np.ravel(tri_index)
134: (8)                    mask_in = (tri_index != -1)
135: (8)                    if self._tri_renum is None:
136: (12)                       valid_tri_index = tri_index[mask_in]
137: (8)                    else:
138: (12)                       valid_tri_index = self._tri_renum[tri_index[mask_in]]
139: (8)                    valid_x = x_scaled[mask_in]
140: (8)                    valid_y = y_scaled[mask_in]
141: (8)                    ret = []
142: (8)                    for return_key in return_keys:
143: (12)                       try:
144: (16)                           return_index = {'z': 0, 'dzdx': 1, 'dzdy': 2}[return_key]
145: (12)                       except KeyError as err:
146: (16)                           raise ValueError("return_keys items shall take values in"
147: (33)                                            " {'z', 'dzdx', 'dzdy'}") from err
148: (12)                       scale = [1., 1./self._unit_x, 1./self._unit_y][return_index]
149: (12)                       ret_loc = np.empty(size_ret, dtype=np.float64)
150: (12)                       ret_loc[~mask_in] = np.nan
151: (12)                       ret_loc[mask_in] = self._interpolate_single_key(
152: (16)                           return_key, valid_tri_index, valid_x, valid_y) * scale
153: (12)                       ret += [np.ma.masked_invalid(ret_loc.reshape(sh_ret), copy=False)]
154: (8)                    return ret
155: (4)                def _interpolate_single_key(self, return_key, tri_index, x, y):
156: (8)                    """
157: (8)                    Interpolate at points belonging to the triangulation
158: (8)                    (inside an unmasked triangles).
159: (8)                    Parameters
160: (8)                    ----------
161: (8)                    return_key : {'z', 'dzdx', 'dzdy'}
162: (12)                       The requested values (z or its derivatives).
163: (8)                    tri_index : 1D int array
164: (12)                       Valid triangle index (cannot be -1).
```

```
(8)                 x, y : 1D arrays, same shape as `tri_index`
(12)                    Valid locations where interpolation is requested.
(8)                 Returns
(8)                 -------
(8)                 1-d array
(12)                    Returned array of the same size as *tri_index*
(8)                 """
(8)                 raise NotImplementedError("TriInterpolator subclasses" +
(34)                                          "should implement _interpolate_single_key!")
(0)         class LinearTriInterpolator(TriInterpolator):
(4)             """
(4)             Linear interpolator on a triangular grid.
(4)             Each triangle is represented by a plane so that an interpolated value at
(4)             point (x, y) lies on the plane of the triangle containing (x, y).
(4)             Interpolated values are therefore continuous across the triangulation, but
(4)             their first derivatives are discontinuous at edges between triangles.
(4)             Parameters
(4)             ----------
(4)             triangulation : `~matplotlib.tri.Triangulation`
(8)                 The triangulation to interpolate over.
(4)             z : (npoints,) array-like
(8)                 Array of values, defined at grid points, to interpolate between.
(4)             trifinder : `~matplotlib.tri.TriFinder`, optional
(8)                 If this is not specified, the Triangulation's default TriFinder will
(8)                 be used by calling `.Triangulation.get_trifinder`.
(4)             Methods
(4)             -------
(4)             `__call__` (x, y) : Returns interpolated values at (x, y) points.
(4)             `gradient` (x, y) : Returns interpolated derivatives at (x, y) points.
(4)             """
(4)             def __init__(self, triangulation, z, trifinder=None):
(8)                 super().__init__(triangulation, z, trifinder)
(8)                 self._plane_coefficients = \
(12)                    self._triangulation.calculate_plane_coefficients(self._z)
(4)             def __call__(self, x, y):
(8)                 return self._interpolate_multikeys(x, y, tri_index=None,
(43)                                                   return_keys=('z',))[0]
(4)             __call__.__doc__ = TriInterpolator._docstring__call__
(4)             def gradient(self, x, y):
(8)                 return self._interpolate_multikeys(x, y, tri_index=None,
(43)                                                   return_keys=('dzdx', 'dzdy'))
(4)             gradient.__doc__ = TriInterpolator._docstringgradient
(4)             def _interpolate_single_key(self, return_key, tri_index, x, y):
(8)                 _api.check_in_list(['z', 'dzdx', 'dzdy'], return_key=return_key)
(8)                 if return_key == 'z':
(12)                    return (self._plane_coefficients[tri_index, 0]*x +
(20)                            self._plane_coefficients[tri_index, 1]*y +
(20)                            self._plane_coefficients[tri_index, 2])
(8)                 elif return_key == 'dzdx':
(12)                    return self._plane_coefficients[tri_index, 0]
(8)                 else:  # 'dzdy'
(12)                    return self._plane_coefficients[tri_index, 1]
(0)         class CubicTriInterpolator(TriInterpolator):
(4)             r"""
(4)             Cubic interpolator on a triangular grid.
(4)             In one-dimension - on a segment - a cubic interpolating function is
(4)             defined by the values of the function and its derivative at both ends.
(4)             This is almost the same in 2D inside a triangle, except that the values
(4)             of the function and its 2 derivatives have to be defined at each triangle
(4)             node.
(4)             The CubicTriInterpolator takes the value of the function at each node -
(4)             provided by the user - and internally computes the value of the
(4)             derivatives, resulting in a smooth interpolation.
(4)             (As a special feature, the user can also impose the value of the
(4)             derivatives at each node, but this is not supposed to be the common
(4)             usage.)
(4)             Parameters
(4)             ----------
(4)             triangulation : `~matplotlib.tri.Triangulation`
```

```
(8)                 The triangulation to interpolate over.
(4)             z : (npoints,) array-like
(8)                 Array of values, defined at grid points, to interpolate between.
(4)             kind : {'min_E', 'geom', 'user'}, optional
(8)                 Choice of the smoothing algorithm, in order to compute
(8)                 the interpolant derivatives (defaults to 'min_E'):
(8)                 - if 'min_E': (default) The derivatives at each node is computed
(10)                  to minimize a bending energy.
(8)                 - if 'geom': The derivatives at each node is computed as a
(10)                  weighted average of relevant triangle normals. To be used for
(10)                  speed optimization (large grids).
(8)                 - if 'user': The user provides the argument *dz*, no computation
(10)                  is hence needed.
(4)             trifinder : `~matplotlib.tri.TriFinder`, optional
(8)                 If not specified, the Triangulation's default TriFinder will
(8)                 be used by calling `.Triangulation.get_trifinder`.
(4)             dz : tuple of array-likes (dzdx, dzdy), optional
(8)                 Used only if  *kind* ='user'. In this case *dz* must be provided as
(8)                 (dzdx, dzdy) where dzdx, dzdy are arrays of the same shape as *z* and
(8)                 are the interpolant first derivatives at the *triangulation* points.
(4)             Methods
(4)             -------
(4)             `__call__` (x, y) : Returns interpolated values at (x, y) points.
(4)             `gradient` (x, y) : Returns interpolated derivatives at (x, y) points.
(4)             Notes
(4)             -----
(4)             This note is a bit technical and details how the cubic interpolation is
(4)             computed.
(4)             The interpolation is based on a Clough-Tocher subdivision scheme of
(4)             the *triangulation* mesh (to make it clearer, each triangle of the
(4)             grid will be divided in 3 child-triangles, and on each child triangle
(4)             the interpolated function is a cubic polynomial of the 2 coordinates).
(4)             This technique originates from FEM (Finite Element Method) analysis;
(4)             the element used is a reduced Hsieh-Clough-Tocher (HCT)
(4)             element. Its shape functions are described in [1]_.
(4)             The assembled function is guaranteed to be C1-smooth, i.e. it is
(4)             continuous and its first derivatives are also continuous (this
(4)             is easy to show inside the triangles but is also true when crossing the
(4)             edges).
(4)             In the default case (*kind* ='min_E'), the interpolant minimizes a
(4)             curvature energy on the functional space generated by the HCT element
(4)             shape functions - with imposed values but arbitrary derivatives at each
(4)             node. The minimized functional is the integral of the so-called total
(4)             curvature (implementation based on an algorithm from [2]_ - PCG sparse
(4)             solver):
(4)             .. math::
(8)                 E(z) = \frac{1}{2} \int_{\Omega} \left(
(12)                    \left( \frac{\partial^2{z}}{\partial{x}^2} \right)^2 +
(12)                    \left( \frac{\partial^2{z}}{\partial{y}^2} \right)^2 +
(12)                    2\left( \frac{\partial^2{z}}{\partial{y}\partial{x}} \right)^2
(8)                 \right) dx\,dy
(4)             If the case *kind* ='geom' is chosen by the user, a simple geometric
(4)             approximation is used (weighted average of the triangle normal
(4)             vectors), which could improve speed on very large grids.
(4)             References
(4)             ----------
(4)             .. [1] Michel Bernadou, Kamal Hassan, "Basis functions for general
(8)                 Hsieh-Clough-Tocher triangles, complete or reduced.",
(8)                 International Journal for Numerical Methods in Engineering,
(8)                 17(5):784 - 789. 2.01.
(4)             .. [2] C.T. Kelley, "Iterative Methods for Optimization".
(4)             """
(4)             def __init__(self, triangulation, z, kind='min_E', trifinder=None,
(17)                         dz=None):
(8)                 super().__init__(triangulation, z, trifinder)
(8)                 self._triangulation.get_cpp_triangulation()
(8)                 tri_analyzer = TriAnalyzer(self._triangulation)
(8)                 (compressed_triangles, compressed_x, compressed_y, tri_renum,
(9)                  node_renum) = tri_analyzer._get_compressed_triangulation()
```

```
303: (8)                    self._triangles = compressed_triangles
304: (8)                    self._tri_renum = tri_renum
305: (8)                    valid_node = (node_renum != -1)
306: (8)                    self._z[node_renum[valid_node]] = self._z[valid_node]
307: (8)                    self._unit_x = np.ptp(compressed_x)
308: (8)                    self._unit_y = np.ptp(compressed_y)
309: (8)                    self._pts = np.column_stack([compressed_x / self._unit_x,
310: (37)                                               compressed_y / self._unit_y])
311: (8)                    self._tris_pts = self._pts[self._triangles]
312: (8)                    self._eccs = self._compute_tri_eccentricities(self._tris_pts)
313: (8)                    _api.check_in_list(['user', 'geom', 'min_E'], kind=kind)
314: (8)                    self._dof = self._compute_dof(kind, dz=dz)
315: (8)                    self._ReferenceElement = _ReducedHCT_Element()
316: (4)                def __call__(self, x, y):
317: (8)                    return self._interpolate_multikeys(x, y, tri_index=None,
318: (43)                                                       return_keys=('z',))[0]
319: (4)                __call__.__doc__ = TriInterpolator._docstring__call__
320: (4)                def gradient(self, x, y):
321: (8)                    return self._interpolate_multikeys(x, y, tri_index=None,
322: (43)                                                       return_keys=('dzdx', 'dzdy'))
323: (4)                gradient.__doc__ = TriInterpolator._docstringgradient
324: (4)                def _interpolate_single_key(self, return_key, tri_index, x, y):
325: (8)                    _api.check_in_list(['z', 'dzdx', 'dzdy'], return_key=return_key)
326: (8)                    tris_pts = self._tris_pts[tri_index]
327: (8)                    alpha = self._get_alpha_vec(x, y, tris_pts)
328: (8)                    ecc = self._eccs[tri_index]
329: (8)                    dof = np.expand_dims(self._dof[tri_index], axis=1)
330: (8)                    if return_key == 'z':
331: (12)                       return self._ReferenceElement.get_function_values(
332: (16)                           alpha, ecc, dof)
333: (8)                    else:  # 'dzdx', 'dzdy'
334: (12)                       J = self._get_jacobian(tris_pts)
335: (12)                       dzdx = self._ReferenceElement.get_function_derivatives(
336: (16)                           alpha, J, ecc, dof)
337: (12)                       if return_key == 'dzdx':
338: (16)                           return dzdx[:, 0, 0]
339: (12)                       else:
340: (16)                           return dzdx[:, 1, 0]
341: (4)                def _compute_dof(self, kind, dz=None):
342: (8)                    """
343: (8)                    Compute and return nodal dofs according to kind.
344: (8)                    Parameters
345: (8)                    ----------
346: (8)                    kind : {'min_E', 'geom', 'user'}
347: (12)                       Choice of the _DOF_estimator subclass to estimate the gradient.
348: (8)                    dz : tuple of array-likes (dzdx, dzdy), optional
349: (12)                       Used only if *kind*=user; in this case passed to the
350: (12)                       :class:`_DOF_estimator_user`.
351: (8)                    Returns
352: (8)                    -------
353: (8)                    array-like, shape (npts, 2)
354: (12)                       Estimation of the gradient at triangulation nodes (stored as
355: (12)                       degree of freedoms of reduced-HCT triangle elements).
356: (8)                    """
357: (8)                    if kind == 'user':
358: (12)                       if dz is None:
359: (16)                           raise ValueError("For a CubicTriInterpolator with "
360: (33)                                            "*kind*='user', a valid *dz* "
361: (33)                                            "argument is expected.")
362: (12)                       TE = _DOF_estimator_user(self, dz=dz)
363: (8)                    elif kind == 'geom':
364: (12)                       TE = _DOF_estimator_geom(self)
365: (8)                    else:  # 'min_E', checked in __init__
366: (12)                       TE = _DOF_estimator_min_E(self)
367: (8)                    return TE.compute_dof_from_df()
368: (4)                @staticmethod
369: (4)                def _get_alpha_vec(x, y, tris_pts):
370: (8)                    """
371: (8)                    Fast (vectorized) function to compute barycentric coordinates alpha.
```

```
(8)                     Parameters
(8)                     ----------
(8)                     x, y : array-like of dim 1 (shape (nx,))
(12)                        Coordinates of the points whose points barycentric coordinates are
(12)                        requested.
(8)                     tris_pts : array like of dim 3 (shape: (nx, 3, 2))
(12)                        Coordinates of the containing triangles apexes.
(8)                     Returns
(8)                     -------
(8)                     array of dim 2 (shape (nx, 3))
(12)                        Barycentric coordinates of the points inside the containing
(12)                        triangles.
(8)                     """
(8)                     ndim = tris_pts.ndim-2
(8)                     a = tris_pts[:, 1, :] - tris_pts[:, 0, :]
(8)                     b = tris_pts[:, 2, :] - tris_pts[:, 0, :]
(8)                     abT = np.stack([a, b], axis=-1)
(8)                     ab = _transpose_vectorized(abT)
(8)                     OM = np.stack([x, y], axis=1) - tris_pts[:, 0, :]
(8)                     metric = ab @ abT
(8)                     metric_inv = _pseudo_inv22sym_vectorized(metric)
(8)                     Covar = ab @ _transpose_vectorized(np.expand_dims(OM, ndim))
(8)                     ksi = metric_inv @ Covar
(8)                     alpha = _to_matrix_vectorized([
(12)                        [1-ksi[:, 0, 0]-ksi[:, 1, 0]], [ksi[:, 0, 0]], [ksi[:, 1, 0]]])
(8)                     return alpha
(4)                 @staticmethod
(4)                 def _get_jacobian(tris_pts):
(8)                     """
(8)                     Fast (vectorized) function to compute triangle jacobian matrix.
(8)                     Parameters
(8)                     ----------
(8)                     tris_pts : array like of dim 3 (shape: (nx, 3, 2))
(12)                        Coordinates of the containing triangles apexes.
(8)                     Returns
(8)                     -------
(8)                     array of dim 3 (shape (nx, 2, 2))
(12)                        Barycentric coordinates of the points inside the containing
(12)                        triangles.
(12)                        J[itri, :, :] is the jacobian matrix at apex 0 of the triangle
(12)                        itri, so that the following (matrix) relationship holds:
(15)                           [dz/dksi] = [J] x [dz/dx]
(12)                        with x: global coordinates
(17)                             ksi: element parametric coordinates in triangle first apex
(17)                             local basis.
(8)                     """
(8)                     a = np.array(tris_pts[:, 1, :] - tris_pts[:, 0, :])
(8)                     b = np.array(tris_pts[:, 2, :] - tris_pts[:, 0, :])
(8)                     J = _to_matrix_vectorized([[a[:, 0], a[:, 1]],
(35)                                               [b[:, 0], b[:, 1]]])
(8)                     return J
(4)                 @staticmethod
(4)                 def _compute_tri_eccentricities(tris_pts):
(8)                     """
(8)                     Compute triangle eccentricities.
(8)                     Parameters
(8)                     ----------
(8)                     tris_pts : array like of dim 3 (shape: (nx, 3, 2))
(12)                        Coordinates of the triangles apexes.
(8)                     Returns
(8)                     -------
(8)                     array like of dim 2 (shape: (nx, 3))
(12)                        The so-called eccentricity parameters [1] needed for HCT
triangular
(12)                        element.
(8)                     """
(8)                     a = np.expand_dims(tris_pts[:, 2, :] - tris_pts[:, 1, :], axis=2)
(8)                     b = np.expand_dims(tris_pts[:, 0, :] - tris_pts[:, 2, :], axis=2)
(8)                     c = np.expand_dims(tris_pts[:, 1, :] - tris_pts[:, 0, :], axis=2)
```

```
440: (8)                    dot_a = (_transpose_vectorized(a) @ a)[:, 0, 0]
441: (8)                    dot_b = (_transpose_vectorized(b) @ b)[:, 0, 0]
442: (8)                    dot_c = (_transpose_vectorized(c) @ c)[:, 0, 0]
443: (8)                    return _to_matrix_vectorized([[(dot_c-dot_b) / dot_a],
444: (38)                                                 [(dot_a-dot_c) / dot_b],
445: (38)                                                 [(dot_b-dot_a) / dot_c]])
446: (0)            class _ReducedHCT_Element:
447: (4)                """
448: (4)                Implementation of reduced HCT triangular element with explicit shape
449: (4)                functions.
450: (4)                Computes z, dz, d2z and the element stiffness matrix for bending energy:
451: (4)                E(f) = integral( (d2z/dx2 + d2z/dy2)**2 dA)
452: (4)                *** Reference for the shape functions: ***
453: (4)                [1] Basis functions for general Hsieh-Clough-Tocher _triangles, complete
or
454: (8)                    reduced.
455: (8)                    Michel Bernadou, Kamal Hassan
456: (8)                    International Journal for Numerical Methods in Engineering.
457: (8)                    17(5):784 - 789.  2.01
458: (4)                *** Element description: ***
459: (4)                9 dofs: z and dz given at 3 apex
460: (4)                C1 (conform)
461: (4)                """
462: (4)                M = np.array([
463: (8)                    [ 0.00, 0.00, 0.00,  4.50,  4.50, 0.00, 0.00, 0.00, 0.00, 0.00],
464: (8)                    [-0.25, 0.00, 0.00,  0.50,  1.25, 0.00, 0.00, 0.00, 0.00, 0.00],
465: (8)                    [-0.25, 0.00, 0.00,  1.25,  0.50, 0.00, 0.00, 0.00, 0.00, 0.00],
466: (8)                    [ 0.50, 1.00, 0.00, -1.50,  0.00, 3.00, 3.00, 0.00, 0.00, 3.00],
467: (8)                    [ 0.00, 0.00, 0.00, -0.25,  0.25, 0.00, 1.00, 0.00, 0.00, 0.50],
468: (8)                    [ 0.25, 0.00, 0.00, -0.50, -0.25, 1.00, 0.00, 0.00, 0.00, 1.00],
469: (8)                    [ 0.50, 0.00, 1.00,  0.00, -1.50, 0.00, 0.00, 3.00, 3.00, 3.00],
470: (8)                    [ 0.25, 0.00, 0.00, -0.25, -0.50, 0.00, 0.00, 0.00, 1.00, 1.00],
471: (8)                    [ 0.00, 0.00, 0.00,  0.25, -0.25, 0.00, 0.00, 1.00, 0.00, 0.50]])
472: (4)                M0 = np.array([
473: (8)                    [ 0.00, 0.00, 0.00,  0.00,  0.00, 0.00, 0.00, 0.00, 0.00,  0.00],
474: (8)                    [ 0.00, 0.00, 0.00,  0.00,  0.00, 0.00, 0.00, 0.00, 0.00,  0.00],
475: (8)                    [ 0.00, 0.00, 0.00,  0.00,  0.00, 0.00, 0.00, 0.00, 0.00,  0.00],
476: (8)                    [-1.00, 0.00, 0.00,  1.50,  1.50, 0.00, 0.00, 0.00, 0.00, -3.00],
477: (8)                    [-0.50, 0.00, 0.00,  0.75,  0.75, 0.00, 0.00, 0.00, 0.00, -1.50],
478: (8)                    [ 0.00, 0.00, 0.00,  0.00,  0.00, 0.00, 0.00, 0.00, 0.00,  0.00],
479: (8)                    [ 1.00, 0.00, 0.00, -1.50, -1.50, 0.00, 0.00, 0.00, 0.00,  3.00],
480: (8)                    [ 0.00, 0.00, 0.00,  0.00,  0.00, 0.00, 0.00, 0.00, 0.00,  0.00],
481: (8)                    [ 0.50, 0.00, 0.00, -0.75, -0.75, 0.00, 0.00, 0.00, 0.00,  1.50]])
482: (4)                M1 = np.array([
483: (8)                    [-0.50, 0.00, 0.00,  1.50, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
484: (8)                    [ 0.00, 0.00, 0.00,  0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
485: (8)                    [-0.25, 0.00, 0.00,  0.75, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
486: (8)                    [ 0.00, 0.00, 0.00,  0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
487: (8)                    [ 0.00, 0.00, 0.00,  0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
488: (8)                    [ 0.00, 0.00, 0.00,  0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
489: (8)                    [ 0.50, 0.00, 0.00, -1.50, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
490: (8)                    [ 0.25, 0.00, 0.00, -0.75, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
491: (8)                    [ 0.00, 0.00, 0.00,  0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00]])
492: (4)                M2 = np.array([
493: (8)                    [ 0.50, 0.00, 0.00, 0.00, -1.50, 0.00, 0.00, 0.00, 0.00, 0.00],
494: (8)                    [ 0.25, 0.00, 0.00, 0.00, -0.75, 0.00, 0.00, 0.00, 0.00, 0.00],
495: (8)                    [ 0.00, 0.00, 0.00, 0.00,  0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
496: (8)                    [-0.50, 0.00, 0.00, 0.00,  1.50, 0.00, 0.00, 0.00, 0.00, 0.00],
497: (8)                    [ 0.00, 0.00, 0.00, 0.00,  0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
498: (8)                    [-0.25, 0.00, 0.00, 0.00,  0.75, 0.00, 0.00, 0.00, 0.00, 0.00],
499: (8)                    [ 0.00, 0.00, 0.00, 0.00,  0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
500: (8)                    [ 0.00, 0.00, 0.00, 0.00,  0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
501: (8)                    [ 0.00, 0.00, 0.00, 0.00,  0.00, 0.00, 0.00, 0.00, 0.00, 0.00]])
502: (4)                rotate_dV = np.array([[ 1.,  0.], [ 0.,  1.],
503: (26)                                     [ 0.,  1.], [-1., -1.],
504: (26)                                     [-1., -1.], [ 1.,  0.]])
505: (4)                rotate_d2V = np.array([[1., 0., 0.], [0., 1., 0.], [ 0.,  0.,  1.],
506: (27)                                      [0., 1., 0.], [1., 1., 1.], [ 0., -2., -1.],
507: (27)                                      [1., 1., 1.], [1., 0., 0.], [-2.,  0., -1.]])
```

```
(4)                 n_gauss = 9
(4)                 gauss_pts = np.array([[13./18.,  4./18.,  1./18.],
(26)                                      [ 4./18., 13./18.,  1./18.],
(26)                                      [ 7./18.,  7./18.,  4./18.],
(26)                                      [ 1./18., 13./18.,  4./18.],
(26)                                      [ 1./18.,  4./18., 13./18.],
(26)                                      [ 4./18.,  7./18.,  7./18.],
(26)                                      [ 4./18.,  1./18., 13./18.],
(26)                                      [13./18.,  1./18.,  4./18.],
(26)                                      [ 7./18.,  4./18.,  7./18.]], dtype=np.float64)
(4)                 gauss_w = np.ones([9], dtype=np.float64) / 9.
(4)                 E = np.array([[1., 0., 0.], [0., 1., 0.], [0., 0., 2.]])
(4)                 J0_to_J1 = np.array([[-1.,  1.], [-1.,  0.]])
(4)                 J0_to_J2 = np.array([[ 0., -1.], [ 1., -1.]])
(4)                 def get_function_values(self, alpha, ecc, dofs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     alpha : is a (N x 3 x 1) array (array of column-matrices) of
(8)                     barycentric coordinates,
(8)                     ecc : is a (N x 3 x 1) array (array of column-matrices) of triangle
(8)                     eccentricities,
(8)                     dofs : is a (N x 1 x 9) arrays (arrays of row-matrices) of computed
(8)                     degrees of freedom.
(8)                     Returns
(8)                     -------
(8)                     Returns the N-array of interpolated function values.
(8)                     """
(8)                     subtri = np.argmin(alpha, axis=1)[:, 0]
(8)                     ksi = _roll_vectorized(alpha, -subtri, axis=0)
(8)                     E = _roll_vectorized(ecc, -subtri, axis=0)
(8)                     x = ksi[:, 0, 0]
(8)                     y = ksi[:, 1, 0]
(8)                     z = ksi[:, 2, 0]
(8)                     x_sq = x*x
(8)                     y_sq = y*y
(8)                     z_sq = z*z
(8)                     V = _to_matrix_vectorized([
(12)                        [x_sq*x], [y_sq*y], [z_sq*z], [x_sq*z], [x_sq*y], [y_sq*x],
(12)                        [y_sq*z], [z_sq*y], [z_sq*x], [x*y*z]])
(8)                     prod = self.M @ V
(8)                     prod += _scalar_vectorized(E[:, 0, 0], self.M0 @ V)
(8)                     prod += _scalar_vectorized(E[:, 1, 0], self.M1 @ V)
(8)                     prod += _scalar_vectorized(E[:, 2, 0], self.M2 @ V)
(8)                     s = _roll_vectorized(prod, 3*subtri, axis=0)
(8)                     return (dofs @ s)[:, 0, 0]
(4)                 def get_function_derivatives(self, alpha, J, ecc, dofs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     *alpha* is a (N x 3 x 1) array (array of column-matrices of
(8)                     barycentric coordinates)
(8)                     *J* is a (N x 2 x 2) array of jacobian matrices (jacobian matrix at
(8)                     triangle first apex)
(8)                     *ecc* is a (N x 3 x 1) array (array of column-matrices of triangle
(8)                     eccentricities)
(8)                     *dofs* is a (N x 1 x 9) arrays (arrays of row-matrices) of computed
(8)                     degrees of freedom.
(8)                     Returns
(8)                     -------
(8)                     Returns the values of interpolated function derivatives [dz/dx, dz/dy]
(8)                     in global coordinates at locations alpha, as a column-matrices of
(8)                     shape (N x 2 x 1).
(8)                     """
(8)                     subtri = np.argmin(alpha, axis=1)[:, 0]
(8)                     ksi = _roll_vectorized(alpha, -subtri, axis=0)
(8)                     E = _roll_vectorized(ecc, -subtri, axis=0)
(8)                     x = ksi[:, 0, 0]
(8)                     y = ksi[:, 1, 0]
```

```
(8)                     z = ksi[:, 2, 0]
(8)                     x_sq = x*x
(8)                     y_sq = y*y
(8)                     z_sq = z*z
(8)                     dV = _to_matrix_vectorized([
(12)                        [    -3.*x_sq,     -3.*x_sq],
(12)                        [     3.*y_sq,           0.],
(12)                        [          0.,      3.*z_sq],
(12)                        [     -2.*x*z, -2.*x*z+x_sq],
(12)                        [-2.*x*y+x_sq,      -2.*x*y],
(12)                        [ 2.*x*y-y_sq,        -y_sq],
(12)                        [      2.*y*z,         y_sq],
(12)                        [        z_sq,       2.*y*z],
(12)                        [       -z_sq,  2.*x*z-z_sq],
(12)                        [     x*z-y*z,      x*y-y*z]])
(8)                     dV = dV @ _extract_submatrices(
(12)                        self.rotate_dV, subtri, block_size=2, axis=0)
(8)                     prod = self.M @ dV
(8)                     prod += _scalar_vectorized(E[:, 0, 0], self.M0 @ dV)
(8)                     prod += _scalar_vectorized(E[:, 1, 0], self.M1 @ dV)
(8)                     prod += _scalar_vectorized(E[:, 2, 0], self.M2 @ dV)
(8)                     dsdksi = _roll_vectorized(prod, 3*subtri, axis=0)
(8)                     dfdksi = dofs @ dsdksi
(8)                     J_inv = _safe_inv22_vectorized(J)
(8)                     dfdx = J_inv @ _transpose_vectorized(dfdksi)
(8)                     return dfdx
(4)                 def get_function_hessians(self, alpha, J, ecc, dofs):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     *alpha* is a (N x 3 x 1) array (array of column-matrices) of
(8)                     barycentric coordinates
(8)                     *J* is a (N x 2 x 2) array of jacobian matrices (jacobian matrix at
(8)                     triangle first apex)
(8)                     *ecc* is a (N x 3 x 1) array (array of column-matrices) of triangle
(8)                     eccentricities
(8)                     *dofs* is a (N x 1 x 9) arrays (arrays of row-matrices) of computed
(8)                     degrees of freedom.
(8)                     Returns
(8)                     -------
(8)                     Returns the values of interpolated function 2nd-derivatives
(8)                     [d2z/dx2, d2z/dy2, d2z/dxdy] in global coordinates at locations alpha,
(8)                     as a column-matrices of shape (N x 3 x 1).
(8)                     """
(8)                     d2sdksi2 = self.get_d2Sidksij2(alpha, ecc)
(8)                     d2fdksi2 = dofs @ d2sdksi2
(8)                     H_rot = self.get_Hrot_from_J(J)
(8)                     d2fdx2 = d2fdksi2 @ H_rot
(8)                     return _transpose_vectorized(d2fdx2)
(4)                 def get_d2Sidksij2(self, alpha, ecc):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     *alpha* is a (N x 3 x 1) array (array of column-matrices) of
(8)                     barycentric coordinates
(8)                     *ecc* is a (N x 3 x 1) array (array of column-matrices) of triangle
(8)                     eccentricities
(8)                     Returns
(8)                     -------
(8)                     Returns the arrays d2sdksi2 (N x 3 x 1) Hessian of shape functions
(8)                     expressed in covariant coordinates in first apex basis.
(8)                     """
(8)                     subtri = np.argmin(alpha, axis=1)[:, 0]
(8)                     ksi = _roll_vectorized(alpha, -subtri, axis=0)
(8)                     E = _roll_vectorized(ecc, -subtri, axis=0)
(8)                     x = ksi[:, 0, 0]
(8)                     y = ksi[:, 1, 0]
(8)                     z = ksi[:, 2, 0]
(8)                     d2V = _to_matrix_vectorized([
```

```
(12)                        [     6.*x,      6.*x,      6.*x],
(12)                        [     6.*y,        0.,        0.],
(12)                        [       0.,      6.*z,        0.],
(12)                        [     2.*z, 2.*z-4.*x, 2.*z-2.*x],
(12)                        [2.*y-4.*x,      2.*y, 2.*y-2.*x],
(12)                        [2.*x-4.*y,        0.,     -2.*y],
(12)                        [     2.*z,        0.,      2.*y],
(12)                        [       0.,      2.*y,      2.*z],
(12)                        [       0., 2.*x-4.*z,     -2.*z],
(12)                        [    -2.*z,     -2.*y,     x-y-z]])
(8)                     d2V = d2V @ _extract_submatrices(
(12)                        self.rotate_d2V, subtri, block_size=3, axis=0)
(8)                     prod = self.M @ d2V
(8)                     prod += _scalar_vectorized(E[:, 0, 0], self.M0 @ d2V)
(8)                     prod += _scalar_vectorized(E[:, 1, 0], self.M1 @ d2V)
(8)                     prod += _scalar_vectorized(E[:, 2, 0], self.M2 @ d2V)
(8)                     d2sdksi2 = _roll_vectorized(prod, 3*subtri, axis=0)
(8)                     return d2sdksi2
(4)                 def get_bending_matrices(self, J, ecc):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     *J* is a (N x 2 x 2) array of jacobian matrices (jacobian matrix at
(8)                     triangle first apex)
(8)                     *ecc* is a (N x 3 x 1) array (array of column-matrices) of triangle
(8)                     eccentricities
(8)                     Returns
(8)                     -------
(8)                     Returns the element K matrices for bending energy expressed in
(8)                     GLOBAL nodal coordinates.
(8)                     K_ij = integral [ (d2zi/dx2 + d2zi/dy2) * (d2zj/dx2 + d2zj/dy2) dA]
(8)                     tri_J is needed to rotate dofs from local basis to global basis
(8)                     """
(8)                     n = np.size(ecc, 0)
(8)                     J1 = self.J0_to_J1 @ J
(8)                     J2 = self.J0_to_J2 @ J
(8)                     DOF_rot = np.zeros([n, 9, 9], dtype=np.float64)
(8)                     DOF_rot[:, 0, 0] = 1
(8)                     DOF_rot[:, 3, 3] = 1
(8)                     DOF_rot[:, 6, 6] = 1
(8)                     DOF_rot[:, 1:3, 1:3] = J
(8)                     DOF_rot[:, 4:6, 4:6] = J1
(8)                     DOF_rot[:, 7:9, 7:9] = J2
(8)                     H_rot, area = self.get_Hrot_from_J(J, return_area=True)
(8)                     K = np.zeros([n, 9, 9], dtype=np.float64)
(8)                     weights = self.gauss_w
(8)                     pts = self.gauss_pts
(8)                     for igauss in range(self.n_gauss):
(12)                        alpha = np.tile(pts[igauss, :], n).reshape(n, 3)
(12)                        alpha = np.expand_dims(alpha, 2)
(12)                        weight = weights[igauss]
(12)                        d2Skdksi2 = self.get_d2Sidksij2(alpha, ecc)
(12)                        d2Skdx2 = d2Skdksi2 @ H_rot
(12)                        K += weight * (d2Skdx2 @ self.E @ _transpose_vectorized(d2Skdx2))
(8)                     K = _transpose_vectorized(DOF_rot) @ K @ DOF_rot
(8)                     return _scalar_vectorized(area, K)
(4)                 def get_Hrot_from_J(self, J, return_area=False):
(8)                     """
(8)                     Parameters
(8)                     ----------
(8)                     *J* is a (N x 2 x 2) array of jacobian matrices (jacobian matrix at
(8)                     triangle first apex)
(8)                     Returns
(8)                     -------
(8)                     Returns H_rot used to rotate Hessian from local basis of first apex,
(8)                     to global coordinates.
(8)                     if *return_area* is True, returns also the triangle area (0.5*det(J))
(8)                     """
(8)                     J_inv = _safe_inv22_vectorized(J)
```

```
(8)                     Ji00 = J_inv[:, 0, 0]
(8)                     Ji11 = J_inv[:, 1, 1]
(8)                     Ji10 = J_inv[:, 1, 0]
(8)                     Ji01 = J_inv[:, 0, 1]
(8)                     H_rot = _to_matrix_vectorized([
(12)                        [Ji00*Ji00, Ji10*Ji10, Ji00*Ji10],
(12)                        [Ji01*Ji01, Ji11*Ji11, Ji01*Ji11],
(12)                        [2*Ji00*Ji01, 2*Ji11*Ji10, Ji00*Ji11+Ji10*Ji01]])
(8)                     if not return_area:
(12)                        return H_rot
(8)                     else:
(12)                        area = 0.5 * (J[:, 0, 0]*J[:, 1, 1] - J[:, 0, 1]*J[:, 1, 0])
(12)                        return H_rot, area
(4)                 def get_Kff_and_Ff(self, J, ecc, triangles, Uc):
(8)                     """
(8)                     Build K and F for the following elliptic formulation:
(8)                     minimization of curvature energy with value of function at node
(8)                     imposed and derivatives 'free'.
(8)                     Build the global Kff matrix in cco format.
(8)                     Build the full Ff vec Ff = - Kfc x Uc.
(8)                     Parameters
(8)                     ----------
(8)                     *J* is a (N x 2 x 2) array of jacobian matrices (jacobian matrix at
(8)                     triangle first apex)
(8)                     *ecc* is a (N x 3 x 1) array (array of column-matrices) of triangle
(8)                     eccentricities
(8)                     *triangles* is a (N x 3) array of nodes indexes.
(8)                     *Uc* is (N x 3) array of imposed displacements at nodes
(8)                     Returns
(8)                     -------
(8)                     (Kff_rows, Kff_cols, Kff_vals) Kff matrix in coo format - Duplicate
(8)                     (row, col) entries must be summed.
(8)                     Ff: force vector - dim npts * 3
(8)                     """
(8)                     ntri = np.size(ecc, 0)
(8)                     vec_range = np.arange(ntri, dtype=np.int32)
(8)                     c_indices = np.full(ntri, -1, dtype=np.int32)  # for unused dofs, -1
(8)                     f_dof = [1, 2, 4, 5, 7, 8]
(8)                     c_dof = [0, 3, 6]
(8)                     f_dof_indices = _to_matrix_vectorized([[
(12)                        c_indices, triangles[:, 0]*2, triangles[:, 0]*2+1,
(12)                        c_indices, triangles[:, 1]*2, triangles[:, 1]*2+1,
(12)                        c_indices, triangles[:, 2]*2, triangles[:, 2]*2+1]])
(8)                     expand_indices = np.ones([ntri, 9, 1], dtype=np.int32)
(8)                     f_row_indices = _transpose_vectorized(expand_indices @ f_dof_indices)
(8)                     f_col_indices = expand_indices @ f_dof_indices
(8)                     K_elem = self.get_bending_matrices(J, ecc)
(8)                     Kff_vals = np.ravel(K_elem[np.ix_(vec_range, f_dof, f_dof)])
(8)                     Kff_rows = np.ravel(f_row_indices[np.ix_(vec_range, f_dof, f_dof)])
(8)                     Kff_cols = np.ravel(f_col_indices[np.ix_(vec_range, f_dof, f_dof)])
(8)                     Kfc_elem = K_elem[np.ix_(vec_range, f_dof, c_dof)]
(8)                     Uc_elem = np.expand_dims(Uc, axis=2)
(8)                     Ff_elem = -(Kfc_elem @ Uc_elem)[:, :, 0]
(8)                     Ff_indices = f_dof_indices[np.ix_(vec_range, [0], f_dof)][:, 0, :]
(8)                     Ff = np.bincount(np.ravel(Ff_indices), weights=np.ravel(Ff_elem))
(8)                     return Kff_rows, Kff_cols, Kff_vals, Ff
(0)             class _DOF_estimator:
(4)                 """
(4)                 Abstract base class for classes used to estimate a function's first
(4)                 derivatives, and deduce the dofs for a CubicTriInterpolator using a
(4)                 reduced HCT element formulation.
(4)                 Derived classes implement ``compute_df(self, **kwargs)``, returning
(4)                 ``np.vstack([dfx, dfy]).T`` where ``dfx, dfy`` are the estimation of the 2
(4)                 gradient coordinates.
(4)                 """
(4)                 def __init__(self, interpolator, **kwargs):
(8)                     _api.check_isinstance(CubicTriInterpolator, interpolator=interpolator)
(8)                     self._pts = interpolator._pts
(8)                     self._tris_pts = interpolator._tris_pts
```

```
        self.z = interpolator._z
        self._triangles = interpolator._triangles
        (self._unit_x, self._unit_y) = (interpolator._unit_x,
                                        interpolator._unit_y)
        self.dz = self.compute_dz(**kwargs)
        self.compute_dof_from_df()
    def compute_dz(self, **kwargs):
        raise NotImplementedError
    def compute_dof_from_df(self):
        """
        Compute reduced-HCT elements degrees of freedom, from the gradient.
        """
        J = CubicTriInterpolator._get_jacobian(self._tris_pts)
        tri_z = self.z[self._triangles]
        tri_dz = self.dz[self._triangles]
        tri_dof = self.get_dof_vec(tri_z, tri_dz, J)
        return tri_dof
    @staticmethod
    def get_dof_vec(tri_z, tri_dz, J):
        """
        Compute the dof vector of a triangle, from the value of f, df and
        of the local Jacobian at each node.
        Parameters
        ----------
        tri_z : shape (3,) array
            f nodal values.
        tri_dz : shape (3, 2) array
            df/dx, df/dy nodal values.
        J
            Jacobian matrix in local basis of apex 0.
        Returns
        -------
        dof : shape (9,) array
            For each apex ``iapex``::
                dof[iapex*3+0] = f(Ai)
                dof[iapex*3+1] = df(Ai).(AiAi+)
                dof[iapex*3+2] = df(Ai).(AiAi-)
        """
        npt = tri_z.shape[0]
        dof = np.zeros([npt, 9], dtype=np.float64)
        J1 = _ReducedHCT_Element.J0_to_J1 @ J
        J2 = _ReducedHCT_Element.J0_to_J2 @ J
        col0 = J @ np.expand_dims(tri_dz[:, 0, :], axis=2)
        col1 = J1 @ np.expand_dims(tri_dz[:, 1, :], axis=2)
        col2 = J2 @ np.expand_dims(tri_dz[:, 2, :], axis=2)
        dfdksi = _to_matrix_vectorized([
            [col0[:, 0, 0], col1[:, 0, 0], col2[:, 0, 0]],
            [col0[:, 1, 0], col1[:, 1, 0], col2[:, 1, 0]]])
        dof[:, 0:7:3] = tri_z
        dof[:, 1:8:3] = dfdksi[:, 0]
        dof[:, 2:9:3] = dfdksi[:, 1]
        return dof
class _DOF_estimator_user(_DOF_estimator):
    """dz is imposed by user; accounts for scaling if any."""
    def compute_dz(self, dz):
        (dzdx, dzdy) = dz
        dzdx = dzdx * self._unit_x
        dzdy = dzdy * self._unit_y
        return np.vstack([dzdx, dzdy]).T
class _DOF_estimator_geom(_DOF_estimator):
    """Fast 'geometric' approximation, recommended for large arrays."""
    def compute_dz(self):
        """
        self.df is computed as weighted average of _triangles sharing a common
        node. On each triangle itri f is first assumed linear (= ~f), which
        allows to compute d~f[itri]
        Then the following approximation of df nodal values is then proposed:
            f[ipt] = SUM ( w[itri] x d~f[itri] , for itri sharing apex ipt)
        The weighted coeff. w[itri] are proportional to the angle of the
```

```
 853: (8)                    triangle itri at apex ipt
 854: (8)                    """
 855: (8)                    el_geom_w = self.compute_geom_weights()
 856: (8)                    el_geom_grad = self.compute_geom_grads()
 857: (8)                    w_node_sum = np.bincount(np.ravel(self._triangles),
 858: (33)                                            weights=np.ravel(el_geom_w))
 859: (8)                    dfx_el_w = np.empty_like(el_geom_w)
 860: (8)                    dfy_el_w = np.empty_like(el_geom_w)
 861: (8)                    for iapex in range(3):
 862: (12)                       dfx_el_w[:, iapex] = el_geom_w[:, iapex]*el_geom_grad[:, 0]
 863: (12)                       dfy_el_w[:, iapex] = el_geom_w[:, iapex]*el_geom_grad[:, 1]
 864: (8)                    dfx_node_sum = np.bincount(np.ravel(self._triangles),
 865: (35)                                              weights=np.ravel(dfx_el_w))
 866: (8)                    dfy_node_sum = np.bincount(np.ravel(self._triangles),
 867: (35)                                              weights=np.ravel(dfy_el_w))
 868: (8)                    dfx_estim = dfx_node_sum/w_node_sum
 869: (8)                    dfy_estim = dfy_node_sum/w_node_sum
 870: (8)                    return np.vstack([dfx_estim, dfy_estim]).T
 871: (4)                def compute_geom_weights(self):
 872: (8)                    """
 873: (8)                    Build the (nelems, 3) weights coeffs of _triangles angles,
 874: (8)                    renormalized so that np.sum(weights, axis=1) == np.ones(nelems)
 875: (8)                    """
 876: (8)                    weights = np.zeros([np.size(self._triangles, 0), 3])
 877: (8)                    tris_pts = self._tris_pts
 878: (8)                    for ipt in range(3):
 879: (12)                       p0 = tris_pts[:, ipt % 3, :]
 880: (12)                       p1 = tris_pts[:, (ipt+1) % 3, :]
 881: (12)                       p2 = tris_pts[:, (ipt-1) % 3, :]
 882: (12)                       alpha1 = np.arctan2(p1[:, 1]-p0[:, 1], p1[:, 0]-p0[:, 0])
 883: (12)                       alpha2 = np.arctan2(p2[:, 1]-p0[:, 1], p2[:, 0]-p0[:, 0])
 884: (12)                       angle = np.abs(((alpha2-alpha1) / np.pi) % 1)
 885: (12)                       weights[:, ipt] = 0.5 - np.abs(angle-0.5)
 886: (8)                    return weights
 887: (4)                def compute_geom_grads(self):
 888: (8)                    """
 889: (8)                    Compute the (global) gradient component of f assumed linear (~f).
 890: (8)                    returns array df of shape (nelems, 2)
 891: (8)                    df[ielem].dM[ielem] = dz[ielem] i.e. df = dz x dM = dM.T^-1 x dz
 892: (8)                    """
 893: (8)                    tris_pts = self._tris_pts
 894: (8)                    tris_f = self.z[self._triangles]
 895: (8)                    dM1 = tris_pts[:, 1, :] - tris_pts[:, 0, :]
 896: (8)                    dM2 = tris_pts[:, 2, :] - tris_pts[:, 0, :]
 897: (8)                    dM = np.dstack([dM1, dM2])
 898: (8)                    dM_inv = _safe_inv22_vectorized(dM)
 899: (8)                    dZ1 = tris_f[:, 1] - tris_f[:, 0]
 900: (8)                    dZ2 = tris_f[:, 2] - tris_f[:, 0]
 901: (8)                    dZ = np.vstack([dZ1, dZ2]).T
 902: (8)                    df = np.empty_like(dZ)
 903: (8)                    df[:, 0] = dZ[:, 0]*dM_inv[:, 0, 0] + dZ[:, 1]*dM_inv[:, 1, 0]
 904: (8)                    df[:, 1] = dZ[:, 0]*dM_inv[:, 0, 1] + dZ[:, 1]*dM_inv[:, 1, 1]
 905: (8)                    return df
 906: (0)            class _DOF_estimator_min_E(_DOF_estimator_geom):
 907: (4)                """
 908: (4)                The 'smoothest' approximation, df is computed through global minimization
 909: (4)                of the bending energy:
 910: (6)                  E(f) = integral[(d2z/dx2 + d2z/dy2 + 2 d2z/dxdy)**2 dA]
 911: (4)                """
 912: (4)                def __init__(self, Interpolator):
 913: (8)                    self._eccs = Interpolator._eccs
 914: (8)                    super().__init__(Interpolator)
 915: (4)                def compute_dz(self):
 916: (8)                    """
 917: (8)                    Elliptic solver for bending energy minimization.
 918: (8)                    Uses a dedicated 'toy' sparse Jacobi PCG solver.
 919: (8)                    """
 920: (8)                    dz_init = super().compute_dz()
 921: (8)                    Uf0 = np.ravel(dz_init)
```

```
922: (8)                    reference_element = _ReducedHCT_Element()
923: (8)                    J = CubicTriInterpolator._get_jacobian(self._tris_pts)
924: (8)                    eccs = self._eccs
925: (8)                    triangles = self._triangles
926: (8)                    Uc = self.z[self._triangles]
927: (8)                    Kff_rows, Kff_cols, Kff_vals, Ff = reference_element.get_Kff_and_Ff(
928: (12)                       J, eccs, triangles, Uc)
929: (8)                    tol = 1.e-10
930: (8)                    n_dof = Ff.shape[0]
931: (8)                    Kff_coo = _Sparse_Matrix_coo(Kff_vals, Kff_rows, Kff_cols,
932: (37)                                                shape=(n_dof, n_dof))
933: (8)                    Kff_coo.compress_csc()
934: (8)                    Uf, err = _cg(A=Kff_coo, b=Ff, x0=Uf0, tol=tol)
935: (8)                    err0 = np.linalg.norm(Kff_coo.dot(Uf0) - Ff)
936: (8)                    if err0 < err:
937: (12)                       _api.warn_external("In TriCubicInterpolator initialization, "
938: (31)                                          "PCG sparse solver did not converge after "
939: (31)                                          "1000 iterations. `geom` approximation is "
940: (31)                                          "used instead of `min_E`")
941: (12)                       Uf = Uf0
942: (8)                    dz = np.empty([self._pts.shape[0], 2], dtype=np.float64)
943: (8)                    dz[:, 0] = Uf[::2]
944: (8)                    dz[:, 1] = Uf[1::2]
945: (8)                    return dz
946: (0)            class _Sparse_Matrix_coo:
947: (4)                def __init__(self, vals, rows, cols, shape):
948: (8)                    """
949: (8)                    Create a sparse matrix in coo format.
950: (8)                    *vals*: arrays of values of non-null entries of the matrix
951: (8)                    *rows*: int arrays of rows of non-null entries of the matrix
952: (8)                    *cols*: int arrays of cols of non-null entries of the matrix
953: (8)                    *shape*: 2-tuple (n, m) of matrix shape
954: (8)                    """
955: (8)                    self.n, self.m = shape
956: (8)                    self.vals = np.asarray(vals, dtype=np.float64)
957: (8)                    self.rows = np.asarray(rows, dtype=np.int32)
958: (8)                    self.cols = np.asarray(cols, dtype=np.int32)
959: (4)                def dot(self, V):
960: (8)                    """
961: (8)                    Dot product of self by a vector *V* in sparse-dense to dense format
962: (8)                    *V* dense vector of shape (self.m,).
963: (8)                    """
964: (8)                    assert V.shape == (self.m,)
965: (8)                    return np.bincount(self.rows,
966: (27)                                      weights=self.vals*V[self.cols],
967: (27)                                      minlength=self.m)
968: (4)                def compress_csc(self):
969: (8)                    """
970: (8)                    Compress rows, cols, vals / summing duplicates. Sort for csc format.
971: (8)                    """
972: (8)                    _, unique, indices = np.unique(
973: (12)                       self.rows + self.n*self.cols,
974: (12)                       return_index=True, return_inverse=True)
975: (8)                    self.rows = self.rows[unique]
976: (8)                    self.cols = self.cols[unique]
977: (8)                    self.vals = np.bincount(indices, weights=self.vals)
978: (4)                def compress_csr(self):
979: (8)                    """
980: (8)                    Compress rows, cols, vals / summing duplicates. Sort for csr format.
981: (8)                    """
982: (8)                    _, unique, indices = np.unique(
983: (12)                       self.m*self.rows + self.cols,
984: (12)                       return_index=True, return_inverse=True)
985: (8)                    self.rows = self.rows[unique]
986: (8)                    self.cols = self.cols[unique]
987: (8)                    self.vals = np.bincount(indices, weights=self.vals)
988: (4)                def to_dense(self):
989: (8)                    """
990: (8)                    Return a dense matrix representing self, mainly for debugging
```

```
purposes.
991: (8)                    """
992: (8)                    ret = np.zeros([self.n, self.m], dtype=np.float64)
993: (8)                    nvals = self.vals.size
994: (8)                    for i in range(nvals):
995: (12)                       ret[self.rows[i], self.cols[i]] += self.vals[i]
996: (8)                    return ret
997: (4)                def __str__(self):
998: (8)                    return self.to_dense().__str__()
999: (4)                @property
1000: (4)               def diag(self):
1001: (8)                   """Return the (dense) vector of the diagonal elements."""
1002: (8)                   in_diag = (self.rows == self.cols)
1003: (8)                   diag = np.zeros(min(self.n, self.n), dtype=np.float64)  # default 0.
1004: (8)                   diag[self.rows[in_diag]] = self.vals[in_diag]
1005: (8)                   return diag
1006: (0)           def _cg(A, b, x0=None, tol=1.e-10, maxiter=1000):
1007: (4)               """
1008: (4)               Use Preconditioned Conjugate Gradient iteration to solve A x = b
1009: (4)               A simple Jacobi (diagonal) preconditioner is used.
1010: (4)               Parameters
1011: (4)               ----------
1012: (4)               A : _Sparse_Matrix_coo
1013: (8)                   *A* must have been compressed before by compress_csc or
1014: (8)                   compress_csr method.
1015: (4)               b : array
1016: (8)                   Right hand side of the linear system.
1017: (4)               x0 : array, optional
1018: (8)                   Starting guess for the solution. Defaults to the zero vector.
1019: (4)               tol : float, optional
1020: (8)                   Tolerance to achieve. The algorithm terminates when the relative
1021: (8)                   residual is below tol. Default is 1e-10.
1022: (4)               maxiter : int, optional
1023: (8)                   Maximum number of iterations.  Iteration will stop after *maxiter*
1024: (8)                   steps even if the specified tolerance has not been achieved. Defaults
1025: (8)                   to 1000.
1026: (4)               Returns
1027: (4)               -------
1028: (4)               x : array
1029: (8)                   The converged solution.
1030: (4)               err : float
1031: (8)                   The absolute error np.linalg.norm(A.dot(x) - b)
1032: (4)               """
1033: (4)               n = b.size
1034: (4)               assert A.n == n
1035: (4)               assert A.m == n
1036: (4)               b_norm = np.linalg.norm(b)
1037: (4)               kvec = A.diag
1038: (4)               kvec = np.maximum(kvec, 1e-6)
1039: (4)               if x0 is None:
1040: (8)                   x = np.zeros(n)
1041: (4)               else:
1042: (8)                   x = x0
1043: (4)               r = b - A.dot(x)
1044: (4)               w = r/kvec
1045: (4)               p = np.zeros(n)
1046: (4)               beta = 0.0
1047: (4)               rho = np.dot(r, w)
1048: (4)               k = 0
1049: (4)               while (np.sqrt(abs(rho)) > tol*b_norm) and (k < maxiter):
1050: (8)                   p = w + beta*p
1051: (8)                   z = A.dot(p)
1052: (8)                   alpha = rho/np.dot(p, z)
1053: (8)                   r = r - alpha*z
1054: (8)                   w = r/kvec
1055: (8)                   rhoold = rho
1056: (8)                   rho = np.dot(r, w)
1057: (8)                   x = x + alpha*p
1058: (8)                   beta = rho/rhoold
```

```
(8)                 k += 1
(4)             err = np.linalg.norm(A.dot(x) - b)
(4)             return x, err
(0)         def _safe_inv22_vectorized(M):
(4)             """
(4)             Inversion of arrays of (2, 2) matrices, returns 0 for rank-deficient
(4)             matrices.
(4)             *M* : array of (2, 2) matrices to inverse, shape (n, 2, 2)
(4)             """
(4)             _api.check_shape((None, 2, 2), M=M)
(4)             M_inv = np.empty_like(M)
(4)             prod1 = M[:, 0, 0]*M[:, 1, 1]
(4)             delta = prod1 - M[:, 0, 1]*M[:, 1, 0]
(4)             rank2 = (np.abs(delta) > 1e-8*np.abs(prod1))
(4)             if np.all(rank2):
(8)                 delta_inv = 1./delta
(4)             else:
(8)                 delta_inv = np.zeros(M.shape[0])
(8)                 delta_inv[rank2] = 1./delta[rank2]
(4)             M_inv[:, 0, 0] = M[:, 1, 1]*delta_inv
(4)             M_inv[:, 0, 1] = -M[:, 0, 1]*delta_inv
(4)             M_inv[:, 1, 0] = -M[:, 1, 0]*delta_inv
(4)             M_inv[:, 1, 1] = M[:, 0, 0]*delta_inv
(4)             return M_inv
(0)         def _pseudo_inv22sym_vectorized(M):
(4)             """
(4)             Inversion of arrays of (2, 2) SYMMETRIC matrices; returns the
(4)             (Moore-Penrose) pseudo-inverse for rank-deficient matrices.
(4)             In case M is of rank 1, we have M = trace(M) x P where P is the orthogonal
(4)             projection on Im(M), and we return trace(M)^-1 x P == M / trace(M)**2
(4)             In case M is of rank 0, we return the null matrix.
(4)             *M* : array of (2, 2) matrices to inverse, shape (n, 2, 2)
(4)             """
(4)             _api.check_shape((None, 2, 2), M=M)
(4)             M_inv = np.empty_like(M)
(4)             prod1 = M[:, 0, 0]*M[:, 1, 1]
(4)             delta = prod1 - M[:, 0, 1]*M[:, 1, 0]
(4)             rank2 = (np.abs(delta) > 1e-8*np.abs(prod1))
(4)             if np.all(rank2):
(8)                 M_inv[:, 0, 0] = M[:, 1, 1] / delta
(8)                 M_inv[:, 0, 1] = -M[:, 0, 1] / delta
(8)                 M_inv[:, 1, 0] = -M[:, 1, 0] / delta
(8)                 M_inv[:, 1, 1] = M[:, 0, 0] / delta
(4)             else:
(8)                 delta = delta[rank2]
(8)                 M_inv[rank2, 0, 0] = M[rank2, 1, 1] / delta
(8)                 M_inv[rank2, 0, 1] = -M[rank2, 0, 1] / delta
(8)                 M_inv[rank2, 1, 0] = -M[rank2, 1, 0] / delta
(8)                 M_inv[rank2, 1, 1] = M[rank2, 0, 0] / delta
(8)                 rank01 = ~rank2
(8)                 tr = M[rank01, 0, 0] + M[rank01, 1, 1]
(8)                 tr_zeros = (np.abs(tr) < 1.e-8)
(8)                 sq_tr_inv = (1.-tr_zeros) / (tr**2+tr_zeros)
(8)                 M_inv[rank01, 0, 0] = M[rank01, 0, 0] * sq_tr_inv
(8)                 M_inv[rank01, 0, 1] = M[rank01, 0, 1] * sq_tr_inv
(8)                 M_inv[rank01, 1, 0] = M[rank01, 1, 0] * sq_tr_inv
(8)                 M_inv[rank01, 1, 1] = M[rank01, 1, 1] * sq_tr_inv
(4)             return M_inv
(0)         def _scalar_vectorized(scalar, M):
(4)             """
(4)             Scalar product between scalars and matrices.
(4)             """
(4)             return scalar[:, np.newaxis, np.newaxis]*M
(0)         def _transpose_vectorized(M):
(4)             """
(4)             Transposition of an array of matrices *M*.
(4)             """
(4)             return np.transpose(M, [0, 2, 1])
(0)         def _roll_vectorized(M, roll_indices, axis):
```

```
(4)             """
(4)             Roll an array of matrices along *axis* (0: rows, 1: columns) according to
(4)             an array of indices *roll_indices*.
(4)             """
(4)             assert axis in [0, 1]
(4)             ndim = M.ndim
(4)             assert ndim == 3
(4)             ndim_roll = roll_indices.ndim
(4)             assert ndim_roll == 1
(4)             sh = M.shape
(4)             r, c = sh[-2:]
(4)             assert sh[0] == roll_indices.shape[0]
(4)             vec_indices = np.arange(sh[0], dtype=np.int32)
(4)             M_roll = np.empty_like(M)
(4)             if axis == 0:
(8)                 for ir in range(r):
(12)                    for ic in range(c):
(16)                        M_roll[:, ir, ic] = M[vec_indices, (-roll_indices+ir) % r, ic]
(4)             else:  # 1
(8)                 for ir in range(r):
(12)                    for ic in range(c):
(16)                        M_roll[:, ir, ic] = M[vec_indices, ir, (-roll_indices+ic) % c]
(4)             return M_roll
(0)         def _to_matrix_vectorized(M):
(4)             """
(4)             Build an array of matrices from individuals np.arrays of identical shapes.
(4)             Parameters
(4)             ----------
(4)             M
(8)                 ncols-list of nrows-lists of shape sh.
(4)             Returns
(4)             -------
(4)             M_res : np.array of shape (sh, nrow, ncols)
(8)                 *M_res* satisfies ``M_res[..., i, j] = M[i][j]``.
(4)             """
(4)             assert isinstance(M, (tuple, list))
(4)             assert all(isinstance(item, (tuple, list)) for item in M)
(4)             c_vec = np.asarray([len(item) for item in M])
(4)             assert np.all(c_vec-c_vec[0] == 0)
(4)             r = len(M)
(4)             c = c_vec[0]
(4)             M00 = np.asarray(M[0][0])
(4)             dt = M00.dtype
(4)             sh = [M00.shape[0], r, c]
(4)             M_ret = np.empty(sh, dtype=dt)
(4)             for irow in range(r):
(8)                 for icol in range(c):
(12)                    M_ret[:, irow, icol] = np.asarray(M[irow][icol])
(4)             return M_ret
(0)         def _extract_submatrices(M, block_indices, block_size, axis):
(4)             """
(4)             Extract selected blocks of a matrices *M* depending on parameters
(4)             *block_indices* and *block_size*.
(4)             Returns the array of extracted matrices *Mres* so that ::
(8)                 M_res[..., ir, :] = M[(block_indices*block_size+ir), :]
(4)             """
(4)             assert block_indices.ndim == 1
(4)             assert axis in [0, 1]
(4)             r, c = M.shape
(4)             if axis == 0:
(8)                 sh = [block_indices.shape[0], block_size, c]
(4)             else:  # 1
(8)                 sh = [block_indices.shape[0], r, block_size]
(4)             dt = M.dtype
(4)             M_res = np.empty(sh, dtype=dt)
(4)             if axis == 0:
(8)                 for ir in range(block_size):
(12)                    M_res[:, ir, :] = M[(block_indices*block_size+ir), :]
(4)             else:  # 1
```

```
1197: (8)                    for ic in range(block_size):
1198: (12)                       M_res[:, :, ic] = M[:, (block_indices*block_size+ic)]
1199: (4)                return M_res

----------------------------------------

File 236 - _tripcolor.py:

1: (0)              import numpy as np
2: (0)              from matplotlib import _api
3: (0)              from matplotlib.collections import PolyCollection, TriMesh
4: (0)              from matplotlib.tri._triangulation import Triangulation
5: (0)              def tripcolor(ax, *args, alpha=1.0, norm=None, cmap=None, vmin=None,
6: (14)                           vmax=None, shading='flat', facecolors=None, **kwargs):
7: (4)                  """
8: (4)                  Create a pseudocolor plot of an unstructured triangular grid.
9: (4)                  Call signatures::
10: (6)                   tripcolor(triangulation, c, *, ...)
11: (6)                   tripcolor(x, y, c, *, [triangles=triangles], [mask=mask], ...)
12: (4)                 The triangular grid can be specified either by passing a `.Triangulation`
13: (4)                 object as the first parameter, or by passing the points *x*, *y* and
14: (4)                 optionally the *triangles* and a *mask*. See `.Triangulation` for an
15: (4)                 explanation of these parameters.
16: (4)                 It is possible to pass the triangles positionally, i.e.
17: (4)                 ``tripcolor(x, y, triangles, c, ...)``. However, this is discouraged.
18: (4)                 For more clarity, pass *triangles* via keyword argument.
19: (4)                 If neither of *triangulation* or *triangles* are given, the triangulation
20: (4)                 is calculated on the fly. In this case, it does not make sense to provide
21: (4)                 colors at the triangle faces via *c* or *facecolors* because there are
22: (4)                 multiple possible triangulations for a group of points and you don't know
23: (4)                 which triangles will be constructed.
24: (4)                 Parameters
25: (4)                 ----------
26: (4)                 triangulation : `.Triangulation`
27: (8)                     An already created triangular grid.
28: (4)                 x, y, triangles, mask
29: (8)                     Parameters defining the triangular grid. See `.Triangulation`.
30: (8)                     This is mutually exclusive with specifying *triangulation*.
31: (4)                 c : array-like
32: (8)                     The color values, either for the points or for the triangles. Which
one
33: (8)                     is automatically inferred from the length of *c*, i.e. does it match
34: (8)                     the number of points or the number of triangles. If there are the same
35: (8)                     number of points and triangles in the triangulation it is assumed that
36: (8)                     color values are defined at points; to force the use of color values
at
37: (8)                     triangles use the keyword argument ``facecolors=c`` instead of just
38: (8)                     ``c``.
39: (8)                     This parameter is position-only.
40: (4)                 facecolors : array-like, optional
41: (8)                     Can be used alternatively to *c* to specify colors at the triangle
42: (8)                     faces. This parameter takes precedence over *c*.
43: (4)                 shading : {'flat', 'gouraud'}, default: 'flat'
44: (8)                     If  'flat' and the color values *c* are defined at points, the color
45: (8)                     values used for each triangle are from the mean c of the triangle's
46: (8)                     three points. If *shading* is 'gouraud' then color values must be
47: (8)                     defined at points.
48: (4)                 other_parameters
49: (8)                     All other parameters are the same as for `~.Axes.pcolor`.
50: (4)                 """
51: (4)                 _api.check_in_list(['flat', 'gouraud'], shading=shading)
52: (4)                 tri, args, kwargs = Triangulation.get_from_args_and_kwargs(*args,
**kwargs)
53: (4)                 if facecolors is not None:
54: (8)                     if args:
55: (12)                        _api.warn_external(
56: (16)                            "Positional parameter c has no effect when the keyword "
57: (16)                            "facecolors is given")
58: (8)                     point_colors = None
```

```
59: (8)                     if len(facecolors) != len(tri.triangles):
60: (12)                        raise ValueError("The length of facecolors must match the number "
61: (29)                                         "of triangles")
62: (4)                 else:
63: (8)                     if not args:
64: (12)                        raise TypeError(
65: (16)                            "tripcolor() missing 1 required positional argument: 'c'; or "
66: (16)                            "1 required keyword-only argument: 'facecolors'")
67: (8)                     elif len(args) > 1:
68: (12)                        raise TypeError(f"Unexpected positional parameters: {args[1:]!r}")
69: (8)                     c = np.asarray(args[0])
70: (8)                     if len(c) == len(tri.x):
71: (12)                        point_colors = c
72: (12)                        facecolors = None
73: (8)                     elif len(c) == len(tri.triangles):
74: (12)                        point_colors = None
75: (12)                        facecolors = c
76: (8)                     else:
77: (12)                        raise ValueError('The length of c must match either the number '
78: (29)                                         'of points or the number of triangles')
79: (4)                 linewidths = (0.25,)
80: (4)                 if 'linewidth' in kwargs:
81: (8)                     kwargs['linewidths'] = kwargs.pop('linewidth')
82: (4)                 kwargs.setdefault('linewidths', linewidths)
83: (4)                 edgecolors = 'none'
84: (4)                 if 'edgecolor' in kwargs:
85: (8)                     kwargs['edgecolors'] = kwargs.pop('edgecolor')
86: (4)                 ec = kwargs.setdefault('edgecolors', edgecolors)
87: (4)                 if 'antialiased' in kwargs:
88: (8)                     kwargs['antialiaseds'] = kwargs.pop('antialiased')
89: (4)                 if 'antialiaseds' not in kwargs and ec.lower() == "none":
90: (8)                     kwargs['antialiaseds'] = False
91: (4)                 if shading == 'gouraud':
92: (8)                     if facecolors is not None:
93: (12)                        raise ValueError(
94: (16)                            "shading='gouraud' can only be used when the colors "
95: (16)                            "are specified at the points, not at the faces.")
96: (8)                     collection = TriMesh(tri, alpha=alpha, array=point_colors,
97: (29)                                         cmap=cmap, norm=norm, **kwargs)
98: (4)                 else:  # 'flat'
99: (8)                     maskedTris = tri.get_masked_triangles()
100: (8)                    verts = np.stack((tri.x[maskedTris], tri.y[maskedTris]), axis=-1)
101: (8)                    if facecolors is None:
102: (12)                       colors = point_colors[maskedTris].mean(axis=1)
103: (8)                    elif tri.mask is not None:
104: (12)                       colors = facecolors[~tri.mask]
105: (8)                    else:
106: (12)                       colors = facecolors
107: (8)                    collection = PolyCollection(verts, alpha=alpha, array=colors,
108: (36)                                               cmap=cmap, norm=norm, **kwargs)
109: (4)                collection._scale_norm(norm, vmin, vmax)
110: (4)                ax.grid(False)
111: (4)                minx = tri.x.min()
112: (4)                maxx = tri.x.max()
113: (4)                miny = tri.y.min()
114: (4)                maxy = tri.y.max()
115: (4)                corners = (minx, miny), (maxx, maxy)
116: (4)                ax.update_datalim(corners)
117: (4)                ax.autoscale_view()
118: (4)                ax.add_collection(collection)
119: (4)                return collection

----------------------------------------

File 237 - _triplot.py:

1: (0)              import numpy as np
2: (0)              from matplotlib.tri._triangulation import Triangulation
3: (0)              import matplotlib.cbook as cbook
```

```
 4: (0)              import matplotlib.lines as mlines
 5: (0)              def triplot(ax, *args, **kwargs):
 6: (4)                  """
 7: (4)                  Draw an unstructured triangular grid as lines and/or markers.
 8: (4)                  Call signatures::
 9: (6)                    triplot(triangulation, ...)
10: (6)                    triplot(x, y, [triangles], *, [mask=mask], ...)
11: (4)                  The triangular grid can be specified either by passing a `.Triangulation`
12: (4)                  object as the first parameter, or by passing the points *x*, *y* and
13: (4)                  optionally the *triangles* and a *mask*. If neither of *triangulation* or
14: (4)                  *triangles* are given, the triangulation is calculated on the fly.
15: (4)                  Parameters
16: (4)                  ----------
17: (4)                  triangulation : `.Triangulation`
18: (8)                      An already created triangular grid.
19: (4)                  x, y, triangles, mask
20: (8)                      Parameters defining the triangular grid. See `.Triangulation`.
21: (8)                      This is mutually exclusive with specifying *triangulation*.
22: (4)                  other_parameters
23: (8)                      All other args and kwargs are forwarded to `~.Axes.plot`.
24: (4)                  Returns
25: (4)                  -------
26: (4)                  lines : `~matplotlib.lines.Line2D`
27: (8)                      The drawn triangles edges.
28: (4)                  markers : `~matplotlib.lines.Line2D`
29: (8)                      The drawn marker nodes.
30: (4)                  """
31: (4)                  import matplotlib.axes
32: (4)                  tri, args, kwargs = Triangulation.get_from_args_and_kwargs(*args,
**kwargs)
33: (4)                  x, y, edges = (tri.x, tri.y, tri.edges)
34: (4)                  fmt = args[0] if args else ""
35: (4)                  linestyle, marker, color = matplotlib.axes._base._process_plot_format(fmt)
36: (4)                  kw = cbook.normalize_kwargs(kwargs, mlines.Line2D)
37: (4)                  for key, val in zip(('linestyle', 'marker', 'color'),
38: (24)                                     (linestyle, marker, color)):
39: (8)                      if val is not None:
40: (12)                         kw.setdefault(key, val)
41: (4)                  linestyle = kw['linestyle']
42: (4)                  kw_lines = {
43: (8)                      **kw,
44: (8)                      'marker': 'None',  # No marker to draw.
45: (8)                      'zorder': kw.get('zorder', 1),  # Path default zorder is used.
46: (4)                  }
47: (4)                  if linestyle not in [None, 'None', '', ' ']:
48: (8)                      tri_lines_x = np.insert(x[edges], 2, np.nan, axis=1)
49: (8)                      tri_lines_y = np.insert(y[edges], 2, np.nan, axis=1)
50: (8)                      tri_lines = ax.plot(tri_lines_x.ravel(), tri_lines_y.ravel(),
51: (28)                                         **kw_lines)
52: (4)                  else:
53: (8)                      tri_lines = ax.plot([], [], **kw_lines)
54: (4)                  marker = kw['marker']
55: (4)                  kw_markers = {
56: (8)                      **kw,
57: (8)                      'linestyle': 'None',  # No line to draw.
58: (4)                  }
59: (4)                  kw_markers.pop('label', None)
60: (4)                  if marker not in [None, 'None', '', ' ']:
61: (8)                      tri_markers = ax.plot(x, y, **kw_markers)
62: (4)                  else:
63: (8)                      tri_markers = ax.plot([], [], **kw_markers)
64: (4)                  return tri_lines + tri_markers
```

File 238 - _trirefine.py:

```
 1: (0)              """
 2: (0)              Mesh refinement for triangular grids.
```

```
3: (0)              """
4: (0)              import numpy as np
5: (0)              from matplotlib import _api
6: (0)              from matplotlib.tri._triangulation import Triangulation
7: (0)              import matplotlib.tri._triinterpolate
8: (0)              class TriRefiner:
9: (4)                  """
10: (4)                 Abstract base class for classes implementing mesh refinement.
11: (4)                 A TriRefiner encapsulates a Triangulation object and provides tools for
12: (4)                 mesh refinement and interpolation.
13: (4)                 Derived classes must implement:
14: (4)                 - ``refine_triangulation(return_tri_index=False, **kwargs)`` , where
15: (6)                   the optional keyword arguments *kwargs* are defined in each
16: (6)                   TriRefiner concrete implementation, and which returns:
17: (6)                   - a refined triangulation,
18: (6)                   - optionally (depending on *return_tri_index*), for each
19: (8)                     point of the refined triangulation: the index of
20: (8)                     the initial triangulation triangle to which it belongs.
21: (4)                 - ``refine_field(z, triinterpolator=None, **kwargs)``, where:
22: (6)                   - *z* array of field values (to refine) defined at the base
23: (8)                     triangulation nodes,
24: (6)                   - *triinterpolator* is an optional `~matplotlib.tri.TriInterpolator`,
25: (6)                   - the other optional keyword arguments *kwargs* are defined in
26: (8)                     each TriRefiner concrete implementation;
27: (6)                   and which returns (as a tuple) a refined triangular mesh and the
28: (6)                   interpolated values of the field at the refined triangulation nodes.
29: (4)                 """
30: (4)                 def __init__(self, triangulation):
31: (8)                     _api.check_isinstance(Triangulation, triangulation=triangulation)
32: (8)                     self._triangulation = triangulation
33: (0)             class UniformTriRefiner(TriRefiner):
34: (4)                 """
35: (4)                 Uniform mesh refinement by recursive subdivisions.
36: (4)                 Parameters
37: (4)                 ----------
38: (4)                 triangulation : `~matplotlib.tri.Triangulation`
39: (8)                     The encapsulated triangulation (to be refined)
40: (4)                 """
41: (4)                 def __init__(self, triangulation):
42: (8)                     super().__init__(triangulation)
43: (4)                 def refine_triangulation(self, return_tri_index=False, subdiv=3):
44: (8)                     """
45: (8)                     Compute a uniformly refined triangulation *refi_triangulation* of
46: (8)                     the encapsulated :attr:`triangulation`.
47: (8)                     This function refines the encapsulated triangulation by splitting each
48: (8)                     father triangle into 4 child sub-triangles built on the edges midside
49: (8)                     nodes, recursing *subdiv* times.  In the end, each triangle is hence
50: (8)                     divided into ``4**subdiv`` child triangles.
51: (8)                     Parameters
52: (8)                     ----------
53: (8)                     return_tri_index : bool, default: False
54: (12)                        Whether an index table indicating the father triangle index of
each
55: (12)                        point is returned.
56: (8)                     subdiv : int, default: 3
57: (12)                        Recursion level for the subdivision.
58: (12)                        Each triangle is divided into ``4**subdiv`` child triangles;
59: (12)                        hence, the default results in 64 refined subtriangles for each
60: (12)                        triangle of the initial triangulation.
61: (8)                     Returns
62: (8)                     -------
63: (8)                     refi_triangulation : `~matplotlib.tri.Triangulation`
64: (12)                        The refined triangulation.
65: (8)                     found_index : int array
66: (12)                        Index of the initial triangulation containing triangle, for each
67: (12)                        point of *refi_triangulation*.
68: (12)                        Returned only if *return_tri_index* is set to True.
69: (8)                     """
70: (8)                     refi_triangulation = self._triangulation
```

```
 71: (8)                     ntri = refi_triangulation.triangles.shape[0]
 72: (8)                     ancestors = np.arange(ntri, dtype=np.int32)
 73: (8)                     for _ in range(subdiv):
 74: (12)                        refi_triangulation, ancestors = self._refine_triangulation_once(
 75: (16)                            refi_triangulation, ancestors)
 76: (8)                     refi_npts = refi_triangulation.x.shape[0]
 77: (8)                     refi_triangles = refi_triangulation.triangles
 78: (8)                     if return_tri_index:
 79: (12)                        found_index = np.full(refi_npts, -1, dtype=np.int32)
 80: (12)                        tri_mask = self._triangulation.mask
 81: (12)                        if tri_mask is None:
 82: (16)                            found_index[refi_triangles] = np.repeat(ancestors,
 83: (56)                                                                    3).reshape(-1, 3)
 84: (12)                        else:
 85: (16)                            ancestor_mask = tri_mask[ancestors]
 86: (16)                            found_index[refi_triangles[ancestor_mask, :]
 87: (28)                                        ] = np.repeat(ancestors[ancestor_mask],
 88: (42)                                                      3).reshape(-1, 3)
 89: (16)                            found_index[refi_triangles[~ancestor_mask, :]
 90: (28)                                        ] = np.repeat(ancestors[~ancestor_mask],
 91: (42)                                                      3).reshape(-1, 3)
 92: (12)                        return refi_triangulation, found_index
 93: (8)                     else:
 94: (12)                        return refi_triangulation
 95: (4)                 def refine_field(self, z, triinterpolator=None, subdiv=3):
 96: (8)                     """
 97: (8)                     Refine a field defined on the encapsulated triangulation.
 98: (8)                     Parameters
 99: (8)                     ----------
100: (8)                     z : (npoints,) array-like
101: (12)                        Values of the field to refine, defined at the nodes of the
102: (12)                        encapsulated triangulation. (``n_points`` is the number of points
103: (12)                        in the initial triangulation)
104: (8)                     triinterpolator : `~matplotlib.tri.TriInterpolator`, optional
105: (12)                        Interpolator used for field interpolation. If not specified,
106: (12)                        a `~matplotlib.tri.CubicTriInterpolator` will be used.
107: (8)                     subdiv : int, default: 3
108: (12)                        Recursion level for the subdivision.
109: (12)                        Each triangle is divided into ``4**subdiv`` child triangles.
110: (8)                     Returns
111: (8)                     -------
112: (8)                     refi_tri : `~matplotlib.tri.Triangulation`
113: (13)                         The returned refined triangulation.
114: (8)                     refi_z : 1D array of length: *refi_tri* node count.
115: (13)                         The returned interpolated field (at *refi_tri* nodes).
116: (8)                     """
117: (8)                     if triinterpolator is None:
118: (12)                        interp = matplotlib.tri.CubicTriInterpolator(
119: (16)                            self._triangulation, z)
120: (8)                     else:
121: (12)                        _api.check_isinstance(matplotlib.tri.TriInterpolator,
122: (34)                                              triinterpolator=triinterpolator)
123: (12)                        interp = triinterpolator
124: (8)                     refi_tri, found_index = self.refine_triangulation(
125: (12)                        subdiv=subdiv, return_tri_index=True)
126: (8)                     refi_z = interp._interpolate_multikeys(
127: (12)                        refi_tri.x, refi_tri.y, tri_index=found_index)[0]
128: (8)                     return refi_tri, refi_z
129: (4)                 @staticmethod
130: (4)                 def _refine_triangulation_once(triangulation, ancestors=None):
131: (8)                     """
132: (8)                     Refine a `.Triangulation` by splitting each triangle into 4
133: (8)                     child-masked_triangles built on the edges midside nodes.
134: (8)                     Masked triangles, if present, are also split, but their children
135: (8)                     returned masked.
136: (8)                     If *ancestors* is not provided, returns only a new triangulation:
137: (8)                     child_triangulation.
138: (8)                     If the array-like key table *ancestor* is given, it shall be of shape
139: (8)                     (ntri,) where ntri is the number of *triangulation* masked_triangles.
```

```
(8)                     In this case, the function returns
(8)                     (child_triangulation, child_ancestors)
(8)                     child_ancestors is defined so that the 4 child masked_triangles share
(8)                     the same index as their father: child_ancestors.shape = (4 * ntri,).
(8)                     """
(8)                     x = triangulation.x
(8)                     y = triangulation.y
(8)                     neighbors = triangulation.neighbors
(8)                     triangles = triangulation.triangles
(8)                     npts = np.shape(x)[0]
(8)                     ntri = np.shape(triangles)[0]
(8)                     if ancestors is not None:
(12)                        ancestors = np.asarray(ancestors)
(12)                        if np.shape(ancestors) != (ntri,):
(16)                            raise ValueError(
(20)                                "Incompatible shapes provide for "
(20)                                "triangulation.masked_triangles and ancestors: "
(20)                                f"{np.shape(triangles)} and {np.shape(ancestors)}")
(8)                     borders = np.sum(neighbors == -1)
(8)                     added_pts = (3*ntri + borders) // 2
(8)                     refi_npts = npts + added_pts
(8)                     refi_x = np.zeros(refi_npts)
(8)                     refi_y = np.zeros(refi_npts)
(8)                     refi_x[:npts] = x
(8)                     refi_y[:npts] = y
(8)                     edge_elems = np.tile(np.arange(ntri, dtype=np.int32), 3)
(8)                     edge_apexes = np.repeat(np.arange(3, dtype=np.int32), ntri)
(8)                     edge_neighbors = neighbors[edge_elems, edge_apexes]
(8)                     mask_masters = (edge_elems > edge_neighbors)
(8)                     masters = edge_elems[mask_masters]
(8)                     apex_masters = edge_apexes[mask_masters]
(8)                     x_add = (x[triangles[masters, apex_masters]] +
(17)                             x[triangles[masters, (apex_masters+1) % 3]]) * 0.5
(8)                     y_add = (y[triangles[masters, apex_masters]] +
(17)                             y[triangles[masters, (apex_masters+1) % 3]]) * 0.5
(8)                     refi_x[npts:] = x_add
(8)                     refi_y[npts:] = y_add
(8)                     new_pt_corner = triangles
(8)                     new_pt_midside = np.empty([ntri, 3], dtype=np.int32)
(8)                     cum_sum = npts
(8)                     for imid in range(3):
(12)                        mask_st_loc = (imid == apex_masters)
(12)                        n_masters_loc = np.sum(mask_st_loc)
(12)                        elem_masters_loc = masters[mask_st_loc]
(12)                        new_pt_midside[:, imid][elem_masters_loc] = np.arange(
(16)                            n_masters_loc, dtype=np.int32) + cum_sum
(12)                        cum_sum += n_masters_loc
(8)                     mask_slaves = np.logical_not(mask_masters)
(8)                     slaves = edge_elems[mask_slaves]
(8)                     slaves_masters = edge_neighbors[mask_slaves]
(8)                     diff_table = np.abs(neighbors[slaves_masters, :] -
(28)                                        np.outer(slaves, np.ones(3, dtype=np.int32)))
(8)                     slave_masters_apex = np.argmin(diff_table, axis=1)
(8)                     slaves_apex = edge_apexes[mask_slaves]
(8)                     new_pt_midside[slaves, slaves_apex] = new_pt_midside[
(12)                        slaves_masters, slave_masters_apex]
(8)                     child_triangles = np.empty([ntri*4, 3], dtype=np.int32)
(8)                     child_triangles[0::4, :] = np.vstack([
(12)                        new_pt_corner[:, 0], new_pt_midside[:, 0],
(12)                        new_pt_midside[:, 2]]).T
(8)                     child_triangles[1::4, :] = np.vstack([
(12)                        new_pt_corner[:, 1], new_pt_midside[:, 1],
(12)                        new_pt_midside[:, 0]]).T
(8)                     child_triangles[2::4, :] = np.vstack([
(12)                        new_pt_corner[:, 2], new_pt_midside[:, 2],
(12)                        new_pt_midside[:, 1]]).T
(8)                     child_triangles[3::4, :] = np.vstack([
(12)                        new_pt_midside[:, 0], new_pt_midside[:, 1],
(12)                        new_pt_midside[:, 2]]).T
```

```
209: (8)                    child_triangulation = Triangulation(refi_x, refi_y, child_triangles)
210: (8)                    if triangulation.mask is not None:
211: (12)                       child_triangulation.set_mask(np.repeat(triangulation.mask, 4))
212: (8)                    if ancestors is None:
213: (12)                       return child_triangulation
214: (8)                    else:
215: (12)                       return child_triangulation, np.repeat(ancestors, 4)
```

File 239 - _tritools.py:

```
1: (0)              """
2: (0)              Tools for triangular grids.
3: (0)              """
4: (0)              import numpy as np
5: (0)              from matplotlib import _api
6: (0)              from matplotlib.tri import Triangulation
7: (0)              class TriAnalyzer:
8: (4)                  """
9: (4)                  Define basic tools for triangular mesh analysis and improvement.
10: (4)                 A TriAnalyzer encapsulates a `.Triangulation` object and provides basic
11: (4)                 tools for mesh analysis and mesh improvement.
12: (4)                 Attributes
13: (4)                 ----------
14: (4)                 scale_factors
15: (4)                 Parameters
16: (4)                 ----------
17: (4)                 triangulation : `~matplotlib.tri.Triangulation`
18: (8)                     The encapsulated triangulation to analyze.
19: (4)                 """
20: (4)                 def __init__(self, triangulation):
21: (8)                     _api.check_isinstance(Triangulation, triangulation=triangulation)
22: (8)                     self._triangulation = triangulation
23: (4)                 @property
24: (4)                 def scale_factors(self):
25: (8)                     """
26: (8)                     Factors to rescale the triangulation into a unit square.
27: (8)                     Returns
28: (8)                     -------
29: (8)                     (float, float)
30: (12)                        Scaling factors (kx, ky) so that the triangulation
31: (12)                        ``[triangulation.x * kx, triangulation.y * ky]``
32: (12)                        fits exactly inside a unit square.
33: (8)                     """
34: (8)                     compressed_triangles = self._triangulation.get_masked_triangles()
35: (8)                     node_used = (np.bincount(np.ravel(compressed_triangles),
36: (33)                                             minlength=self._triangulation.x.size) != 0)
37: (8)                     return (1 / np.ptp(self._triangulation.x[node_used]),
38: (16)                            1 / np.ptp(self._triangulation.y[node_used]))
39: (4)                 def circle_ratios(self, rescale=True):
40: (8)                     """
41: (8)                     Return a measure of the triangulation triangles flatness.
42: (8)                     The ratio of the incircle radius over the circumcircle radius is a
43: (8)                     widely used indicator of a triangle flatness.
44: (8)                     It is always ``<= 0.5`` and ``== 0.5`` only for equilateral
45: (8)                     triangles. Circle ratios below 0.01 denote very flat triangles.
46: (8)                     To avoid unduly low values due to a difference of scale between the 2
47: (8)                     axis, the triangular mesh can first be rescaled to fit inside a unit
48: (8)                     square with `scale_factors` (Only if *rescale* is True, which is
49: (8)                     its default value).
50: (8)                     Parameters
51: (8)                     ----------
52: (8)                     rescale : bool, default: True
53: (12)                        If True, internally rescale (based on `scale_factors`), so that
the
54: (12)                        (unmasked) triangles fit exactly inside a unit square mesh.
55: (8)                     Returns
56: (8)                     -------
```

```
(8)                     masked array
(12)                        Ratio of the incircle radius over the circumcircle radius, for
(12)                        each 'rescaled' triangle of the encapsulated triangulation.
(12)                        Values corresponding to masked triangles are masked out.
(8)                     """
(8)                     if rescale:
(12)                        (kx, ky) = self.scale_factors
(8)                     else:
(12)                        (kx, ky) = (1.0, 1.0)
(8)                     pts = np.vstack([self._triangulation.x*kx,
(25)                                     self._triangulation.y*ky]).T
(8)                     tri_pts = pts[self._triangulation.triangles]
(8)                     a = tri_pts[:, 1, :] - tri_pts[:, 0, :]
(8)                     b = tri_pts[:, 2, :] - tri_pts[:, 1, :]
(8)                     c = tri_pts[:, 0, :] - tri_pts[:, 2, :]
(8)                     a = np.hypot(a[:, 0], a[:, 1])
(8)                     b = np.hypot(b[:, 0], b[:, 1])
(8)                     c = np.hypot(c[:, 0], c[:, 1])
(8)                     s = (a+b+c)*0.5
(8)                     prod = s*(a+b-s)*(a+c-s)*(b+c-s)
(8)                     bool_flat = (prod == 0.)
(8)                     if np.any(bool_flat):
(12)                        ntri = tri_pts.shape[0]
(12)                        circum_radius = np.empty(ntri, dtype=np.float64)
(12)                        circum_radius[bool_flat] = np.inf
(12)                        abc = a*b*c
(12)                        circum_radius[~bool_flat] = abc[~bool_flat] / (
(16)                            4.0*np.sqrt(prod[~bool_flat]))
(8)                     else:
(12)                        circum_radius = (a*b*c) / (4.0*np.sqrt(prod))
(8)                     in_radius = (a*b*c) / (4.0*circum_radius*s)
(8)                     circle_ratio = in_radius/circum_radius
(8)                     mask = self._triangulation.mask
(8)                     if mask is None:
(12)                        return circle_ratio
(8)                     else:
(12)                        return np.ma.array(circle_ratio, mask=mask)
(4)                 def get_flat_tri_mask(self, min_circle_ratio=0.01, rescale=True):
(8)                     """
(8)                     Eliminate excessively flat border triangles from the triangulation.
(8)                     Returns a mask *new_mask* which allows to clean the encapsulated
(8)                     triangulation from its border-located flat triangles
(8)                     (according to their :meth:`circle_ratios`).
(8)                     This mask is meant to be subsequently applied to the triangulation
(8)                     using `.Triangulation.set_mask`.
(8)                     *new_mask* is an extension of the initial triangulation mask
(8)                     in the sense that an initially masked triangle will remain masked.
(8)                     The *new_mask* array is computed recursively; at each step flat
(8)                     triangles are removed only if they share a side with the current mesh
(8)                     border. Thus, no new holes in the triangulated domain will be created.
(8)                     Parameters
(8)                     ----------
(8)                     min_circle_ratio : float, default: 0.01
(12)                        Border triangles with incircle/circumcircle radii ratio r/R will
(12)                        be removed if r/R < *min_circle_ratio*.
(8)                     rescale : bool, default: True
(12)                        If True, first, internally rescale (based on `scale_factors`) so
(12)                        that the (unmasked) triangles fit exactly inside a unit square
(12)                        mesh.  This rescaling accounts for the difference of scale which
(12)                        might exist between the 2 axis.
(8)                     Returns
(8)                     -------
(8)                     array of bool
(12)                        Mask to apply to encapsulated triangulation.
(12)                        All the initially masked triangles remain masked in the
(12)                        *new_mask*.
(8)                     Notes
(8)                     -----
(8)                     The rationale behind this function is that a Delaunay
```

```
126: (8)                    triangulation - of an unstructured set of points - sometimes contains
127: (8)                    almost flat triangles at its border, leading to artifacts in plots
128: (8)                    (especially for high-resolution contouring).
129: (8)                    Masked with computed *new_mask*, the encapsulated
130: (8)                    triangulation would contain no more unmasked border triangles
131: (8)                    with a circle ratio below *min_circle_ratio*, thus improving the
132: (8)                    mesh quality for subsequent plots or interpolation.
133: (8)                    """
134: (8)                    ntri = self._triangulation.triangles.shape[0]
135: (8)                    mask_bad_ratio = self.circle_ratios(rescale) < min_circle_ratio
136: (8)                    current_mask = self._triangulation.mask
137: (8)                    if current_mask is None:
138: (12)                       current_mask = np.zeros(ntri, dtype=bool)
139: (8)                    valid_neighbors = np.copy(self._triangulation.neighbors)
140: (8)                    renum_neighbors = np.arange(ntri, dtype=np.int32)
141: (8)                    nadd = -1
142: (8)                    while nadd != 0:
143: (12)                       wavefront = (np.min(valid_neighbors, axis=1) == -1) &
~current_mask
144: (12)                       added_mask = wavefront & mask_bad_ratio
145: (12)                       current_mask = added_mask | current_mask
146: (12)                       nadd = np.sum(added_mask)
147: (12)                       valid_neighbors[added_mask, :] = -1
148: (12)                       renum_neighbors[added_mask] = -1
149: (12)                       valid_neighbors = np.where(valid_neighbors == -1, -1,
150: (39)                                                  renum_neighbors[valid_neighbors])
151: (8)                    return np.ma.filled(current_mask, True)
152: (4)                def _get_compressed_triangulation(self):
153: (8)                    """
154: (8)                    Compress (if masked) the encapsulated triangulation.
155: (8)                    Returns minimal-length triangles array (*compressed_triangles*) and
156: (8)                    coordinates arrays (*compressed_x*, *compressed_y*) that can still
157: (8)                    describe the unmasked triangles of the encapsulated triangulation.
158: (8)                    Returns
159: (8)                    -------
160: (8)                    compressed_triangles : array-like
161: (12)                       the returned compressed triangulation triangles
162: (8)                    compressed_x : array-like
163: (12)                       the returned compressed triangulation 1st coordinate
164: (8)                    compressed_y : array-like
165: (12)                       the returned compressed triangulation 2nd coordinate
166: (8)                    tri_renum : int array
167: (12)                       renumbering table to translate the triangle numbers from the
168: (12)                       encapsulated triangulation into the new (compressed) renumbering.
169: (12)                       -1 for masked triangles (deleted from *compressed_triangles*).
170: (8)                    node_renum : int array
171: (12)                       renumbering table to translate the point numbers from the
172: (12)                       encapsulated triangulation into the new (compressed) renumbering.
173: (12)                       -1 for unused points (i.e. those deleted from *compressed_x* and
174: (12)                       *compressed_y*).
175: (8)                    """
176: (8)                    tri_mask = self._triangulation.mask
177: (8)                    compressed_triangles = self._triangulation.get_masked_triangles()
178: (8)                    ntri = self._triangulation.triangles.shape[0]
179: (8)                    if tri_mask is not None:
180: (12)                       tri_renum = self._total_to_compress_renum(~tri_mask)
181: (8)                    else:
182: (12)                       tri_renum = np.arange(ntri, dtype=np.int32)
183: (8)                    valid_node = (np.bincount(np.ravel(compressed_triangles),
184: (34)                                             minlength=self._triangulation.x.size) != 0)
185: (8)                    compressed_x = self._triangulation.x[valid_node]
186: (8)                    compressed_y = self._triangulation.y[valid_node]
187: (8)                    node_renum = self._total_to_compress_renum(valid_node)
188: (8)                    compressed_triangles = node_renum[compressed_triangles]
189: (8)                    return (compressed_triangles, compressed_x, compressed_y, tri_renum,
190: (16)                           node_renum)
191: (4)                @staticmethod
192: (4)                def _total_to_compress_renum(valid):
193: (8)                    """
```

```
194: (8)                    Parameters
195: (8)                    ----------
196: (8)                    valid : 1D bool array
197: (12)                       Validity mask.
198: (8)                    Returns
199: (8)                    -------
200: (8)                    int array
201: (12)                       Array so that (`valid_array` being a compressed array
202: (12)                       based on a `masked_array` with mask ~*valid*):
203: (12)                       - For all i with valid[i] = True:
204: (14)                         valid_array[renum[i]] = masked_array[i]
205: (12)                       - For all i with valid[i] = False:
206: (14)                         renum[i] = -1 (invalid value)
207: (8)                    """
208: (8)                    renum = np.full(np.size(valid), -1, dtype=np.int32)
209: (8)                    n_valid = np.sum(valid)
210: (8)                    renum[valid] = np.arange(n_valid, dtype=np.int32)
211: (8)                    return renum
```

File 240 - _version.py:

```
1: (0)              version = "3.9.2"
```

File 241 - SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRYCOMBINER_aligner_20_characters_for_pythons_codes.py:

```
1: (0)              import os
2: (0)              from datetime import datetime
3: (0)              def get_file_info(root_folder):
4: (4)                  file_info_list = []
5: (4)                  for root, dirs, files in os.walk(root_folder):
6: (8)                      for file in files:
7: (12)                         try:
8: (16)                             if file.endswith('.py'):
9: (20)                                 file_path = os.path.join(root, file)
10: (20)                                creation_time = 
datetime.fromtimestamp(os.path.getctime(file_path))
11: (20)                                modified_time = 
datetime.fromtimestamp(os.path.getmtime(file_path))
12: (20)                                file_extension = os.path.splitext(file)[1].lower()
13: (20)                                file_info_list.append([file, file_path, creation_time, 
modified_time, file_extension, root])
14: (12)                        except Exception as e:
15: (16)                            print(f"Error processing file {file}: {e}")
16: (4)                 file_info_list.sort(key=lambda x: (x[2], x[3], len(x[0]), x[4]))  # Sort 
by creation, modification time, name length, extension
17: (4)                 return file_info_list
18: (0)             def process_file(file_info_list):
19: (4)                 combined_output = []
20: (4)                 for idx, (file_name, file_path, creation_time, modified_time, 
file_extension, root) in enumerate(file_info_list):
21: (8)                     with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
22: (12)                        content = f.read()
23: (12)                        content = "\n".join([line for line in content.split('\n') if 
line.strip() and not line.strip().startswith("#")])
24: (12)                        content = content.replace('\t', '    ')
25: (12)                        processed_lines = []
26: (12)                        for i, line in enumerate(content.split('\n')):
27: (16)                            leading_spaces = len(line) - len(line.lstrip(' '))
28: (16)                            line_number_str = f"{i+1}: ({leading_spaces})"
29: (16)                            padding = ' ' * (20 - len(line_number_str))
30: (16)                            processed_line = f"{line_number_str}{padding}{line}"
31: (16)                            processed_lines.append(processed_line)
32: (12)                        content_with_line_numbers = "\n".join(processed_lines)
```

```
 33: (12)                        combined_output.append(f"File {idx + 1} - {file_name}:\n")
 34: (12)                        combined_output.append(content_with_line_numbers)
 35: (12)                        combined_output.append("\n" + "-"*40 + "\n")
 36: (4)                 return combined_output
 37: (0)             root_folder_path = '.'  # Set this to the desired folder
 38: (0)             file_info_list = get_file_info(root_folder_path)
 39: (0)             combined_output = process_file(file_info_list)
 40: (0)             output_file =
'SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt'
 41: (0)             with open(output_file, 'w', encoding='utf-8') as logfile:
 42: (4)                 logfile.write("\n".join(combined_output))
 43: (0)             print(f"Processed file info logged to {output_file}")

----------------------------------------
```